SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

**Zvonimir Vanjak**

# OKRUŽENJE ZA RJEŠAVANJE OPTIMIZACIJSKIH PROBLEMA

DOKTORSKA DISERTACIJA

Zagreb, 2006.

Doktorska disertacija je izrađena na **Zavodu za primijenjeno računarstvo Fakulteta elektrotehnike i računarstva u Zagrebu**

Mentor: **Prof. dr. sc. Vedran Mornar**

Disertacija ima 198 stranica

Disertacija br:

Povjerenstvo za ocjenu doktorske disertacije:

1. Dr.sc. Damir Kalpić, red. prof. FER Zagreb
2. Dr.sc. Vedran Mornar, red. prof. FER Zagreb
3. Dr.sc. Luka Neralić, red. prof. Ekonomskog fakulteta Zagreb

Povjerenstvo za obranu doktorske disertacije:

1. Dr.sc. Damir Kalpić, red. prof. FER Zagreb
2. Dr.sc. Vedran Mornar, red. prof. FER Zagreb
3. Dr.sc. Luka Neralić, red. prof. Ekonomskog fakulteta Zagreb
4. Dr.sc. Mario Kovač, red. prof. FER Zagreb
5. Dr.sc. Zoran Kalafatić, doc. FER-a Zagreb

Datum obrane disertacije: 27. studeni 2006.g.

Posvećujem

*roditeljima Dariji i Tomislavu*,

koji su mi omogućili da ostvarim svoje snove

*i uspomeni na brata Hrvoja (1976 – 2006)*

koji me učinio boljim čovjekom

# Zahvala

Koliko god izrada doktorske disertacije zahtijevala mnogobrojne samotne sate promišljanja i proučavanja, pisanja i ispravljanja, u njene temelje je nedvojbeno ugrađen i veliki doprinos mojih učitelja, profesora, suradnika, prijatelja i članova obitelji.

Najprije bih se htio zahvaliti svom stricu Andriji Vanjaku. Zahvaljujući njegovom entuzijazmu u vođenju astronomske sekciju u Osnovnoj školi Sukošan nepovratno i beznadežno sam se zaljubio u znanost i stekao trajnu fascinaciju tajnama svemira koji nas okružuje. Upravo tada je zasađena klica znatiželje i ljubavi prema istraživanju koja je u konačnici rezultirala ovom disertacijom, i zbog toga ti striče, od srca hvala.

Profesoru Nikici Simiću hvala na predanom i posvećenom radu u sekciji za fiziku zadarskog MIOC-a te poticanju i ohrabrivanju u tim pubertetskim godinama, koje su uz to bile temeljito poremećene ratom koji je harao Hrvatskom. Uspomene na rad, natjecanja i druženja s talentiranim mladim ljudima iz cijele Hrvatske će zauvijek ostati u mom srcu kao podsjetnik na sretne dane mladosti.

Kolegama sa Zavoda za primijenjeno računarstvo hvala na podršci i razumijevanju za moje, ponešto specifične, crte karaktera. Usprkos svim problemima s kojima je suočeno hrvatsko visoko obrazovanje, Zavod je bio i ostao izuzetno ugodna okolina za rad i istraživanje.

A iskoristiti ću ovu priliku i zahvaliti se i svojim prvim računalima. Iako su već odavno postali dio povijesti, Commodore VC-20 i Atari 1040ST su bili moji najbolji, a često i jedini suputnici u počecima savladavanja tajni računarstva.

I na kraju, ali, naravno, nikako ne i najmanje važno, najveće hvala ide mojoj obitelji bez čije potpore i podrške ništa od ovoga ne bi bilo moguće. Roditelji Darija i Tomislav, supruga Ivanka, te djeca Toni i Veronika su, svatko u svoje vrijeme i na svoj način, podnijeli dio tereta i pripomogli da se cijeli projekt uspješno privede kraju.

# Sadržaj

# Uvod

Svakim danom smo svjedoci sve brže transformacije suvremenog društva u informacijsko društvo. Nesmiljena globalizacija mijenja povijesne obrasce ekonomskog i društvenog ponašanja prisiljavajući gospodarstva razvijenih zemalja na prijelaz u tercijarni sektor djelatnosti  Eksplozivan razvoj informacijske tehnologije dodatno pomaže i potiče taj prijelaz, te stoga činjenica da količina stvorenih i pohranjenih informacija u svijetu raste za 50 % svake godine [Lyman2003] i nije iznenađujuća. Iako količina stvorenih informacija u „analognom“ obliku, prvenstveno u obliku dokumenata na papiru, raste i dalje, čineći ured bez papira (engl. *paperless office*) naizgled sve teže dostižnim ciljem, napredak u informacijskoj tehnologiji zajedno s pripadajućim smanjivanjem troškova obrade informacija, predstavljaju moćan poticaj za prijelaz iz analogno-papirnate u digitalnu informacijsku domenu.

Raznoliki čimbenici čine tu tranziciju sve lakšom i isplativijom, a kao najvažnije možemo identificirati: sve brže i jeftinije računalno sklopovlje, sve kvalitetnije programske alate za razvoj aplikacija te sve bolje mogućnosti međusobne komunikacije informacijskih sustava pri čemu Internet ima presudnu ulogu. Promatrajući uže područje računarske znanosti vezano uz izgradnju programskih sustava, gdje hardver i komunikacijske tehnologije promatramo kao bazične resurse uz pomoć kojih gradimo cjelokupni sustav, može se reći da je razvoj sustava za upravljanje relacijskim bazama podataka kao što su npr. DB2, Oracle i SQL Server s jedne,  te programskih tehnologija kao što su npr. .NET i J2EE okruženja s pripadajućim tehnologijama: ADO.NET, JavaBeans, EJB i web servisi, koje su sve moćnije i jednostavnije za korištenje s druge strane, omogućio izgradnju sve složenijih sustava za pohranu i obradu informacija, kako u korporativnoj tako i u javnoj domeni.

Razvijeni sustavi udruženi s mogućnošću komunikacije preko Interneta stavljaju informaciju na dohvat ruke, a sama tehnologija otvara mnogobrojne nove mogućnosti iako valja priznati da mnoge od njih nikada ne ostvare očekivanja. Ugradnja ERP (engl. *Enterprise Resource Planning*) i CRM (engl. *Customer Relationship Management*) sustava u gospodarske subjekte vodi ka kvalitetnijem planiranju i praćenju poslovanja, a rađanje i brzi rast područja dubinske analize podataka (engl. *data mining*), gdje se istraživanjem sve složenijih odnosa među podacima nastoji što bolje iskoristiti dostupne informacije predstavlja prirodnu evoluciju u cjelokupnom procesu informatizacije.

Međutim, prebacivanje informacija u digitalni oblik uz laku mogućnost dohvata, pretraživanja i prikaza predstavlja tek prvi korak u iskorištavanju informacijske tehnologije. Kao krajnji cilj se nameće mogućnost analize dostupnih informacija radi što bolje potpore donošenju odluka, ponovno, bilo u korporativnoj ili javnoj domeni. Tehnike dubinske analize podataka predstavljaju primjer takvog sustava i kad su ispravno ugrađene u informacijski sustav, pružaju značajnu pomoć u procesu poslovnog odlučivanja, poglavito u sektorima bankarstva i trgovine gdje se generira velika količina informacija te je potrebno primijeniti složene tehnike analize kako bi se uočili relevantni uzorci među podacima. Ali, postoji značajan skup problema koji se javljaju u praksi, a koji nisu prikladni za rješavanje tehnikama dubinske analize podataka.

Primjerice, „digitalizacija“ cjelokupnog procesa naručivanja i isporuke robe za lokalne trgovine s veleskladišta, u smislu informatizacije cijelog sustava i pripadnih procesa, donosi sama po sebi značajan doprinos u vidu boljeg praćenja i kontrole stanja skladišta, ali nije od velike pomoći kod donošenja odluke kako će se naručena roba prevesti do lokalnih trgovina koristeći ograničene resurse prijevoza, konkretne kombije i kamione, a zadovoljavajući pritom različita ograničenja. Isto tako, informatizacija visokih učilišta putem projekta ISVU [Baranović2003] sama po sebi donosi značajan napredak u mogućnostima interaktivnog praćenja različitih statistika vezanih uz nastavu. Međutim, rješavanje problema definiranja satnice, tj. određivanje koji nastavnik predaje koji predmet, kojim grupama studenata, u koje vrijeme i u kojoj dvorani, jednom kad zaživi Bolonjski proces zajedno s pridruženim personaliziranjem nastave i većim mogućnostima izbora za studente organizirane u manje nastavne grupe, biti će prilično težak problem, pogotovo imajući u vidu vrlo ograničene prostorne i ljudske resurse na većini naših fakulteta.

Rješavanjem takvih problema bavi se optimiranje. Optimiranje možemo definirati kao „postupak kojim se pri projektiranju ili planiranju, u ekonomiji, tehničkim i prirodnim znanostima, ostvaruje ili određuje najbolji mogući izbor ekonomskih i/ili tehničkih veličina na temelju prethodno određenih kriterija“ [Anić1999]. Problem optimiranja se javlja u raznim granama znanosti kao što su optimiranje funkcija u matematici (problem pronalaženja minimuma ili maksimuma), različite vrste matematičkog programiranja (linearno, kvadratično i nelinearno), kombinatorno optimiranje kao grana računarske znanosti i još mnoge druge. Iako svaka vrsta optimiranja ima svoje specifičnosti, poglavito vezane uz pripadajuću strukturu problema, svima im je zajedničko postojanje precizno definiranog problema i određenog skupa optimizacijskih postupaka ili metoda koje su primjenjive na taj problem.

Metode optimiranja su u zadnjih petnaestak godina doživjele nagli razvoj, zahvaljujući ponajviše dvama čimbenicima. Prvi važan čimbenik je rapidno povećanje dostupne računarske snage (Moore-ov zakon), što je omogućilo širu primjenjivost optimizacijskih metoda u IT zajednici i rješavanje sve složenijih problema. To je bio nužan preduvjet ukoliko se ima u vidu „glad za gigahercima“ koju pokazuje većina optimizacijskih metoda. Drugi bitan faktor čini razvoj heurističkih algoritama kao što su genetički algoritmi, evolucijske strategije, simulirano kaljenje, koji iako ne garantiraju nalaženje globalno optimalnog rješenja problema, pokazuju vrlo dobre rezultate u nalaženju približno optimalnog rješenja, ali u bitno kraćem vremenu. Mnogobrojni primjeru iz prakse koji su bili tehnički nerješivi tradicionalnim matematičkim egzaktnim tehnikama, ili su se rješavali određenim aproksimacijama, korištenjem npr. linearnog programiranja, primjenom heurističkih algoritama odjednom postaju rješivi u smislu da se može doći do kvalitetnih rješenja problema. Može se reći da je pojava heurističkih metoda optimiranja dala novi poticaj raspravi o osnovnom pitanju optimiranja: „Je li bolje točno riješiti približan problem, ili približno riješiti pravi problem ?“

Za razliku od egzaktnih matematičkih metoda, npr. različitih vrsta matematičkog programiranja, algoritama iz teorije mreža i grafova, koje imaju dugu tradiciju i dobro su teorijski utemeljene, heurističke metode su nešto novijeg datuma, ali su ih uspjesi u rješavanju teških optimizacijskih problema istaknuli u središte pozornosti. Međutim,

iako je napredak u razvoju pojedinačnih algoritama bio vrlo brz, čemu svjedoči proliferacija različitih heurističkih optimizacijskih postupaka te mnogobrojne konferencije i stručni časopisi posvećene tim temama, kao i potaknut činjenicom da je sama programska ugradnja osnovnih varijanti heurističkih metoda značajno jednostavnija u odnosu na konkurentne egzaktne metode, mora se naglasiti da u toj lijepoj slici postoje i dvije sive točke.

Jedna je vezana uz još nedovoljno razvijenu teoriju heurističkih optimizacijskih postupaka. Uzevši za primjer genetičke algoritme, može se sa sigurnošću reći da je teorija u zaostatku za praksom. Osim pojedinačnih detaljnih analiza primjena genetičkih algoritama na jednostavne probleme, što znači na način da se dobiveni teorijski rezultati o radu algoritma mogu provjeriti eksperimentom, *modus operandi* genetičkih algoritama je uvelike neistraženo područje. Ovo nije neočekivano kad se uzme u obzir širina njihove primjene i različitosti reprezentacija s kojima mogu raditz, a što značajno otežava općenitu teorijsku analizu. Naravno, ovdje se pod *modus operandi* misli na precizan matematički opis rada algoritma nad dobro definiranim matematičkim strukturama. Intuitivni razlozi zbog kojih je „jasno“ da bi genetički algoritmi trebali biti kvalitetan optimizacijski postupak su prilično čvrsti, i za većinu istraživača posve dovoljni. Međutim, rezultat toga je velik broj članaka koji istraživanju heurističkih postupaka pristupaju isključivo s eksperimentalnog stanovišta (ovdje je zgodno napomenuti da je kod egzaktnih matematičkih postupaka situacija gotovo posve obrnuta). Osmisli se neka varijacija na temu postojećeg algoritma, ili neka nova vrsta operatora, ili neka nova reprezentacija, itd.., *ad-hoc* se izgradi programska podrška, usporede se rezultati s postojećim algoritmima, i ako su bolji, eto jednog malog doprinosa. Mora se naglasiti da su upravo zahvaljujući velikom broju takvih malih koraka genetički algoritmi napredovali u statusu i danas predstavljaju respektabilni dio arsenala svakog istraživača optimizacijskih problema. Ipak, ostaje kao cilj bolje razumijevanje teorijske podloge rada genetičkih algoritama, odnosno heurističkih algoritama općenito, iz čega bi trebao proizaći daljnji napredak u kvaliteti njihovog rada.

Međutim, ukoliko se uzme u obzir da teorijsko proučavanje heurističkih algoritama zahtijeva primjenu vrlo naprednih matematičkih tehnika kao što su diskretna i stohastička matematika, Markovljevi lanci, itd., a da je programska izvedba različitih varijanti heurističkih algoritama relativno jednostavna budući da se određeni izbori kod izgradnje algoritma mogu obrazložiti intuitivno, ako već ne i teorijski, jasno je da se odnos snaga u natjecanju teorije i pokusa neće tako skoro radikalno promijeniti.

U tom kontekstu na vidjelo izlazi prije spomenuta druga siva točka na području heurističke optimizacije. Naime, eksperimentalno istraživanje i uspoređivanje različitih algoritama zahtijeva dosta ulaganja u njihovu izgradnju, definiranje testnih problema, izgradnju pomoćne programske infrastrukture za usporedbu i vizualizaciju rezultata te obavljanje mnogih aktivnosti koje nisu izravno povezane s izgradnjom što kvalitetnijih optimizacijskih postupaka, a što bi trebala biti osnovna specijalnost istraživača u tom području. Nepostojanje kompletnih programskih okruženja koja bi omogućavala jednostavnu primjenu heurističkih metoda na širok skup problema, bilo u praktične ili istraživačke-znanstvene svrhe stoga predstavlja veliki problem za istraživače u cjelini.

Iz tog razloga su kod primjene određene optimizacijske metode na dani problem istraživači vrlo često prisiljeni krenuti od nule, uz eventualno korištenje neke rudimentarne biblioteke razreda. Iako postoje i sofisticirane biblioteke razreda s ugrađenim mnogobrojnim optimizacijskim algoritmima (npr. GALib, EOLib, JEO), od kojih neke predstavljaju i kompletno okruženje za provođenje optimizacije (npr. HeuristicLab), čest nedostatak tih rješenja je slaba upotrebljivost i s tim povezana visoka razina programerskog znanja potrebna za primjenu. Najčešće su izgrađena korištenjem jezika C++, uz napredne tehnike predložaka i generičkog programiranja, a „optimizacijsko okruženje“ najčešće čini C++ prevodilac. Uz to navedena okruženja pate i od problema nedovoljne fleksibilnosti i proširljivosti s obzirom da su uglavnom predviđena za korištenje samo nekih optimizacijskih metoda, npr. genetičkih algoritama, te da vrlo rijetko imaju ugrađene mogućnosti za vizualizaciju i analizu dobivenih rješenja. Kad se ima u vidu heuristički karakter optimiranja, upravo nedostatak ugrađenih mogućnosti za vizualizaciju i usporedbu rješenja predstavlja veliku boljku s obzirom da eksperimentiranje s različitim vrstama algoritama, operatora i reprezentacija varijabli predstavlja važnu i vremenski zahtjevnu aktivnost prilikom primjene heurističkih metoda. Stoga se nameće potreba za razvojem okruženja za rješavanje optimizacijskih problema sa sljedećim karakteristikama:

- prijaznost za korištenje – razvijeno programsko rješenje mora biti neposredno iskoristivo, odnosno da dolazi s ugrađenim skupom problema, algoritama, reprezentacija i operatora koji su široko primjenjivi i omogućuju daljnju nadgradnju
- proširljivost i iskoristivost – jednostavno dodavanje novih vrsta problema, algoritama, operatora i prikaza rješenja (varijabli), a s njima i pripadajućih programskih komponenti za vizualizaciju i analizu rezultata
- jednostavnost – izgradnja novih komponenata mora biti maksimalno jednostavna sa stajališta primijenjenih tehnika programiranja
- efikasnost izvršavanja – ugradnja navedenih mogućnosti mora imati minimalan utjecaj na efikasnost numeričkog izračunavanja

Izgradnja programskog okruženja za optimiranje s navedenim karakteristikama je osnovni cilj ove doktorske disertacije a njen originalni znanstveni doprinos čine:

- klasifikacija koncepata iz domene optimizacijskih problema
- izrada objektnog modela za rješavanje optimizacijskih problema pomoću iterativnih heurističkih algoritama
- identifikacija međusobnih interakcija optimizacijskih koncepata
- oblikovanje univerzalnog prezentacijskog sloja za unaprjeđenje primjenjivosti postupaka operacijskih istraživanja.

Rad je podijeljen u tri osnovne cjeline. Prvu cjelinu čine prva tri poglavlja u kojima je dan teorijski pregled područja optimizacije zajedno sa osvrtom na postojeća optimizacijska okruženja. U prvom poglavlju su opisani različiti optimizacijski problemi te je provedena njihova klasifikacija s ciljem definiranja osnovnih elemenata konceptualnog modela domene koji će predstavljati podlogu za kasniju izgradnju

optimizacijskog okruženja ESOP (***E**nvironment for **S**olving **O**ptimization **P**roblems*). U drugom poglavlju je s istim ciljem opisana klasifikacija optimizacijskih postupaka te su opisani pojedini široko korišteni heurisitčki optimizacijski postupci. Treće poglavlje je posvećeno temama vezanim uz programsku podršku za optimizaciju. Opisani su različiti mogući scenariji optimizacije sa stajališta iskoristivosti dostupne programske podrške, dan je popis traženih karakteristika općenitog optimizacijskog okruženja te su analizirana neka postojeća programska rješenja (optimizacijska okruženja i biblioteke).

Drugu cjelinu čine četvrto i peto poglavlje i u njima je opisana izgradnja ESOP optimizacijskog okruženja u .NET razvojnom okruženju. U skladu s procesom razvoja programske podrške baziranom na analizi domene problema (engl. *domain driven design*) najprije je u četvrtom poglavlju opisan konceptualni model domene optimizacije. Dan je kratki pregled primijenjene metodologije razvoja programske podrške nakon čega su analizirani pojedini elementi definiranog konceptualnog modela koji proizlaze iz teorijske klasifikacije provedene u prvom dijelu disertacije, te su opisane njihove karakteristike i međusobne interakcije. U petom poglavlju je opisana sama izgradnja ESOP optimizacijskog okruženja, u klasičnom smislu opisa dizajna i implementacije. Najprije je detaljno opisan objektni model izrađenog okvira za razvoj (engl. *framemwork*) kroz opis uloga, odgovornosti i kolaboracija pojedinih objekata/razreda a zatim je opisana i izgradnja samog ESOP okruženja. Na kraju poglavlja su prikazane mogućnosti primjene ESOP okruženja na rješavanje standardnih optimizacijskih problema, a dan je i prikaz mogućnosti proširenja ESOP-a.

Treću cjelinu disertacije čini šesto poglavlje u kojem je opisana primjena ESOP optimizacijskog okruženja na problem optimiranja parametara mehaničke strukture brodske konstrukcije.

# 1. Općenito o problemu optimizacije

U uvodu je optimizacija definirana kao „postupak kojim se pri projektiranju ili planiranju (u ekonomiji, tehničkim i prirodnim znanostima) ostvaruje (određuje) najbolji mogući izbor ekonomskih i/ili tehničkih veličina na temelju prethodno određenih kriterija“.

Iako ova definicija sažeto i jasno opisuje bit optimizacije, prije pristupanja opisu i izgradnji optimizacijskog okruženja koje će zadovoljavati zahtjeve navedene u uvodu potrebno je detaljnije opisati i raščlaniti koncepte koji se pojavljuju kod primjene optimizacijskih postupaka. Analizom gornje definicije dolazimo do sljedećih bitnih karakteristika optimizacije.

### 1. Postoji skup varijabli problema

Kao prvo, postoji skup veličina čije optimalne vrijednosti s obzirom na neki kriterij želimo odrediti. Brojnost i pojedinačne karakteristike tih veličina proizlaze iz definicije problema koji se rješava i mogu biti vrlo raznoliki. Od jednostavnih realnih varijabli kod problema optimiranja matematičkih funkcija do složenih struktura permutacije i pridruživanja kod različitih kombinatorijalnih problema. Provođenje postupka optimiranja zahtijeva predstavljanje svake od tih veličina apstraktnim matematičkim objektom – varijablom, a tako definiran skup varijabli određuje prostor rješenja unutar kojega će se pomoću optimizacijskog postupka tražiti optimalno rješenje.

### 2. Definirani su kriteriji optimizacije

Druga bitna karakteristika optimizacije je postojanje skupa kriterija koji na osnovu vrijednosti varijabli problema određuje kvalitetu izbora, odnosno određuje relativni odnos kvalitete dvaju mogućih rješenja, koje je bolje a koje lošije. Ovdje je bitno napomenuti da kvalitativno postoje dvije vrste kriterija: jedni koji određuju optimalnost danog rješenja kroz neku definiranu mjeru, najčešće određenu vrijednošću funkcije/a cilja, i drugi koji predstavljaju ograničenja na problem i određuju je li dano rješenje uopće moguće (engl. *feasible*). Kod obje skupine kriterija vrlo je važna kardinalnost definiranog skupa.

Postojanje samo jednog definiranog kriterija optimalnosti znači da je cilj optimizacije nalaženje globalnog optimuma danog problema. U slučaju nekoliko definiranih kriterija, cilj optimizacije više nije nalaženje jednoznačno definiranog globalnog optimuma, već je cilj naći Pareto skup svih nedominiranih rješenja (rješenja koja nije moguće poboljšati po niti jednom kriteriju a da se istovremeno ne pogoršaju po nekom drugom kriteriju).

Kod kriterija vezanih uz ograničenja važno je samo postoje li ograničenja ili ne. Ukoliko nema definiranih ograničenja radi se o jednostavnijem problemu ne-ograničene (engl. *unconstrained*) optimizacije, dok prisutnost ograničenja u optimizaciji, pri čemu je brojnost ograničenja prvenstveno kvantitativna a ne kvalitativna karakteristika, u značajnoj mjeri komplicira provođenje optimizacije zbog nužnosti prilagodbe optimizacijskog postupka.

### 3. Postoji optimizacijski postupak primjenjiv na zadani problem

Treća bitna karakteristika optimizacije proizlazi iz činjenice da provođenje optimizacije zahtijeva postojanje optimizacijskog postupka (u pravilu realiziranog putem računalnog algoritma) koji će kroz definirani niz operacija doći do što boljeg rješenja. Po principu rada optimizacijske postupke ugrubo možemo podijeliti na egzaktne i heurističke. Egzaktne metode imaju precizno definiran slijed koraka koji, ukoliko su zadovoljeni određeni matematički postavljeni uvjeti garantirano vodi k optimalnom rješenju. S druge strane, heurističke metode se oslanjaju na tehniku iterativnog poboljšavanja rješenja i uglavnom ne garantiraju nalaženje globalno optimalnog rješenja[1]. Na osnovu navedenoga bi se moglo (pogrešno) zaključiti kako su egzaktne metode apsolutno prvi izbor. Razlog zašto tome nije tako leži u činjenici da njihova primjena u praksi često nije moguća zbog dugotrajnog vremena izvršavanja koje s povećanjem složenosti problema često eksponencijalno raste. Iako bismo svaki problem teoretski mogli riješiti potpunim pretraživanjem (enumeracijom) prostora rješenja, to je zbog kardinalnosti tog skupa praktično nemoguće već i za vrlo jednostavne probleme, čak i uz korištenje najbržeg danas dostupnog računala.

Pored toga, kod odabira i izgradnje optimizacijskih postupaka vrlo su važni čimbenici i širina njihove primjene (na koliki skup različitih problema se mogu primijeniti) te složenost implementacije (koliko je teško izgraditi pripadni računalni algoritam). Egzaktne metode su vrlo često primjenjive samo na točno određeni problem, rješavanje kojeg je najčešće i potaknulo pronalaženje dotičnog optimizacijskog postupka. S druge strane, heurističke metode imaju značajno šire područje primjene, uglavnom zahvaljujući tome što pripadni algoritmi operiraju nad veličinama (objektima) na višem nivou apstrakcije. Primjerice, genetički algoritmi su primjenjivi na bilo kakav optimizacijski problem – jedini uvjet je da se definira reprezentacija rješenja i operatori križanja i mutacije koji će operirati nad tako definiranom reprezentacijom s ciljem nalaženje boljeg rješenja. U pogledu složenosti izgradnje sa sigurnošću se može reći da je jednostavnost izgradnje većine heurističkih algoritama, kod kojih se kompetitivne implementacije često mogu izgraditi već u nekoliko stotina linija programskog kôda, jedan od važnih razloga, ako ne i najvažniji, njihove popularnosti.

Analizom navedenih karakteristika dolazimo do dva fundamentalna koncepta koja su uvijek prisutna u procesu optimiranja i na kojima se bazira izgradnja svakog općenitog optimizacijskog okruženja. Prvi je koncept optimizacijskog problema koji obuhvaća sve vezano uz opis problema koji se rješava (broj i vrsta varijabli, kriteriji optimizacije) dok drugi fundamentalni koncept predstavlja optimizacijski postupak koji određuje način potrage za rješenjem i koji se u implementaciji realizira putem računalnog algoritma.

Optimizacijski problem možemo definirati kao „izračunski (eng. *computational*) problem u kojem je cilj naći najbolje od svih mogućih rješenja. Formalnije, cilj je naći rješenje iz područja mogućih rješenja (eng. *feasible region*) koje ima minimalnu (ili

[1] Ovdje se mora naglasiti da postoje i heuristički algoritmi koji garantiraju nalaženje optimalnog rješenja. Primjerice, simulirano kaljenje uz korištenje *raporeda kaljenja* (eng. *annealing schedule*) kod kojeg se temperatura smanjuje po logaritamskoj ovisnosti garantira nalaženje optimuma, ali je problem što se vrijeme optimizacije eksponencijalno povećava te time poprima karakteristike egzaktne metode.

maksimalnu) vrijednost funkcije/a cilja“ [Atallah1998]. Nastavak ovog poglavlja je posvećen temama vezanim uz optimizacijske probleme, dok su različiti postupci (metode) optimizacije opisani u sljedećem poglavlju.

## 1.1 Optimizacijski problemi

Primjeri optimizacijskih problema su svuda oko nas. Neki od njih su jednostavni i većina ljudi ih automatski rješava, posve nesvjesna činjenice da pri tome provode postupak optimizacije. Tipični primjeri su određivanje puta (rute) kojom se ide automobilom na posao gdje je cilj što brži dolazak na posao, odabir dana u tjednu za obavljanje kupovine kad želimo što manju gužvu u trgovni, i pakiranje stvari u prtljažnik automobila kod odlaska na godišnji odmor u kojem slučaju je cilj uspješno slaganje sve prtljage u ograničeni prostor prtljažnika.

Iako naizgled trivijalni, slični problemi se pojavljuju i na mnogo većoj skali kada je nalaženje optimalnog rješenja puno teže te zahtjeva provođenje složenih optimizacijskih postupaka. Tipični primjeri sa vrlo širokim područjem primjene su:

- problem trgovačkog putnika (engl. *Traveling Salesman Problem* - TSP)

  Za dani skup gradova i definiran skup udaljenosti među pojedinim gradovima, naći takav raspored obilaska svih gradova kod kojeg će se prijeći minimalni put.

- problem raspoređivanja strojeva (engl. *Job Shop Scheduling Problem* - JSSP)

  Neka je dan skup strojeva, i neka je dan skup poslova od kojih se svaki u točno određenom redoslijedu mora obaviti na svakom stroju. Potrebno je naći takav redoslijed obavljanja poslova na strojevima kojim će se cijeli posao najbrže obaviti.

- problem određivanja rute vozila (engl. *Vehicle Routing Problem* - VRP)

  Neka su zadani sljedeći skupovi: skup skladišta na kojima se nalazi roba, skup trgovina s definiranim potrebnim količinama robe (narudžbama) koja se mora dovesti iz skladišta, skup puteva (ruta) kojima se može doći od skladišta do pojedine trgovine, i skup vozila sa određenim kapacitetom prijevoza. Tada je potrebno odrediti takav plan prijevoza robe od skladišta do trgovina koji će imati minimalan trošak uz zadovoljavanje svih zadanih ograničenja.

- problem naprtnjače (engl. *Knapsack* ili *Bin-packing* problem)

  Neka je dan skup predmeta, svaki sa određenom zapreminom i vrijednošću, koje je potrebno prenijeti u naprtnjači. Potrebno je odrediti koje predmete uzeti tako da svi stanu u naprtnjaču a da vrijednost ponesenih predmeta bude maksimalna.

Pored ovih klasičnih problema, koji imaju mnogobrojne varijacije i predstavljaju izvrsni poligon za testiranje svih optimizacijskih metoda, u praksi se javljaju i mnogobrojne druge klase problema - npr. različiti problemi pridruživanja: linearni, kvadratični, sa i bez ograničenja; problemi optimiranja proizvodnog procesa; problemi određivanja

geometrijskog rasporeda (engl. *layout problem*) i drugi. U praksi je vrlo čest i slučaj kombiniranih problema, npr. kombinacija *VRP* i *Bin-packing* problema kod razvoženja robe sa skladišta u trgovine. Svi navedeni problemi spadaju u klasu tzv. kombinatornih problema [Papadimitriou1982], koji su karakterizirani diskretnim strukturama podataka nad kojima su definirani i velikom algoritamskom složenošću postupaka rješavanja s obzirom da većina spada u klasu NP-teških problema [Garey1979].

Međutim, iako kombinatorijalni problemi spadaju među najzanimljivije i najproučavanije probleme, što zbog njihove široke primjenjivosti a što zbog izazova koji predstavlja njihovo rješavanje, oni ipak čine tek samo jedan dio područja matematičke optimizacije u kojem značajnu ulogu imaju i druge vrste problema. Široko poznati primjeri su: optimizacija skalarnih i vektorskih realnih funkcija u matematičkoj analizi i različite vrste matematičkog programiranja: linearno, kvadratično, cjelobrojno, nelinearno i stohastičko.

Raznolikost optimizacijskih problema kod izgradnje općenitog optimizacijskog okruženja predstavlja određenu poteškoću jer zahtijeva značajnu varijabilnost i fleksibilnost u konačnoj implementaciji optimizacijskog okruženja. Srećom, objektno-orijentirana paradigma koja će biti korištena kod izgradnje ESOP optimizacijskog okruženja omogućava efikasnu realizaciju traženih karakteristika preko programskih tehnika nasljeđivanja, polimorfizma i sučelja. Detaljni opis analize i izgradnje ESOP optimizacijskog okruženja je tema četvrtog i petog poglavlja dok će se u sljedećim potpoglavljima ovog poglavlja dati teorijska obrada optimizacijskih problema s naglaskom na elemente koji će biti važni kod kasnije izgradnje ESOP-a.

## 1.2 Kriteriji za klasifikaciju optimizacijskih problema

Idealno optimizacijsko okruženje bi bilo primjenjivo na rješavanje bilo kakvog optimizacijskog problema. Međutim, takav ideal je vrlo teško, ako ne i nemoguće dosegnuti. Stoga se prilikom izgradnje takvog okruženja mora odrediti širina njegove primjenjivosti, odnosno mora se odrediti skup problema koje će biti moguće ugraditi i riješiti unutar danog optimizacijskog okruženja.

Da bi se uopće moglo pristupiti rješavanju tog pitanja, potrebno je provesti klasifikaciju optimizacijskih problema. Prije navođenja kriterija klasifikacije, važno je naglasiti razliku između vrste problema i instance problema. Tako je TSP primjer dobro definirane vrste problema za koji postoje mnogobrojne instance, s različitim brojem gradova i njihovih međusobnih udaljenosti. Optimizacija skalarne realne funkcije definirane nad određenim brojem realnih varijabli također je vrsta problema za koju postoje mnogobrojne instance koje se razlikuju po rasponima varijabli i obliku funkcije cilja.

Ono što je bitno za proces klasifikacije jest da iz vrste problema proizlaze apstraktna svojstva samog problema koja se mogu softverski modelirati prilikom izgradnje optimizacijskog okruženja. Iako se optimizacijski problemi značajno razlikuju po svojim svojstvima, naslanjajući se na već definirane općenite karakteristike optimizacijskih problema možemo definirati sljedeću listu kriterija koji predstavljaju

podlogu za njihovu klasifikaciju a koji će u sljedećem potpoglavlju biti detaljnije obrađeni:

**1. Broj i vrsta funkcija cilja**

Po ovom kriteriju, optimizacijske probleme dijelimo na jednokriterijske (engl. *single objective*) i višekriterijske (engl. *multi-objective*), odnosno na probleme koji imaju definiran samo jedan kriterij optimalnosti (funkciju cilja) i one koji ih imaju više. Ovaj kriterij je od presudne važnosti kod implementacije optimizacijskog postupka s obzirom na različitosti u implementaciji algoritama koji za konačni cilj imaju jedan broj (jednokriterijski) u odnosu na one koji žele naći Pareto plohu (višekriterijski). Pored toga, za svaki kriterij je potrebno specificirati da li je cilj maksimizacija ili minimizacija pripadne funkcije cilja.

Specifičnu vrstu problema predstavljaju problemi zadovoljavanja ograničenja (eng. *constraint satisfaction problems*) koji nemaju funkciju cilja već je cilj optimizacije naći rješenje koje zadovoljava sva postavljena ograničenja. Ova vrsta problema se može svesti na jednokriterijski problem kojem je vrijednost funkcije cilja jednaka broju prekršenih ograničenja s konačnim ciljem minimizacije te vrijednosti na nulu.

**2. Broj i vrsta varijabli**

Iako svaki optimizacijski problem ima svoj specifičan skup varijabli koje predstavljaju ulazne veličine problema i čije optimalne vrijednosti želimo odrediti, klasifikaciju možemo provesti po vrstama varijabli u tom skupu: cjelobrojne i realne varijable, nizovi varijabli, permutacije, grafovi, matrice. S obzirom da se svaka varijabla prilikom izgradnje optimizacijskog algoritma mora reprezentirati nekom programskom strukturom koja je element programskog jezika u kojem se gradi optimizacijsko okruženje, kod izgradnje optimizacijskog okruženja je nužno klasificirati podržane vrste varijabli te modelirati njihove pripadajuće matematičke karakteristike preko odgovarajućih programskih struktura. Značaj vrsta varijabli nad kojima je definiran problem najbolje se vidi u postojanju zasebnih grana teorije optimizacije vezanih uz diskretne, kontinuirane i kombinatorne probleme, a prilikom same implementacije optimizacijskih postupaka tehnike pronalaženja rješenja moraju biti prilagođene definiranom skupu ulaznih varijabli problema.

**3. Prisutnost ograničenja**

Po ovom kriteriju, optimizacijske probleme dijelimo na one koji imaju definirana ograničenja koje konačno rješenje mora zadovoljavati i one koji takvih ograničenja nemaju. U slučaju da ne postoje definirana ograničenja, sva rješenja iz skupa rješenja su moguća i implementacija postupka (algoritma) koji će tražiti optimalno rješenje je značajno jednostavnija u odnosu na problem s definiranim ograničenjima.

**4. Prisutnost parametara problema**

Parametri problema također u određenom smislu predstavljaju varijable problema, ali se od ulaznih varijabli razlikuju po tome što su njihove vrijednosti za pojedinačno provođenje optimizacije konstantne. Prisutnost parametara problema omogućava provođenje analize osjetljivosti (istraživanje kako promjena parametara problema

utječe na optimalno rješenje) a mogućnost njenog jednostavnog provođenja predstavlja veliki plus za optimizacijsko okruženje.

### 1.2.1 Jednokriterijski i višekriterijski problemi

Svaki optimizacijski problem nužno mora imati definiran jedan ili više kriterija po kojem/kojima će se ocijeniti kvaliteta pojedinih rješenja, a koji su opisani preko matematičkih funkcija definiranih nad skupom ulaznih varijabli problema. U literaturi i praksi je uobičajeno takve kriterije nazivati funkcijama cilja (engl. *objective function*)

Kod jednokriterijskih problema, funkciju cilja definiramo kao preslikavanje $f : S \rightarrow R$, gdje S predstavlja Kartezijev produkt skupova nad kojima su definirane pojedinačne varijable problema. Efektivno, funkcija cilja za svaku moguću kombinaciju vrijednosti varijabli problema kao izlaz daje jedan (najčešće realni) broj koji predstavlja mjeru kvalitete rješenja. Ovisno o tome radi li se o problemu minimizacije ili maksimizacije, cilj optimizacije je naći takve vrijednosti ulaznih varijabli za koje će vrijednost funkcije cilja biti minimalna, odnosno maksimalna.

U slučaju optimizacijskog problema s više definiranih kriterija optimalnosti, odnosno s dvije ili više definiranih funkcija, cilj optimizacije je kompliciranije definirati. Naime, kod postojanja više kriterija po kojima se određuje kvaliteta nekog rješenja, malo je vjerojatno da će baš u točno jednom te istom rješenju biti istovremeno postignute optimalne vrijednosti svih funkcija cilja. Stoga se pojavljuje situacija u kojoj je jedno rješenje bolje po jednom kriteriju, a neko drugu rješenje je bolje s obzirom na neki drugi kriterij.

U takvoj situaciji je potrebno (re)definirati značenje optimuma. Najčešće korištena i najšire prihvaćena je definicija *Pareto optimuma* (iako ga je originalno predložio Francis Ysidro Edgeworth [Edgeworth1881], pojam je generalizirao Vilfredo Pareto [Pareto1896] po kome je i dobio ime). Ukoliko pretpostavimo da je cilj optimizacije minimizirati sve funkcije cilja, imamo sljedeću definiciju:

**Definicija:** Neka je $F$ skup svih mogućih rješenja optimizacijskog problema $P$. Tada kažemo da je rješenje $x^* \in F$ Pareto optimalno ukoliko ne postoji $x \in F$ takav da je $f_i(x) \leq f_i(x^*)$ za svaki $i = 1, .., k$ i $f_j(x) < f_j(x^*)$ za barem jedan $j$; gdje je $k$ ukupan broj kriterija, a $f_i(x)$ funkcija cilja koja odgovara $i$-tom kriteriju optimalnosti danog problema $P$ [Coello2001].

Ili, opisano riječima, rješenje je Pareto optimalno ukoliko ne postoji rješenje koje bi bilo strogo bolje po jednom kriteriju optimalnosti a da je istovremeno barem jednako dobro po svim ostalim kriterijima optimalnosti. Odmah vidimo da će, osim u slučaju kad postoji jedno rješenje koje je najbolje po svim kriterijima optimalnosti, postojati više Pareto optimalnih rješenja koja predstavljaju rješenje višekriterijskog optimizacijskog problema. Skup tih rješenja se naziva *Pareto skupom* i njegovo nalaženje predstavlja konačni cilj kod višekriterijske optimizacije. Kod višekriterijske optimizacije su još važni pojmovi dominiranog odnosno nedominiranog rješenja, pa imamo sljedeću definiciju:

**Definicija:** Za rješenje definirano vektorom $u = (u_1, ... , u_k)$ kažemo da dominira nad rješenjem $v = (v_1, ..., v_k)$ ako i samo ako vrijedi da je $u$ parcijalno manji od $v$, odnosno, vrijedi $u_i \leq v_i$ za svaki $i \in \{1, ..., k\}$ i postoji takav $i \in \{1, ..., k\}$ da vrijedi $u_i < v_i$. [Coello2001].

## 1.2.2 Varijable problema

Pored kriterija optimalnosti, skup varijabli nad kojima je definiran problem predstavlja drugu ključnu karakteristiku svakog optimizacijskog problema. Kako je već navedeno, veličine prisutne u konkretnom optimizacijskom problemu modeliramo pomoću varijabli. S obzirom da su veličine koje se pojavljuju u optimizacijskim problemima vrlo raznolike (od jednostavnih skalarnih veličina do veličina koje predstavljaju redoslijed obilaska gradova ili raspored tvornica na određene lokacije), a imajući u vidu da je krajnji cilj rješavanje optimizacijskih problema pomoću računalnih algoritama koji su kao i veliki dio teorije o računarstvu (engl. *computer science*) blisko povezani s dijelovima matematičke teorije, oportuno se kod modeliranja veličina iz problema ograničiti na varijable koje predstavljaju dobro definirane matematičke strukture.

Najprirodniji i intuitivno najjasniji primjeri varijabli problema su realne i cjelobrojne varijable, zajedno sa za primjenu vrlo značajnim podskupom binarnih varijabli, kojima se mogu modelirati različite skalarne veličine u optimizacijskom problemu i koje se poglavito koriste u različitim varijantama standardne matematičke optimizacije realnih funkcija. Vrijednosti realnih i cjelobrojnih varijabli su elementi skupa prirodnih i realnih brojeva, odnosno njihovih podskupova definiranih zatvorenim intervalom dopuštenih vrijednosti varijable, a prilikom programske realizacije putem računalnog algoritma se bezbolno preslikavaju u pripadni cjelobrojni (*int*) odnosno realni (*float* ili *double*) tip podatka.

Međutim, u matematičkom modeliranju optimizacijskih problema prirodno se nameću i mnogobrojne vrste varijabli sa složenijom strukturom. Primjerice, kod klasičnog problema trgovačkog putnika u kojem je za dane udaljenosti između gradova potrebno naći rutu kojom će se svi gradovi obići jedanput uz minimalnu pređenu ukupnu udaljenost, najprirodniju reprezentaciju za varijablu problema čini skup permutacija $S_n$ čime se na prirodni način modelira očita činjenica da rješenje problema mora definirati redoslijed obilaska gradova. Ukoliko bi problem trgovačkog putnika željeli definirati nad skupom od $n$ cjelobrojnih varijabli, gdje je $n$ broj gradova i gdje vrijednost $i$-te varijable određuje redni broj grada koji će se posjetiti u $i$-tom koraku obilaska, bilo bi nužno uvesti dodatna ograničenja koja bi osigurala da se svaki grad obiđe točno jedanput, a što bi taj skup cjelobrojnih varijabli svelo upravo na element iz skupa permutacija.

Pored skupa permutacija kao podloge za definiranje varijabli problema, a koji svoju primjenu nalazi i kod mnogih drugih, poglavito kombinatornih problema, često se kao varijable problema javljaju i druge matematički dobro definirane strukture poput grafova i matrica. Kod različitih problema pridruživanja (npr. linearno, kvadratično) i problema raspoređivanja (npr. problem rasporeda strojeva ili izrade satnice – engl.

*timetabling*), kao prirodna varijabla problema pojavljuje se kombinatorna struktura *pridruživanja.* U slučaju bijektivnog pridruživanja *n* jedinki na *n* raspoloživih mjesta pridruživanje se može reprezentirati permutacijom, ali u općenitom slučaju predstavlja složeniju matematičku strukturu. S druge strane, veliki dio područja genetičkog programiranja (engl. *genetic programming*) se bazira na operiranju nad stablima koja predstavljaju LISP izraze.

Značaj vrsta varijabli nad kojima je definiran optimizacijski problem do posebnog izražaja dolazi u kontekstu optimizacijskih postupaka. Naime, da bi se optimizacijski postupak mogao primijeniti na rješavanje optimizacijskog problema, nužno je da taj postupak u svojoj implementaciji preko računalnog algoritma ima podatak o tome nad kakvim varijablama operira. S obzirom da se većina optimizacijskih postupaka razmatranih u ovoj disertaciji može primijeniti na rješavanje različitih optimizacijskih problema, jedan od osnovnih uvjeta izgradnje općenitog optimizacijskog okruženja je razdvajanje (engl. *decoupling*) optimizacijskog algoritma od konkretnog skupa varijabli problema nad kojima on operira. Detaljan opis načina na koji je to razdvajanje postignuto kod izgradnje ESOP okruženja je dan u četvrtom i petom poglavlju (poglavito u odjeljku 4.2.2.).

### 1.2.3 Optimizacijski problemi s ograničenjima

Velika većina problema koji se javljaju u praksi imaju definirana ograničenja koje optimalno rješenje mora zadovoljavati. Njihova prisutnost značajno komplicira izradu optimizacijskog postupka budući da prostor mogućih rješenja više nije predstavljen kompletnim Kartezijevim produktom skupova nad kojima su definirane varijable.

Kod nekih problema se ograničenja mogu ukloniti prikladnom reprezentacijom. Ukoliko kao primjer uzmemo već spomenuti problem trgovačkog putnika, jedino ograničenje je da se svaki grad posjeti točno jedanput. Međutim, ukoliko se za reprezentaciju rješenja (odnosno varijabla nad kojom je definiran problem) uzme skup permutacija $S_n$, koji predstavlja skup svih permutacija niza brojeva od 1 do *n*, gdje je interpretacija takve reprezentacije da ona definira redoslijed kojim se posjećuju gradovi, ograničenje će automatski biti zadovoljeno za sva moguća rješenja i mogu se primijeniti optimizacijski postupci optimizacije bez ograničenja. S druge strane, ukoliko se za reprezentaciju rješenja uzme skup matrica *n* x *n*, sa elementima iz skupa {0, 1} gdje vrijednost elementa $x_{i,j} = 1$ povlači da se iz *i*-tog grada ide u *j*-ti, očigledno je da će se morati definirati ograničenja kako bi se osiguralo da se svaki grad posjeti samo jednom (odnosno, suma po svakom retku i po svakom stupcu mora biti jednaka 1 – iz svakog grada se po jednom odlazi i u svaki grad se jednom dolazi). Slično vrijedi i za ostale probleme kombinatorijalne optimizacije (JSSP, VRP, QAP) gdje se u većini slučajeva eksplicitna ograničenja mogu ukloniti upotrebom prikladne reprezentacije.

U slučajevima gdje takav postupak nije moguće provesti, ograničenja se moraju ugraditi u samu definiciju optimizacijskog problema. Slično kao i kod kriterija optimalnosti koje modeliramo preko funkcija cilja, ograničenja problema su također definirana kao matematičke funkcije nad varijablama problema s tom razlikom što tako

definirane funkcije predstavljaju lijevu stranu nejednakosti ili jednakosti koju konačno rješenje mora zadovoljavati. Iako postoje tri različite vrste ograničenja ($g(x) \leq$, 0 $g(x) \geq$ 0 i $g(x) = 0$), sve tri vrste se elementarnim matematičkim operacijama mogu svesti na jedan tip pa se tako u literaturi kod definiranja optimizacijskog problema najčešće sva ograničenja svode na oblik $g(x) \leq 0$. S obzirom na ograničenja u prikazu točnosti realnih brojeva u računalu, jednakost $g(x) = 0$ se najčešće ugrađuje kao nejednakost $|g(x)| \leq \Delta$, gdje je $\Delta$ mali realni broj određen točnošću prikaza realnih brojeva u računalu ili definiranim relaksacijskim kriterijima problema. Iz istog razloga se vrlo često i kod ograničenja tipa $\leq$ i $\geq$ primjenjuje relaksacijski kriterij te ona prelaze u oblik $g(x) \leq |\Delta|$ i $g(x) \geq -|\Delta|$ (dakle, dopuštamo minimalna kršenja ograničenja).

### 1.2.4 Problemi s parametrima

Ukoliko proanaliziramo optimizacijske probleme koji se javljaju u praksi, uočiti ćemo da kod velikog broja problema postoje parametri. Neki od tih parametara fundamentalno utječu na samu definiciju problema (primjerice broj gradova kod problema trgovačkog putnika ili broj strojeva i poslova kod JSSP problema) dok neki proizlaze iz ograničenja koja se postavljaju kod modeliranja problema (npr. broj i kapacitet vozila raspoloživih za prijevoz robe sa skladišta do trgovina – varijacija VRP problema).

Kod problema koji spadaju u prvu skupinu u neku ruku i ne možemo govoriti o parametarskim problemima budući da promjena vrijednosti parametra vodi na sasvim novi optimizacijski problem. Stoga je analiziranje i uspoređivanje rezultata primjene optimizacijskih postupaka na problem za različite vrijednosti parametra smisleno samo u kontekstu promatranja performansi optimizacijskog postupka (npr. usporedba performansi genetičkog algoritma na TSP problemu sa 10, 100 i 1000 gradova).

S druge strane, problemi koji spadaju u drugu skupinu su sa stajališta izgradnje optimizacijskog okruženja zanimljiviji jer se kao prirodan zahtjev nameće ugradnja modula za provođenje analize osjetljivosti. Programska podrška za jednostavno definiranje i rješavanje zadanog problema s različitim vrijednostima parametara i međusobno uspoređivanje nađenih rješenja može biti od velike koristi kod donošenja odluka (kako u poslovnoj tako i u drugim domenama primjene).

Kao tipičan (jednostavan) primjer se može navesti problem razvoženja robe od skladišta do trgovina, gdje imamo dva parametra: broj vozila i kapacitet svakog vozila. Za rukovoditelja koji mora odlučiti kakve kamione kupiti i koliko njih, informacija o optimalnim rutama razvoženja i pripadnim troškovima za različite izbore je vrlo zanimljiva (a moglo bi se reći i nužna).

Kao drugi i značajno složeniji primjer se može uzeti problem izrade rasporeda (satnice) za visokoškolsku ustanovu. Uzevši u obzir relativno mali broj sati direktne nastave koje svaki nastavnik mora odraditi (u usporedbi s osnovnim i srednjim školama) i velik broj izbornih predmeta kod upisa kojih se studentima daje značajna razina fleksibilnosti (što povlači i veliki broj preklapanja i nužnost formiranja malih nastavnih grupa), definiranje kriterija kod određivanja optimalnog rasporeda je vrlo subjektivan posao. Stoga mogućnost usporedbe nađenih rasporeda za različito postavljene situacije

(npr. „sva predavanja se moraju održati u terminima do 16 sati„ ili „predavanja iz predmeta XYZ se uvijek moraju održavati srijedom“ ili „predavanja uvijek moraju biti bar N dana prije vježbi“) izgleda vrlo primamljivo, osobito u slučaju kad satničar mora opravdavati odluke o donesenom rasporedu.

Ugradnja mogućnosti provođenja analize osjetljivosti postavlja određene zahtjeve na dizajn i arhitekturu optimizacijskog okruženja o čemu će više riječi biti u kasnijim poglavljima.

## 1.3 Standardni optimizacijski problemi

Kriteriji klasifikacije optimizacijskih problema navedeni u prethodnih nekoliko poglavlja će predstavljati osnovu za modeliranje objektnog modela na kojem će se bazirati ESOP optimizacijsko okruženje. Važnost tih kriterija klasifikacije proizlazi iz činjenice da oni u određenom smislu definiraju zahtjeve (engl. *requirements*) na optimizacijsko okruženje a mogućnost modeliranja, „ugrađivanja“ i rješavanja različitih vrsta problema predstavlja direktnu mjeru širine primjenjivosti danog optimizacijskog okruženja, a samim tim i mjeru njegove iskoristivosti (engl. *usability*).

S obzirom na raznolikost optimizacijskih problema s kojima se istraživači susreću u praksi, jasno je da optimizacijsko okruženje nikada ne može biti kompletno u smislu da dolazi s „*out-of-the-box*“ mogućnošću primjene na bilo koji problem. Stoga je najbolje što se može očekivati da optimizacijsko okruženje pruža mogućnost *proširivanja* s novodefiniranim vrstama optimizacijskih problema na način da istraživači samostalno izgrade određene programske komponente, koje se zatim mogu iskoristiti unutar optimizacijskog okruženja.

Kako bi se bolje pojasnilo značenje pojedinih kriterija klasifikacije i njihova veza s konkretnim optimizacijskim problemima, u nastavku poglavlja je dan detaljan opis odabranog skupa standardnih optimizacijskih problema. Svrha opisa odabranih problema je dvojaka. S jedne strane, cilj je na osnovu detaljnog opisa karakteristika konkretnih problema vidjeti kako se oni uklapaju u navedenu klasifikaciju te steći dodatni uvid u zahtjeve na objektni model koji proizlaze iz mogućnosti ugradnje tih problema u optimizacijsko okruženje. S druge strane, taj opis će predstavljati osnovu za izgradnju optimizacijskih komponenti koje će činiti standardni dio optimizacijskog okruženja.

### 1.3.1 Optimizacija realnih funkcija

Optimizacija realnih funkcija, bilo skalarnih bilo vektorskih, je dobro poznati problem u matematici kod kojega je cilj minimizirati ili maksimizirati zadanu realnu funkciju sistematičnim odabirom vrijednosti realnih ili cjelobrojnih varijabli nad kojima je funkcija definirana. Preciznije se problem može definirati na sljedeći način:

**Definicija**: Neka je zadana funkcija $f : A \rightarrow R$, gdje je A neki skup realnih brojeva. Tada je potrebno pronaći takav element $x_0 \in A$ da vrijedi $f(x_0) \leq f(x) \ \forall x \in A$ (u slučaju minimizacije), odnosno naći takav $x_0 \in A$ da vrijedi $f(x_0) \geq f(x) \ \forall x \in A$ ukoliko se radi o problemu maksimizacije [Wikipedia2006a].

Takav problem se u literaturi naziva problemom matematičkog programiranja i mnogobrojni problemi, kako teorijske tako i praktične prirode, se mogu svesti na ovakvu formulaciju. Skup A je u tipičnom slučaju podskup Euklidskog prostora $R^n$ koji je najčešće određen ograničenjima koje elementi skupa moraju zadovoljavati (odnosno, zadani su intervali dopuštenih vrijednosti za varijable nad kojima je problem definiran). Unutar područja matematičkog programiranja postoji nekoliko potpodručja koja se razlikuju po karakteristikama funkcija cilja i varijabli nad kojima je problem definiran:

- linearno programiranje
- kvadratično programiranje
- cjelobrojno programiranje
- nelinearno programiranje

Pored navedenih, moglo bi se reći standardnih matematičkih problema, na ovakvu formulaciju se mogu svesti i problemi kombinatorne optimizacije kod kojih je skup mogućih rješenja (odnosno skup A) diskretan i najčešće definiran nad složenijim matematičkim strukturama (npr. korištenjem matematičkih struktura permutacije ili pridruživanja). Međutim, kako su tehnike rješavanja takvih problema donekle različite u odnosu na tehnike koje se primjenjuju za rješavanje navedenih standardnih problema, u daljnjem tekstu će se pod problemom optimizacije realnih funkcija podrazumijevati probleme u kojima su varijable problema definirane nad skupom $R^n$.

U slučaju da je funkcija cilja dvostruko diferencijabilna, za nalaženje rješenja se mogu koristiti gradijentne metode koje se oslanjaju na činjenicu da je u ekstremalnim točkama funkcije gradijent jednak nuli. U ovu klasu metoda rješavanja spadaju Newton-ova metoda, tehnika konjugiranog gradijenta (engl. *conjugate gradient*) i tehnika linijskog pretraživanja (engl. *line search*).

Kod primjene navedenih metoda se javlja i problem nalaženja globalnog optimuma. Naime, ukoliko funkcija cilja nije monotona, postoji mogućnost postojanja više lokalnih optimuma dane funkcije. U tom slučaju je potrebno modificirati primijenjene tehnike kako bi se osiguralo da je nađeno rješenje doista globalni optimum. U slučaju funkcija koje imaju veliki broj lokalnih minimuma, značajno bolji rezultati se postižu primjenom heurističkih metoda optimizacije poput genetičkih algoritama, evolucijskih strategija ili simuliranog kaljenja. ESOP okruženje je primarno namijenjeno upravo izgradnji heurističkih metoda optimizacije, i detaljniji opis navedenih metoda će biti dan u sljedećem poglavlju.

U skladu s klasifikacijom provedenom na početku ovog poglavlja, možemo definirati nekoliko vrsta problema optimizacije realnih funkcija. Ukoliko ograničenja kojima je određen skup A nisu zadana kao ograničenja na raspon varijabli problema već su definirana preko nejednadžbi definiranih nad vrijednostima tih varijabli, dolazimo do problema optimizacije realne funkcije s ograničenjima kojeg možemo definirati na sljedeći način:

**Definicija**: Neka je zadana funkcija cilja $f(x) : A \rightarrow R$, gdje je A neki skup realnih brojeva i $x \in A$ i neka je zadan skup nejednadžbi ograničenja $g_i(x) : A \rightarrow R$, oblika $g_i(x) \leq 0$. Tada je potrebno pronaći takav element $x_0 \in A$ koji zadovoljava zadana ograničenja

i da vrijedi $f(x_0) \leq f(x)\ \forall x \in A$ (u slučaju minimizacije), odnosno da vrijedi $f(x_0) \geq f(x)$ $\forall x \in A$ ukoliko se radi o problemu maksimizacije.

Dodatnu podvrstu problema optimizacije realnih funkcija predstavljaju problemi višekriterijske optimizacije, gdje je funkcija cilja definirana kao vektorska realna funkcija što znači da postoji više (skalarnih) realnih funkcija koje definiraju kriterije optimizacije. Kako je već ranije opisano ovom poglavlju, u tom slučaju je cilj optimizacije pronaći Pareto plohu nedominiranih rješenja, i dostizanje tog cilja zahtijeva značajnu modifikaciju navedenih optimizacijskih postupaka. S obzirom da i višekriterijski problemi mogu imati definiran skup ograničenja, gdje je način njihovog modeliranja isti kao i u već opisanom slučaju jednokriterijske optimizacije, vidimo da probleme optimizacije realnih funkcija možemo podijeliti u četiri različita razreda koji će prilikom izgradnje ESOP optimizacijskog okruženja biti direktno programski modelirani i realizirani, a što će biti opisano u kasnijim poglavljima.

## 1.3.2 Problem trgovačkog putnika

Problem trgovačkog putnika (engl. *traveling salesman problem* – skraćeno TSP) je jedan od najpoznatijih problema kombinatorne optimizacije. Problem se može definirati na sljedeći način:

**Definicija**: Neka je zadan skup gradova $\{c_1, c_2, ..., c_N\}$ i neka je za svaki par gradova $\{c_i, c_j\}$ zadana udaljenost $d(c_i, c_j)$. Tada je cilj pronaći raspored obilaska gradova $\pi$ koji minimizira vrijednost funkcije cilja zadane kao:

$$\sum_{i=1}^{N-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(N)}, c_{\pi(1)}) \tag{1.3.1}$$

Vrijednost funkcije cilja odgovara udaljenosti koju bi trgovački putnik prošao prilikom obilaska svih gradova, po čemu je problem i dobio ime. Ukoliko su udaljenosti $d(c_i, c_j)$ i $d(c_j, c_i)$ jednake, govorimo o *simetričnom* problemu trgovačkog putnika, a inače se radi o *asimetričnom* TSP problemu. Detaljan pregled problema trgovačkog putnika i njegovih različitih varijacija se može naći u [Gutin2002].

Usprkos svojoj (prividnoj) jednostavnosti, TSP spada u skupinu NP-teških problema [Garey1979] te stoga za svaki algoritam za pronalaženje optimalnog rješenja postoji najgori slučaj kod kojega vrijeme pronalaženja optimalnog rješenja raste brže od bilo kojeg polinoma (ukoliko je ispravna pretpostavka $P \neq NP$). Problem trgovačkog putnika ima mnogobrojne primjene u praksi, od dizajna VLSI čipova do primjene u kristalografiji pomoću X-zraka, a detaljan pregled se može naći u [Lawler1985].

Zbog jednostavnosti definiranja problema i činjenice da se radi o NP-teškom problemu, problem trgovačkog putnika često služi kao „poligon za testiranje“ različitih optimizacijskih postupaka. Različiti istraživači su primjenjivali različite optimizacijske postupke za rješavanje. Od varijanti lokalnog pretraživanja (gdje se korištenje Lin-Kernigan-ovog operatora lokalnog pretraživanja pokazuje kao jedna od najefikasnijih metoda [Lin73]), do primjena simuliranog kaljenja i genetičkih algoritama.

## 1.3.3 Problemi pridruživanja

Problemi pridruživanja se bave pitanjem kako pridružiti *n* jedinki (poslovi, tvornice, radnici) na *n* drugih jedinki (strojevi, lokacije, zadaci). Kombinatorna struktura na kojoj se ti problemi zasnivaju je *pridruživanje*, koje je u stvari bijektivno preslikavanje $\phi$ između dva konačna skupa elemenata. Kod problema optimizacije tražimo najbolje pridruživanje; znači, ono koje optimizira neku zadanu funkciju cilja.

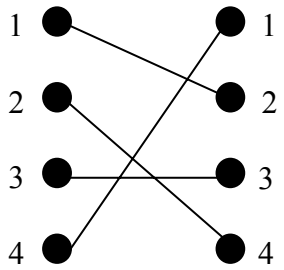
Problem pridruživanja (ili slaganja) entiteta iz jednog skupa na entitete drugog skupa se pojavljuje u mnogim praktičnim problemima dizajna ili alokacije resursa. Ukoliko su pridruženi troškovi konstante za svako moguće sparivanje, tada se radi o klasičnom problemu pridruživanja, za koji su algoritmi rješavanja poznati već dugo vremena. S druge strane, ukoliko je struktura troškova složenija, tako da trošak određenog pridruživanja ovisi i o drugim sparivanjima, dolazimo do teških kombinatornih problema, od kojih je kvadratični problem pridruživanja jedan od primjera.

### 1.3.3.1 Matematička definicija pridruživanja

Neka su $V$ i $W$ dva konačna skupa elemenata, $V = \{ v_1, v_2, ..., v_n \}$ i $W = \{w_1, w_2, ..., w_n\}$. Tada pridruživanje $\phi : V \rightarrow W$ koje svakom elementu iz skupa $V$ pridružuje element iz skupa $W$ možemo prikazati permutacijom $\phi$ skupa $N = \{1, 2, ..., n\}$ gdje je elementu $v_i$ iz skupa $V$ pridružen element $w_{\phi(i)}$ iz skupa $W$. Drugi način na koji možemo prikazati pridruživanje je pomoću permutacijske matrice $X_\phi = (x_{ij})$ gdje je $x_{ij} = 1$ za $j = \phi(i)$ i $x_{ij} = 0$ za $j \neq \phi$ (i). Oba načina su prikazana, za isto pridruživanje, na slici 1, na kojoj je pridruživanje prikazano i kao bipartitni graf.

$$\phi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$$

$$X_\phi = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$




**Slika 1.1 Različiti prikazi pridruživanja**

Lako je uočiti da postoji jedan-naprema-jedan ekvivalencija između skupa svih permutacija $S_n$ i skupa permutacijskih matrica, koja je definirana na sljedeći način: neka je $X=(x_{ij})$ matrica dimenzija $n \times n$. Ukoliko elementi matrice $x_{ij}$ zadovoljavaju sljedeće uvjete :

$$\sum_{i=1}^{n} x_{ij} = 1, \quad 1 \le j \le n$$
$$\sum_{j=1}^{n} x_{ij} = 1, \quad 1 \le i \le n \tag{1.3.2}$$
$$x_{ij} \in \{0,1\}, \quad 1 \le i \le n, \; 1 \le j \le n$$

tada $X$ nazivamo *permutacijskom matricom*. Skup svih permutacijskih matrica dimenzija $n \times n$ označavamo sa $\Pi_n$. Jedan-naprema-jedan korespondencija se realizira tako da se svakoj permutacijskoj matrici $X=(x_{ij}) \in \Pi_n$ pridruži permutacija $\pi_X \in S_n$, gdje je $\pi_X(i) = j$ ako i samo ako vrijedi $x_{ij} = 1$.

### 1.3.3.2 Linearni problem pridruživanja

Linearni problem pridruživanja predstavlja varijantu problema pridruživanja kod koje je funkcija cilja linearna i kod kojeg je potrebno rasporediti $n$ strojeva na $n$ poslova, tako da ukupni trošak raspoređivanja bude minimalan, s time da je poznat trošak raspoređivanja svakog stroja na svaki pojedinačni posao. Tada problem možemo matematički formulirati kao:

$$\min \quad \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

uz ograničgraa

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad 1 \le j \le n \tag{1.3.3}$$
$$\sum_{j=1}^{n} x_{ij} = 1 \qquad 1 \le i \le n$$
$$x_{ij} \in \{0,1\} \qquad 1 \le i \le n, 1 \le j \le n$$

gdje je C = ($c_{ij}$) matrica troška raspoređivanja $i$-tog elementa $j$-tu lokaciju. Uz korištenje simboličke notacije gdje $\phi$ označava permutaciju a $c_{ij}$ su koeficijenti troška, problem se može definirati i kao:

$$\min_{\phi} \left( c_{1\phi(1)} + c_{2\phi(2)} + \ldots + c_{n\phi(n)} \right) \tag{1.3.4}$$

Za rješavanje ovakvog problema pridruživanja je razvijen veliki broj algoritama, npr. primalno-dualni algoritmi, simpleks metode ili algoritmi sa relaksacijskim pristupom. Pregled metoda i raspoloživi računalni programi se mogu naći u [Burkard1999]. Ovako definiran problem pridruživanja je moguće riješiti u polinomnom vremenu $O(n^3)$, ali su razvijeni algoritmi s očekivanim trajanjem $O(n^2 log\ n)$ u slučaju nezavisnih i uniformno distribuiranih koeficijenata troška $c_{ij}$ unutar intervala [0,1].

Linearni problem pridruživanja se u praksi najčešće pojavljuje kao potproblem u složenijim problemima, primjerice u problemu trgovačkog putnika, problemu pridruživanja osoblja i slično.

### 1.3.3.3 Kvadratični problem pridruživanja

Kvadratični problem pridruživanja (u literaturi poznat po skraćenici QAP koja dolazi od engleskog termina *Quadratic Assignment Problem*) formulirali su Koopmans i Beckman 1957. godine prilikom rješavanja problema određivanja lokacija za skup tvornica uz minimizaciju ukupnih troškova toka materijala među tvornicama [Koopmans1957]. Za razliku od linearnog problema pridruživanja, koji je rješiv u polinomnom vremenu, kod QAP-a je funkcija cilja kvadratična u svojim varijablama (odakle mu potječe i ime), te je nalaženje optimalnih rješenja puno teže.

Jedan od glavnih razloga velikog interesa za kvadratični problem pridruživanja je činjenica da se mnogobrojni konkretni problemi iz područja diskretne optimizacije mogu svesti upravo na QAP. Na popisu tih problema se nalaze problemi iz područja raspoređivanja, proizvodnje, dizajna VLSI krugova, statističke analize i drugih. Uz to se i neki poznati problemi iz kombinatorijalne optimizacije mogu formulirati kao kvadratični problem pridruživanja: npr. problem trgovačkog putnika i problem particioniranja grafa.

Opis problema, koristeći originalni primjer određivanja rasporeda tvornica je sljedeći. Kao ulazni podaci su dane tri matrice: matrica toka između tvornica $A = (a_{ij})$, gdje je $a_{ij}$ tok materijala između tvornica $i$ i $j$, matrica udaljenosti $B = (b_{kl})$, gdje je $b_{kl}$ udaljenost između lokacija $k$ i $l$, te matrica $C = (c_{ik})$ koja predstavlja trošak postavljanja $i$-te tvornice na $k$-tu lokaciju. Potrebno je naći takav raspored tvornica po lokacijama kako bi ukupan trošak bio minimalan.

Pridruživanje svih tvornica na određene lokacije se može prikazati permutacijom $\pi \in S_n$, gdje je $S_n$ skup svih mogućih permutacija od $N$ elemenata. Trošak istovremenog postavljanja tvornice $\pi(i)$ na lokaciju $i$ i tvornice $\pi(j)$ na lokaciju $j$ je dan sa $a_{\pi(i)\pi(j)}\, b_{i\,j}$ (kojem, jasno, treba pribrojiti trošak postavljanja pojedinačne tvornice na svaku lokaciju $c_{\pi(i)i} + c_{\pi(j)j}$ ). Ukupan trošak pridruživanja $\pi$ svih tvornica je stoga jednak

$$\min_{\pi \in S_n} \left( \sum_{i=1}^{n} \sum_{j=1}^{n} a_{\pi(i)\pi(j)}\, b_{ij} + \sum_{i=1}^{n} c_{\pi(i)i} \right), \tag{1.3.5}$$

a cilj optimizacije je minimizirati ukupni trošak postavljanja i rada tvornica. Nešto općenitija definicija kvadratičnog problema pridruživanja koju je postavio Lawler [Lawler1963] je dana s

$$\min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{\pi(i)i\,\pi(j)j} \tag{1.3.7}$$

gdje je $D = d_{kilj}$ 4-dimenzionalno polje realnih brojeva, $i, j, k, l = 1,2, ..., n$. Ukoliko za elemente polja vrijedi $d_{kilj} = a_{kl}b_{ij}$, za $1 \le i, j, k, l \le n$, očito je tada Lawlerov problem ekvivalentan formulaciji QAP-a danoj s (1.3.5). Detaljan opis svih aspekata kvadratičnog problema pridruživanja se može naći u [Cela1998].

U primjenama se javlja i tzv. bikvadratični problem pridruživanja (engl. skraćenica BiQAP) koji su uveli Burkard, Çela i Klinz, a čije uvođenje je bilo

motiviranom rješavanje optimizacije povezane sa dizajnom sinkronih sekvencijalnih VLSI krugova. BiQAP se matematički može formulirati na sljedeći način:

$$\min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} a_{\pi(i)\pi(j)\pi(k)\pi(l)} b_{ijkl} \tag{1.3.11}$$

gdje $S_n$ predstavlja skup svih permutacija od {1, 2, ... $n$}, a $A$ i $B$ su četverodimenzionalna polja. Više o BiQAP-u, sa detaljnim referencama se može naći u [Burkard1994].

# 2. Načini rješavanja

U prethodnom poglavlju naglasak je bio na opisivanju različitih optimizacijskih problema, njihovih karakteristika i primjena u praksi. Jasno je, međutim da je definiranje optimizacijskog problema samo prvi korak u procesu optimiranja. Sa stajališta cjelokupnog procesa optimiranja, puno je važniji (i teži!) korak rješavanje definiranog optimizacijskog problema primjenom nekog optimizacijskog postupka realiziranog računalnim algoritmom.

Međutim, slično kao i kod optimizacijskih problema, raznolikost principa rada različitih optimizacijskih postupaka zahtijeva izgradnju vrlo fleksibilnog optimizacijskog okruženja. Dodatni otežavajući čimbenik je i činjenica da bi općenito optimizacijskog okruženje trebalo omogućavati i ugradnju *novih*, znači trenutno nepoznatih i neistraženih algoritama i njihovu primjenu na optimizacijske probleme. Radi ilustracije varijabilnosti optimizacijskih postupaka evo kratkog popisa reprezentativnih i široko korištenih optimizacijskih metoda i načina njihovog rada:

- algoritmi lokalnog pretraživanja (engl. *local search algorithms*)

  Odabire se slučajno početna točka u prostoru rješenja i zatim se kontinuiranim pretraživanjem rješenja iz lokalne okoline nastoji naći bolje rješenje.

- dinamičko programiranje (engl. *dynamical programming*)

  Optimizacijski problem se rješava dijeljenjem na potprobleme čija se optimalna rješenja nakon prvog izračunavanja pohranjuju radi kasnijeg (ponovnog) iskorištavanja.

- genetički algoritmi (engl. *genetic algorithms*)

  Uvode pojam dobrote (engl. *fitness*) pojedinog rješenja te s primjenom operatora selekcije, križanja i mutacije koji djeluju nad populacijom jedinki (rješenja) nastoje poboljšati nađeno rješenje.

- simulirano kaljenje (engl. *simulated annealing*)

  Varijanta lokalnog pretraživanja u kojem se s određenom vjerojatnošću dopuštaju i pomaci u prostoru rješenja koji *pogoršavaju* funkciju cilja (radi izlaska iz lokalnih optimuma), s time da se ta vjerojatnost polako smanjuje po nekom rasporedu (engl. *annealing schedule*).

- simpleks metoda (engl. *simplex algorithm*)

  Iterativna metoda za rješavanje linearnih programa koja također predstavlja varijantu lokalnog pretraživanja i kod koje se u svakoj iteraciji odabire kao sljedeće ono rješenje koje najviše poboljšava funkciju cilja.

Imajući u vidu raznolikost postojećih optimizacijskih postupaka, prije pristupanja izgradnji samog optimizacijskog okruženja je potrebno proučiti različite optimizacijske postupke s ciljem pronalaženja skupa njihovih zajedničkih i individualnih karakteristika. Identificirane karakteristike će prilikom izgradnje okruženja predstavljati osnovu za

definiranje konceptualnog (analitičkog) i objektnog modela na kojem će se graditi optimizacijsko okruženje.

Ostatak poglavlja je podijeljen u dva dijela. U prvom dijelu su opisani različiti načini klasifikacije optimizacijskih postupaka, dok je u drugom dijelu dan opis pojedinih optimizacijskih metoda.

# 2.1 Klasifikacija optimizacijskih postupaka

Optimizacijski postupci se mogu klasificirati na različite načine, ovisno o primijenjenom kriteriju klasifikacije. Nekoliko načina klasifikacije je predloženo u [Wikipedia2006b] od kojih su sa stajališta izgradnje optimizacijskog okruženja od posebnog značenja klasifikacija po načinu implementacije i klasifikacija po paradigmi/metodologija oblikovanja (engl. *design paradigm / methodology*).

## 2.1.1 Klasifikacija po načinu implementacije

Kod klasifikacije po načinu implementacije možemo definirati sljedeći skup kriterija:

- rekurzivni / iterativni

  Rekurzivni algoritam koriste uzastopne rekurzivne pozive, najčešće s ciljem reduciranja složenosti instance problema koji se rješava („podijeli pa vladaj" paradigma), dok iterativni algoritmi rade na principu obavljanja određenog broja iteracija gdje se u svakoj iteraciji obavlja izračunavanje specifično za pojedini algoritam.

  Iako se svaki rekurzivni algoritam može pretvoriti u iterativni (i obrnuto), pojedini problemi se prirodnije rješavaju korištenjem određene vrste algoritma.

- slijedni / paralelni

  Ovaj kriterij klasifikacije je blisko povezan s arhitekturom programske implementacije algoritma. Slijedni algoritmi obavljaju jedan po jedan korak algoritma operirajući nad cijelim problemom, dok paralelni algoritmi dijele problem na više jednostavnijih potproblema koji se korištenjem višeprocesorskog računala rješavaju u paraleli.

  Neki algoritmi su inherentno serijski i ne mogu se paralelizirati. Za većinu ostalih problema osnovni kriterij efikasnosti paralelizacije čini trošak komunikacije (engl. *communication overhead*) razmjenjivanja podataka između različitih procesora.

- deterministički / stohastički

  Deterministički algoritmi rješavaju problem donošenjem egzaktne odluke u svakom koraku algoritma. Slučajni algoritmi istražuju prostor rješenja s većom ili manjom dozom stohastičnosti sve dok ne pronađu prihvatljivo rješenje. Različiti heuristički algoritmi spadaju upravo u ovu kategoriju.

- egzaktni / aproksimativni

  Neki algoritmi su izgrađeni tako da traže točno rješenje problema (odnosno globalni optimum u slučaju optimizacijskih problema), dok neki traže što bolju aproksimaciju točnog rješenja. Aproksimativni algoritmi su nužni za rješavanje teških problema za koje nije moguće naći točno rješenje.

### 2.1.2 Klasifikacija po paradigmi oblikovanja

Kod klasifikacije algoritama po paradigmi oblikovanja možemo definirati sljedeći skup paradigmi koje se najčešće javljaju u praksi, a koje pojedinačno obuhvaćaju različite tipove algoritama:

- „podijeli pa vladaj“ (engl. *divide and conquer*)

  Algoritmi koji spadaju u ovu kategoriju rade na način da uzastopno reduciraju instancu problema na sve manje i manje instance istog problema, najčešće primjenjujući tehniku rekurzije, sve dok se ne dođe do lako rješivih instanci problema.

- dinamičko programiranje

  Ova paradigma je primjenjiva ukoliko problem koji se rješava ima optimalnu podstrukturu (optimalno rješenje problema može se konstruirati od optimalnih rješenja potproblema) i prekrivajuće potprobleme (isti potproblemi koriste se kod rješavanja više različitih instanci problema). Tada se primjenom tehnike dinamičkog programiranja, čija je osnovna karakteristika izbjegavanje izračunavanja rješenja koja su već ranije izračunata, može vrlo brzo naći optimalno rješenje problema.

- „pohlepne“ (engl. *greedy*) metode

  Pohlepni algoritmi su slični algoritmima dinamičkog programiranja, s tom razlikom da ne izračunavaju optimalna rješenja svih potproblema, već se nakon pronalaženja rješenja za nekoliko potproblema primjenjuje „pohlepni“ kriterij prihvaćanja rješenja koje je trenutno najbolje.

- linearno programiranje

  Kod rješavanja problema korištenjem linearnog programiranja, program se izražava preko funkcije cilja i skupa linearnih nejednadžbi te se primjenom odgovarajućeg algoritma (simpleksni ili Karmakarov algoritam) nalazi optimalno rješenje. Pomoću linearnog programa se mogu reprezentirati mnogi problemi iz područja optimiranja, pogotovo ukoliko se koriste cjelobrojne i binarne varijable (u kojem slučaju se radi o području mješovito-cjelobrojnih linearnih programa).

- pretraživanje i enumeracija

  U ovisnosti o strukturi, složenosti i kardinalnosti prostora rješenja za dani problem, ponekad su primjenjivi i algoritmi koji se baziraju na pretraživanju i enumeraciji prostora rješenja. Tipičan primjer su problemi koji se mogu modelirati kao problemi nad strukturom grafa (npr. igranje šaha).

- probabilistička i heuristička paradigma

  Algoritmi koji spadaju u ovu skupinu se donekle razlikuju od prethodno opisanih algoritama po tome što tijek i konačni rezultat njihovog proračunavanja nije precizno determiniran (po načinu implementacije uglavnom spadaju u skupinu stohastičkih i aproksimativnih algoritama). Tipični primjeri su:

  - stohastički (slučajni, probabilistički) algoritmi – određeni izbori tijekom proračunavanja se obavljaju slučajno (ili pseudo-slučajno)
  - evolucijski algoritmi – algoritmi koji oponašaju biološke evolucijske procese s ciklusom križanja i mutacija jedinki (koje predstavljaju rješenja) čime se oponašaju reprodukcija i „preživljavanje najboljih". Primjeri su genetički algoritmi, evolucijske strategije i algoritmi genetičkog programiranja.
  - heuristički algoritmi – algoritmi kojima je osnovni cilj nalaženje što boljeg približnog rješenja kada nalaženje optimalnog rješenja nije praktično ili moguće. Osnovna metoda rada algoritama ove vrste je slučajno variranje nađenih rješenja s ciljem njihovog poboljšavanja a primjeri algoritama koji spadaju u ovu skupinu su lokalno pretraživanje, tabu pretraživanje (engl. *taboo search*) i algoritmi simuliranog kaljenja.

## 2.2 Pregled optimizacijskih postupaka

Klasifikacija dana u prethodnom potpoglavlju daje generalnu sliku područja optimizacijskih postupaka i iz nje je razvidna raznolikost postojećih optimizacijskih postupaka koja izgradnju potpuno općenitog optimizacijskog okruženja čini složenim tehničkim izazovom.

Stoga je prilikom izgradnje optimizacijskog okruženja najprije potrebno definirati vrste optimizacijskih postupaka koji će biti podržane od strane tog okruženja. S obzirom da su egzaktne optimizacijske metode (linearno programiranje, algoritmi grananja i ograđivanja, različiti rekurzivni algoritmi) dobro pokrivene komercijalnim alatima i široko dostupnim i besplatnim programskim bibliotekama, naglasak kod izgradnje ESOP optimizacijskog okruženja je stavljen na podršku za heurističke optimizacijske postupke. Ili, u skladu sa navedenom klasifikacijom, na iterativne i aproksimativne algoritme, koji spadaju u heurističku i probabilističku paradigmu.

Ovdje se odmah mora naglasiti da to ne znači da unutar ESOP-a nije moguće ugraditi i neke druge vrste optimizacijskih postupaka, npr. neke vrste egzaktnih algoritama. Primjerice, simpleksni algoritam za rješavanje linearnih programa je također moguće ugraditi u ESOP. Radi se samo o tome da je kod izgradnje stavljen naglasak na podršku za heurističke algoritme iz čega su proizašle određene arhitekturne i implementacijske karakteristike izgrađenog okruženja koje su detaljno opisane u četvrtom i petom poglavlju.

Ipak, mora se reći da postoji i jedna karakteristika koju optimizacijski algoritam mora zadovoljavati da bi mogao biti ugrađen u ESOP. Naime, ESOP je okruženje koje podržava isključivo iterativne algoritme. U tom smislu, optimizacijski postupci koji su

bazirani na rekurzivnoj implementaciji se ne mogu ugraditi u ESOP optimizacijsko okruženje. Razlog tome leži u osnovama postavljene arhitekture ESOP, a koja će biti opisana u petom poglavlju.

Što se tiče ostalih kategorija algoritama navedenih u klasifikaciji, arhitektura ESOP-a je dovoljno široko postavljena da se unutar nje mogu ugraditi i deterministički (egzaktni) algoritmi, sve dok se radi o iterativnoj implementaciji algoritma. U pogledu podjele na slijedne i paralelne algoritme radi se prvenstveno o tehničkom detalju implementacije okruženja, odnosno ugradnji mogućnosti paralelnog izvršavanja optimizacije na više računala / procesora. Iako je napredak modernih razvojnih okruženja (.NET) značajno olakšao izgradnju distribuiranih informacijskih sustava, zbog činjenice da se radi prvenstveno tehničkom problemu koji nije povezan s temom ove disertacije, ESOP okruženje je arhitekturno postavljeno tako da ostavlja mogućnost proširenja u smislu omogućavanja paraleliziranog izvršavanja optimizacije, ali je ugradnja konkretnih programskih komponenata preko kojih bi se realizirala ta mogućnost ostavljena za kasniji razvoj ESOP-a.

Ostatak ovog poglavlja čini detaljan opis pojedinih heurističkih optimizacijskih postupaka s ciljem što boljeg uvida u njihovu strukturu i zahtjeve koje će mogućnost njihove ugradnje postaviti na izgradnju ESOP okruženja.

### 2.2.1 Lokalno pretraživanje

Lokalno pretraživanje predstavlja vrstu heurističkog optimizacijskog postupka koji je primjenjiv na rješavanje optimizacijskih problema kod kojih je cilj naći optimalno rješenje među skupom potencijalnih rješenja. Algoritam radi na principu pretraživanja lokalne okoline trenutnog rješenja te u slučaju pronalaženja boljeg rješenja, to rješenje postaje novo rješenje s kojim se cijeli proces ponavlja ispočetka sve dok se više ne može pronaći bolje rješenje.

Pseudo kôd algoritma za lokalno pretraživanje je vrlo jednostavan:

```
generiraj početno rješenje x
ponavljaj
        generiraj novo rješenje x' iz okoline od x
        ako je x' bolje od x
                uzmi x' kao novo rješenje (x = x')
dok nije zadovoljen kriterij završetka
```

Iz navedenog opisa direktno proizlazi i najveći nedostatak algoritma lokalnog pretraživanja, a to je mogućnost da se postupak zaustavi u lokalnom optimumu. Stoga je kod većine implementacija algoritma lokalnog pretraživanja potrebno ugraditi metodu kojom će se izbjeći zaglavljivanje u lokalnom optimumu.

Različiti algoritmi tom problemu pristupaju na različite načine. Najjednostavnija strategija je reinicijalizacija postupka optimizacije, odnosno slučajno generiranje novog početnog rješenja od kojega se cijeli postupak pretraživanja ponavlja ispočetka. Međutim, primjena te metode se u većini slučajeva ne pokazuje probitačnom jer tako

izgrađen postupak previše ovisi o pogađanju dobre početne točke od koje kreće optimizacije.

Modifikaciju standardnog postupka lokalnog pretraživanja predstavlja varijanta *tabu pretraživanja* (engl. *taboo search*) [Glover1989] kod koje se definira lista zabranjenih poteza određene duljine (tabu lista) što omogućava algoritmu da izađe iz lokalnog optimuma. Klasični postupak lokalnog pretraživanja ne može izaći iz lokalnog optimuma jer nakon izlaska iz točke optimuma sljedeće bolje rješenje predstavlja potez (pomak) koji ponovno vodi u taj isti optimum. Kod tabu pretraživanja će taj pomak biti zabranjen jer se nalazi u tabu listi, čime će algoritam biti prisiljen istražiti ostale dijelove prostora rješenja. Drugačiji princip izbjegavanja zaglavljivanja u lokalnom optimumu se primjenjuje kod algoritma simuliranog kaljenja, koji će biti opisan u sljedećem odjeljku. Pored opisanog klasičnog algoritma lokalnog pretraživanja (koji je u engleskoj literaturi poznat pod nazivom *hill-climbing*), dodatnu varijaciju predstavlja i GRASP algoritam (engl. *greedy randomized adaptive search procedure*) [Festa2001]. Kao najsvjetliji primjer algoritma lokalnog pretraživanja se može spomenuti 2-opt algoritam [Lin1973] koji se primjenjuje kod rješavanja problema trgovačkog putnika i koji daje rezultate koji su usporedivi s implementacijama značajno složenijih algoritama.

Algoritam lokalnog pretraživanja se može primijeniti samo na optimizacijske probleme koji definiraju *strukturu susjedstva* (engl. *neighborhood relation*) preko koje se pronalaze rješenja koja su susjedna trenutnom rješenju i koja predstavljaju potencijalno bolje rješenje u odnosu na trenutno. Stoga se algoritmi lokalnog pretraživanja uglavnom primjenjuju na rješavanje diskretnih optimizacijskih problema kod kojih je strukturu susjedstva moguće jednostavno definirati.

### 2.2.2 Simulirano kaljenje

Postupak simuliranog kaljenja predstavlja varijantu algoritma lokalnog pretraživanja i zasniva se na interesantnoj analogiji između problema kombinatorijalne optimizacije i procesa kaljenja u metalurgiji koju su prvi uočili Kirkpatrick, Gelatti i Vecchi [Kirkpatrick1983], i neovisno od njih V.Cerny [Cerny1985].

U metalurgiji se termin kaljenje koristi za opis procesa kojim se povećava veličina kristala od kojih je izgrađen materijal i smanjuje broj defekata (nepravilnosti u kristalnoj rešetci). U procesu kaljenja se najprije provodi zagrijavanje materijala do točke taljenja čime se atomi oslobađaju iz kristalne rešetke (stanje u kojem atomi imaju minimalnu unutrašnju energiju) te oni prelaze u stanje više energije, nakon čega se provodi spori proces hlađenja čime se omogućava tim istim atomima da se poslože u konfiguraciju s nižom unutrašnjom energijom od početne. Određivanje rasporeda kaljenja (engl. *annealing schedule*) koji definira brzinu hlađenja je krucijalni faktor prilikom provođenja kaljenja. Prebrzo hlađenje neće dati atomima dovoljno vremena da se poslože u bolju konfiguraciju, dok je s druge strane predugačko vrijeme kaljenja neprihvatljivo s proizvodne strane.

Iterativno poboljšavanje, koje je karakteristika algoritama lokalnog pretraživanja, je vrlo slično mikroskopskom preuređenju do kojeg dolazi tijekom procesa kaljenja, s funkcijom cilja u ulozi energije sustava. Po analogiji s fizikalnim

procesom kaljenja, algoritam simuliranog kaljenja u svakom koraku zamjenjuje trenutno rješenje s nekim susjednim rješenjem, uz ključnu razliku da se kao novo rješenje mogu prihvatiti i gora rješenja, ovisno o vrijednosti globalnog parametra $T$ (koji se naziva *temperaturom* sustava) i koji se postupno smanjuje tijekom postupka optimizacije.

Pseudo-kod za implementaciju simuliranog kaljenja je sljedeći:

```
generiraj početno rješenje
definiraj početnu temperaturu
ponavljaj
    ponavljaj
        nađi susjedno rješenje
        izračunaj promjenu vrijednosti funkcije cilja dE
        ako je novo rješenje bolje
                prihvati novo rješenje kao trenutno
        inače
                prihvati novo rješenje s vjerojatnošću P(dE) = e^-dE/kT
    dok se sistem nije smirio na danoj temperaturi
    smanji temperaturu sistema
dok nije zadovoljen uvjet završetka
```

$k$ je konstanta proporcionalnosti (oznaka je istovjetna Boltzmanovoj konstanti koja tu ulogu ima u fizikalnom procesu kaljenja), a $T$ temperatura sistema koja predstavlja kontrolni parametar procedure optimizacije i kojom se kontrolira koliko će se prihvatiti promjena konfiguracije koje vode na lošija rješenja.

Ugrađivanjem mogućnosti prihvaćanja i rješenja koja su lošija od trenutačnog rješenja (s određenom vjerojatnošću), algoritam simuliranog kaljenja izbjegava problem zaglavljivanja u lokalnom optimumu, koji je karakteristika jednostavnijih metoda lokalnog pretraživanja. Algoritam simuliranog kaljenja je primijenjen na rješavanje mnogobrojnih problema u praksi [Koulmas1994], a pogotovo dobri rezultati su postignuti prilikom njegove primjene na problem trgovačkog putnika.

## 2.2.3 Genetički algoritmi

Genetički algoritmi spadaju u širu klasu optimizacijskih postupaka koje nazivamo evolucijskim algoritmima i čiji način rada je inspiriran analogijom s biološkom evolucijom i pripadnim procesima nasljeđivanja, mutacije, prirodne selekcije i nasljeđivanja. Osnovna ideja je potekla od Hollanda koji ih je primijenio u studijama staničnih automata [Holland1975], a danas predstavljaju jednu od najšire korištenih heurističkih optimizacijskih metoda.

### 2.2.3.1 Princip rada

Za razliku od prethodno opisanih optimizacijskih postupaka, genetički algoritmi operiraju nad *populacijom* rješenja, gdje pojedinačne elemente te populacije nazivamo

*jedinkama*. S obzirom da svako rješenje sadrži određenu podatkovnu strukturu čiji sadržaj definira dano rješenje, po analogiji s biološkim sistemima se jedinke često nazivaju i *kromosomima.* Prvi korak u radu genetičkog algoritma je pridruživanje mjere kvalitete svakoj pojedinačnoj jedinke (tu mjeru nazivamo dobrotom rješenja - engl. *fitness*) koje se obavlja pomoću funkcije dobrote (engl. *fitness function*). Kod primjene genetičkih algoritama na probleme optimizacije, dobrota rješenja je u direktnoj vezi s kriterijima optimalnosti definiranih za optimizacijski problem. Nakon obavljanja tog pridruživanja se iz postojeće populacije rješenja generira nova populacija, s time da bolje jedinke imaju veću vjerojatnost ulaska u novu populaciju. Jedinke u novoj populaciji se podvrgavaju utjecaju genetičkih operatora koji iz njih stvaraju nove jedinke. Operatore dijelimo na unarne, koji iz jedne jedinke stvaraju novu jedinku mijenjanjem dijela genetičkog materijala (*mutacijski* operatori) i operatore višeg reda koji kombiniranjem svojstava više jedinki, najčešće dvije, stvaraju nove jedinke (operatori *križanja*). Algoritam se zaustavlja nakon provođenja određenog broja iteracija (odnosno kreiranja određenog broja generacija populacije) i konačno rješenje predstavlja najbolja jedinka u završnoj populaciji.

Pseudo-kod općenitog genetičkog algoritma je sljedeći:

```
t = 0
generiraj početnu populaciju potencijalnih rješenja P(0);
ponavljaj
        t = t + 1;
        selektiraj P'(t) iz P(t-1);
        križaj jedinke iz P'(t) i djecu spremi u P(t);
        mutiraj jedinke iz P(t);
dok nije zadovoljen uvjet završetka evolucijskog procesa
```

#### 2.2.3.1.1 Kodiranje – odabir reprezentacije

Kod implementacije genetičkog algoritma, najprije je potrebno pronaći način kodiranja jedinki, odnosno način reprezentacije rješenja u jedinkama populacije. Genetički algoritmi standardno koriste binarno kodiranje, odnosno, rješenja se reprezentiraju pomoću polja bitova. Korištenje takve reprezentacije direktno proizlazi iz analogije s biološkim sistemima gdje se kromosomi sastoje od određenog broja gena koji se tijekom procesa reprodukcije križaju i doživljavaju mutaciju. U kontekstu optimizacijskih problema pojedinačni „gen“ predstavlja binarnu reprezentaciju jedne ulazne varijable nad kojom je problem definiran, a „kromosom“ tada sačinjava skup svih pojedinačnih gena. Korištenje ovakve reprezentacije omogućava korištenje generičkih operatora križanja i mutacije, odnosno, prilikom rješavanja bilo kojeg optimizacijskog problema je jedino potrebno prevesti rješenje u binarnu reprezentaciju nakon čega se dani problem može riješiti generičkim genetičkim algoritmom. Mogućnost primjene iste implementacije algoritma na različite optimizacijske probleme je jedan od osnovnih razloga popularnosti genetičkih algoritama.

Međutim, istraživanja su pokazala da je sa stajališta efikasnosti optimizacije povoljno koristiti i drugačije reprezentacije rješenja, koje bolje oslikavaju strukturu prostora rješenja optimizacijskog problema. Naziv genetički algoritmi se inicijalno koristio samo za postupke koji su koristili binarnu reprezentaciju, dok se za postupke koji su operirali nad reprezentacijama specifičnim za dani problem koristio termin *evolucijski programi*. Iako se ta distinkcija i dalje može naći u stručnoj literaturi, u praksi se uvriježio termin genetički algoritam i za postupke koji koriste drugačije reprezentacije.

Bitna karakteristika postupaka koji koriste specifične reprezentacije je potreba za izgradnjom posebnih operatora križanja i mutacije. Nezgodna posljedica takvog pristupa je da se time poništava jedna od najvećih prednosti genetičkih algoritama (njihova generičnost i mogućnost „*out-of-the-box*“ primjene na različite optimizacijske probleme) s obzirom da je u tom slučaju potrebno izgraditi nove operatore križanja i mutacije. Međutim, poboljšanja u efikasnosti optimizacije kod korištenja tako prilagođene reprezentacije i pripadnih operatora su uvjetovala da se u praksi najčešće koriste reprezentacije prilagođene problemu koji se rješava.

#### 2.2.3.1.2 Stvaranje početne populacije

Efikasnost rada genetičkog algoritma ovisi o kvaliteti početne populacije jedinki. Valjanost početne populacije ovisi o prosječnoj vrijednosti funkcije cilja jedinki (dobroti) i o raznolikosti populacije (engl. *population diversity*). Što su jedinke u početnoj populaciji bolje, to će biti bolje i krajnje jedinke nakon obavljene optimizacije. Pored toga, velika raznolikost među jedinkama sprječava preranu konvergenciju algoritma u lokalno optimalno rješenje. Najčešće se kreiranje početne populacije obavlja slučajno, generiranjem slučajnih vrijednosti za varijable reprezentacije, ali se ponekad koriste i tehnike usmjerenog kreiranja skupa početnih jedinki, pogotovo u situacijama kad je otprilike poznato (ili se može pretpostaviti) gdje leže optimalna rješenja danog optimizacijskog problema.

#### 2.2.3.1.3 Pridruživanje dobrote

Prvi korak u iteraciji genetičkog algoritma je pridruživanje dobrote svakoj jedinki u populaciji. Dobrota jedinke oslikava kvalitetu dane jedinke i direktno je povezana s vjerojatnošću preživljavanja jedinke u sljedećoj generaciji. Kod primjene genetičkih algoritama na rješavanje optimizacijskih problema, dobrota se proračunava na osnovu kvalitete rješenje. Najčešće se primjenjuju dva principa. Kod prvoga je dobrota pridružena jedinki proporcionalna s mjerom optimalnosti funkcije cilja dok se kod drugoga jedinke (odnosno rješenja problema koje jedinke predstavljaju) rangiraju po optimalnosti pripadnog rješenja te se u skladu s rangom pojedine jedinke pridružuje dobrota.

Kod jednokriterijskih optimizacijskih problema je primjena principa proporcionalnog pridruživanja dobrote vrlo jednostavna. Međutim, takav pristup pati od dva nedostataka. Prvi nedostatak je mogućnost prerane konvergencije ukoliko u populaciji postoje jedinke koje su značajno bolje od ostalih jedinki u populaciji. Zbog toga će takve jedinke imati značajno veću vjerojatnost selekcije za preživljavanje u

sljedeću generaciju te će vrlo brzo kopije takvih jedinki popuniti cijelu populaciju. Drugi nedostatak se pojavljuje u slučaju da su sve jedinke vrlo slične (što povlači i približno istu vrijednost dobrote), jer tada ne postoji *selekcijski pritisak* koji bi algoritam usmjerio ka istraživanju obećavajućih dijelova prostora rješenja.

Pridruživanje dobrote na osnovu ranga rješenja ispravlja navedena dva nedostatka jer se različitosti u kvaliteti jedinke uniformnije skaliraju u raspon dobrota pridruženih jedinkama. Takav pristup je nužan kod primjene na rješavanje višekriterijskih optimizacijskih problema. Budući da u tom slučaju postoji više funkcija cilja, proračunavanje dobrote na osnovu postignutih vrijednosti funkcija cilja nije jednostavno. Stoga se u tom slučaju najčešće primjenjuje tehnika pridruživanja dobrote na osnovu proračunatog ranga rješenja koji je proporcionalan nedominiranosti rješenja. Odnosno, rang rješenja je proporcionalan broju rješenja iz trenutne populacije koja dominiraju nad danim rješenjem. Opis različitih načina pridruživanja dobrote koje su istraživači primjenili u praksi prilikom rješavanja višekriterijskih problema pomoću genetičkih algoritama se može naći u [Veldhuizen2000].

#### 2.2.3.1.4 Selekcija

U svakoj iteraciji (generaciji) genetičkog algoritma je potrebno odabrati skup jedinki iz kojih će se međusobnim razmnožavanjem kreirati nova populacija. Selekcija se obavlja na osnovu dobrote jedinki, odnosno, jedinke s većom dobrotom imaju veću vjerojatnost da ih se odabere kao roditelje iz kojih će se kreirati nove jedinke. Međutim, u većini slučajeva se u proces selekcije uvodi određena doza stohastičnosti. Razlog tome je činjenica da postoji mogućnost da čak i izrazito loše jedinke sadrže kvalitetan genetički materijal (u dijelu svog kromosoma) te je sa stajališta efikasnosti optimizacije povoljno ostaviti mogućnost da i takve jedinke budu selektirane za daljnju reprodukciju.

Najčešće korišteni operatori selekcije su *jednostavna selekcije* (engl. *roulette wheel selection*) [Baker1987], turnirska selekcija (engl. *tournament selection*) [Goldberg1991] i selekcija bazirana na stohastičkom uzorkovanju (engl. *stohastic universal sampling*) [Baker1987].

#### 2.2.3.1.5 Križanje

Križanje predstavlja proces kojim se iz dvije (ili više) jedinki kreiraju nove jedinke kombiniranjem njihovog genetičkog materijala. Operatori križanja preko kojih se ugrađuje način rekombiniranja genetičkog materijala jedinki roditelja u novostvorene jedinke djece ovise o odabranoj reprezentaciji jedinki.

***Operatori križanja za binarnu reprezentaciju***

U slučaju korištenja standardne reprezentacije preko polja bitova, mogući operatori križanja su križanje u jednoj točki (engl. *single-point crossover*), križanje u više točaka (engl. *multi-point crossover*) i uniformno križanje (engl. *uniform crossover*). Rad svih operatora se zasniva na razmjeni vrijednosti bitova između dvije jedinke roditelja. Za ilustraciju, slijedi opis načina rada operatora križanja u jednoj točki.

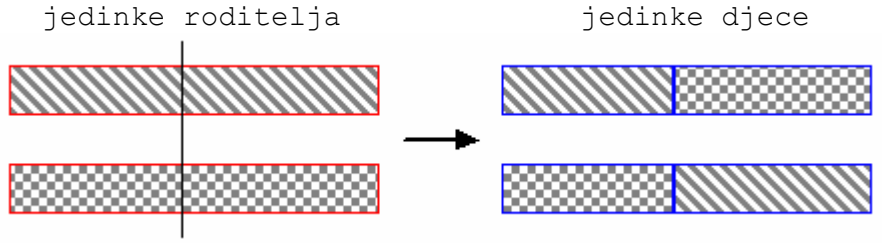
Ukoliko pretpostavimo da su odabrane dvije jedinke roditelja sa sljedećom vrijednošću reprezentacije:

```
jedinka 1     0  1  1  1  0  0  1  1  0  1  0
jedinka 2     1  0  1  0  1  1  0  0  1  0  1
```

i ukoliko se kao točka križanja odabere pozicija 5, generirati će se sljedeće dvije jedinke djece:

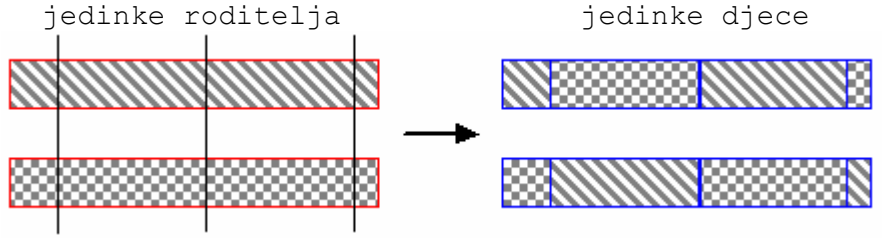
```
dijete 1      0  1  1  1  0| 1  0  0  1  0  1
dijete 2      1  0  1  0  1| 0  1  1  0  1  0
```

Grafička ilustracija procesa križanja je dana na slici 2.1.




**Slika 2.1 Ilustracija operatora križanja u jednoj točki**

Operatori križanja u više točaka rade po sličnom principu, jedino što se izabire više točaka križanja na osnovu kojih se u jedinke djece naizmjenično kopiraju bitovi iz jedinki roditelja. Grafički je taj proces ilustriran na slici 2.2 (u slučaju 3-*point* operatora križanja):




**Slika 2.2 Ilustracija operatora križanja u 3 točke**

Kod primjene uniformnog operatora križanja [Syswerda1989], za svaki bit reprezentacije se slučajno određuje da li će se odgovarajući bit u jedinki djeteta inicijalizirati s bitom iz jednog ili drugog roditelja.

***Operatori križanja za realnu reprezentaciju***

U slučaju korištenja realne reprezentacije moguće je primijeniti načine križanja: diskretna rekombinacija, intermedijarnu rekombinaciju (engl. *intermediate recombination*) ili linijsku rekombinaciju (engl. *line recombination*) [Mühlenbein1993]. Diskretna rekombinacija se može primijeniti u slučaju da reprezentaciju jedinki čini više realnih vrijednosti i tada se slučajnim odabirom bira iz kojega roditelja će se vrijednost realne varijable pridružiti u jedinku djeteta.

Ukoliko pretpostavimo da reprezentaciju jedinke čine tri varijable, sa sljedećim vrijednostima:

```
jedinka 1          12      25       5
jedinka 2         123       4      34
```

i ukoliko pretpostavimo da je slučajnim odabirom odabrano koji roditelj će doprinijeti koju vrijednost u jedinke djece:

```
indeks za dijete 1     2       2       1
indeks za dijete 2     1       2       1
```

nakon provođenja rekombinacije će biti kreirane sljedeće jedinke djece:

```
dijete 1          123      4       5
dijete 2           12      4       5
```

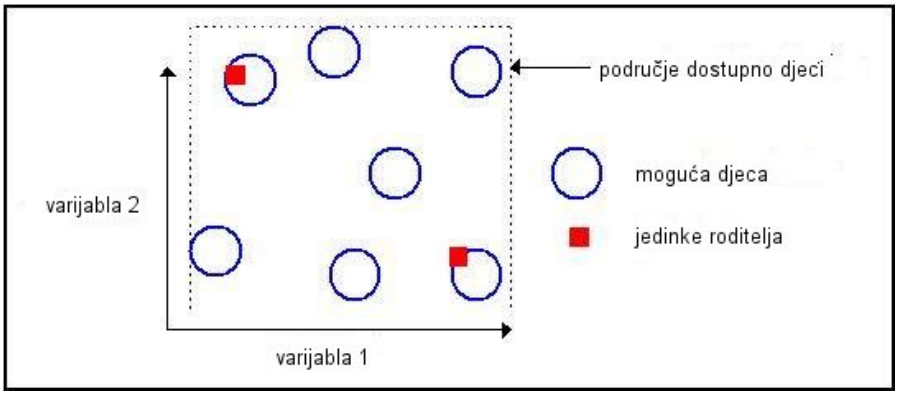
Intermedijarna rekombinacija je primjenjiva samo na realne varijable i kod nje se vrijednosti za jedinke djece odabiru tako da budu unutar ili oko intervala vrijednosti definiranih u jedinkama roditelja. Preciznije, vrijednosti za jedinke djece se izračunavaju na osnovu pravila:

dijete = roditelj 1 + *Alpha* (roditelj 2 - roditelj1).

gdje je *Alpha* skalirajući parametar koji ima slučajnu vrijednost iz intervala [-*d*, 1 + *d*]. Za intermedijarnu rekombinaciju je *d = 0* (odnosno, vrijednosti za djecu će biti strogo unutar intervala definiranog vrijednostima jedinki roditelja), a u slučaju *d* > 0 se radi o proširenoj intermedijarnoj rekombinaciji u kojem slučaju vrijednosti djece mogu biti i izvan intervala definiranog jedinkama roditelja. Dobar izbor predstavlja *d* = 0.25. Grafička ilustracija se vidi na slici 2.3.

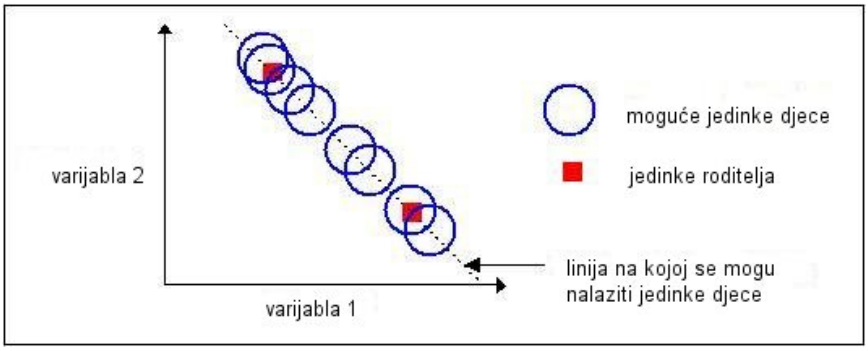

**Slika 2.3 Ilustracija operatora križanja aritmetičkom rekombinacijom**

Korištenjem intermedijarne rekombinacije, jedinke djeteta mogu imati vrijednosti unutar hiperkocke definirane vrijednostima jedinki roditelja kako je ilustrirano na slici 2.4. (u slučaju proširene intermedijarne rekombinacije, dostupna hiperkocka je nešto veća, ovisno o vrijednosti parametra *d*).




**Slika 2.4 Prikaz mogućih vrijednosti jedinki djece nakon provođenja križanja aritmetičkom rekombinacijom**

Linijska rekombinacija predstavlja varijantu intermedijarne rekombinacije s time da se za sve varijable iz reprezentacije koristi ista vrijednost parametra Alpha. U tom slučaju se generiraju jedinke koje se nalaze na pravcu koji prolazi kroz dvije točke definirane jedinkama roditelja.




**Slika 2.5 Prikaz mogućih vrijednosti jedinki djece nakon provođenja križanja linijskom rekombinacijom**

### *Operatori križanja za reprezentaciju preko strukture permutacije*

U slučaju korištenja apstraktnijih matematičkih struktura za reprezentaciju jedinku, potrebno je izgraditi i prikladne operatore križanja. Najčešće korištena matematička struktura za reprezentaciju varijabli je struktura permutacije pomoću koje se može modelirati redoslijed (engl. *ordering*) i pridruživanje (engl. *assignment*). S obzirom da se potreba za takvom reprezentacijom javlja u mnogim kombinatorijalnim problemima (TSP, QAP, JSSP), razvijeni su i mnogobrojni operatori križanja.

Detaljan opis načina rada pojedinih operatora se može naći u referencama, a ovdje će se navesti samo popis najčešće korištenih:

- Parcijalno preslikano križanje (engl. *partially mapped crossover* - u literaturi poznat po skraćenici PMX) [Goldberg1985]
- Cikličko križanje (engl. *cycle crossover* - CX) [Oliver1987]

- Redno križanje (engl. *order crossover* – OX) za koji su definirane dvije varijante, OX#1 [Davis1985] i OX#2 [Syswerda1991]
- Križanje po bridovima (engl. *edge crossover*) [Whitley2000]

#### 2.2.3.1.6 Mutacija

Nakon kreiranja jedinki djece pomoću operatora križanja, novokreirane jedinke prolaze kroz proces mutacije. Cilj primjene operatora mutacije je povećanje raznolikosti u populaciji pomoću uvođenja slučajnih varijacija u članovima populacije. Operator mutacije je za razliku od operatora križanja unarni, odnosno djeluje na jednu jedinku. Operator mutacije također ovisi o vrsti reprezentacije nad kojom operira.

Kod korištenja binarne reprezentacije najčešće se koristi tzv. *flip-bit* mutacija kod koje se zasebno promatra svaki bit u reprezentaciji te se s određenom vjerojatnošću obavlja inverzija njegove vrijednosti (iz 0 u 1, odnosno iz 1 u 0).

Operatori mutacije za realnu reprezentaciju se uglavnom svode na stohastičko mijenjanje vrijednosti varijable. Pri tome možemo identificirati dva tipa mutacije, ovisno o distribuciji vjerojatnosti po kojoj se vrši odabir novih vrijednosti: uniformnu i neuniformnu. Kod uniformne mutacije se nova vrijednost uniformno odabire iz dopuštenog intervala, dok se kod neuniformne na trenutnu vrijednost varijable dodaje vrijednost koja je slučajno odabrana iz Gaussove distribucije sa srednjom vrijednošću 0 i korisnički zadanom standardnom devijacijom. Time se postiže da će u većini slučajeva promjena biti mala, ali postoji i određena, iako mala, vjerojatnost da će prilikom mutacije vrijednost doživjeti veliku promjenu.

Kod operatora mutacije za reprezentaciju permutacije je situacija nešto kompliciranija s obzirom da se prilikom obavljanja mutacije mora paziti da permutacija ostane valjana. Najčešće se koriste sljedeća tri operatora, koja su prvi put definirana u [Syswerda1991]:

- mutacija zamjene (engl. *swap mutation*) - odaberu se dvije pozicije unutar permutacije, te se zamjene odgovarajuće vrijednosti
  - 1 **2** 3 4 **5** 6 7 8 9 → 1 **5** 3 4 **2** 6 7 8 9
- mutacija ubacivanjem (engl. *insert mutation*) – kod ovog operatora se slučajno odabiru dvije pozicije, te se permutacija ažurira tako da se vrijednosti s tih pozicija nađu jedna do druge
  - 1 **2** 3 4 **5** 6 7 8 9 → 1 **2 5** 3 4 6 7 8 9
- mutacija premetanjem (engl. *scramble mutation*) – kod ove mutacije se za cijeli niz, ili njegov dio, slučajno ispremještaju pozicije
  - 1 **2 3 4 5** 6 7 8 9 → 1 **3 5 4 2** 6 7 8 9

## 2.2.4 Optimizacija rojem čestica

Optimizacija rojem čestica (engl. *Particle swarm optimization* - PSO) predstavlja jednu od tehnika optimizacije baziranih na *inteligenciji roja* (engl. *swarm intelligence*). Radi se o tehnici umjetne inteligencije koja je bazirana na studijama kolektivnog ponašanja u decentraliziranim samoorganizirajućim sistemima, a sam termin su uveli Beni i Wan [Beni1989]. Sistemi bazirani na inteligenciji jata se obično sastoje od populacije jednostavnih *agenata* koji su u interakciji jedan s drugim i sa svojom okolinom. Iako obično nema centralizirane kontrolne strukture koja bi diktirala način ponašanja individualnih agenata, lokalne interakcije među agentima često vode pojavljivanju globalno koordiniranog ponašanja.

PSO tehnika se bazira na analogiji s ponašanjem roja insekata, odnosno jata ptica ili riba. Ukoliko jedna od jedinki pronađe povoljan pravac kretanja, npr. prema hrani ili zaštiti od neprijatelja, druge jedinke u jatu će vrlo brzo početi slijediti tu jedinku, čak i u slučaju da se nalaze na sasvim drugoj strani jata.

U kontekstu optimizacije, navedena analogija se realizira modeliranjem agenata pomoću čestica u višedimenzionalnom prostoru koje imaju definiranu poziciju i brzinu. Te čestice „lete“ kroz konfiguracijski prostor rješenja i pamte najbolju poziciju kroz koju su prošle. Članovi jata međusobno prenose informacije o dobrim pozicijama jedan drugome i prilagođavaju svoje pozicije i brzine podacima o nađenim dobrim pozicijama. Komunikacija se provodi na dva načina:

- preko globalno najboljeg rješenja („pozicije“) koja je poznata svim članovima jata
- preko lokalnih najboljih rješenja koja su poznata u određenom susjedstvu jata

Ažuriranje pozicija i brzina članova jata se provodi sljedećim formulama u svakoj iteraciji optimizacijskog postupka:

$$x \leftarrow x + v$$

$$v \leftarrow wv + c_1 r_1 (\hat{x} - x) + c_2 r_2 (\hat{x}_g - x)$$

gdje su značenja pojedinih parametara sljedeća:

- $v$ je brzina čestice
- $w$ je inercijska konstanta (iskustveno je utvrđeno da je prikladan odabir vrijednosti nešto manjih od 1)
- $c_1$ i $c_2$ su konstante koje definiraju koliko se čestice usmjeravaju prema dobrim pozicijama (prikladne su vrijednosti oko 1)
- $r_1$ i $r_2$ su slučajne vrijednosti u rasponu [0 , 1]
- $\hat{x}$ predstavlja najbolju poziciju koju je čestica posjetila
- $\hat{x}_g$ predstavlja najbolju globalno nađenu poziciju.

Pseudo-kod algoritma je sljedeći:

inicijaliziraj $x$ i $v$ za svaku česticu na slučajnu vrijednost
inicijaliziraj svaki $\hat{x}$ na trenutnu poziciju
inicijaliziraj $\hat{x}_g$ na poziciju koja ima najbolji fitnes u jatu
dok je $\hat{x}_g$ ispod postavljene granice i nije dostignut preddefinirani broj iteracija
    za svaku česticu
        ažuriraj $x$ prema gornjoj jednadžbi
        izračunaj fitnes za novu poziciju
        ako je izračunati fitnes bolji od fitnesa od $\hat{x}$, zamijeni $\hat{x}$
        ako je izračunati fitnes bolji od fitnesa za $\hat{x}_g$, zamijeni $\hat{x}_g$
        ažuriraj $v$ prema gornjoj jednadžbi

Pored PSO tehnike, na inteligenciji jata su bazirani i još neki postupci optimizacije a najpoznatiji su optimizacija pomoću mravljih kolonija (engl. *ant colony optimization*) [Dorigo2004] i tehnika stohastičkog difuznog pretraživanja (engl. *stohastic diffusion search*) [Bishop1989].

# 3. Programska podrška za optimizaciju

Optimizacijski problemi se mogu rješavati različitim tehnikama. Jednostavni problemi matematičke optimizacije realnih funkcija se mogu riješiti teorijskim putem, korištenjem samo papira i olovke, uz primjenu znanja matematičke analize i algebre i eventualno korištenje kalkulatora za proračun konkretnih vrijednosti iz dobivenih formula. Slično vrijedi i za jednostavne probleme linearnog programiranja gdje se optimizacija pomoću simpleks algoritma također može provesti samo na papiru, a isto se može reći i za jednostavne probleme iz teorije grafova.

Međutim, za praksu su relevantni samo značajno složeniji problemi, čije rješavanje obavezno zahtijeva implementaciju prikladnog računalnog algoritma. S obzirom da je izgradnja optimizacijskog računalnog algoritma netrivijalna operacija koja zahtijeva dobro poznavanje tehnika programiranja i odabranog programskog jezika, programska podrška koja bi olakšala njihovu izgradnju i primjenu bi bilo vrlo korisna za istraživače optimizacijskih problema u cjelini.

Takav zadatak na prvi pogled izgleda kao nemoguća misija budući da se u konačnici računalni algoritam mora izgraditi u nekom programskom jeziku i prevesti u izvršni kôd koji se stvarno može izvesti na računalu. Stoga je jedina mogućnost za rješavanje tog problema definiranje novog „super-optimizacijskog“ jezika koji će istovremeno biti i jednostavan za korištenje i dovoljno moćan i fleksibilan da se u njemu mogu izgraditi efikasni algoritmi. Iako postoje jezici koji pretendiraju na zadovoljavanje upravo takvih karakteristika, primjerice OPL Studio [VanHentenryck1999], AIMMS [Bisschop1993], AMPL [Fourer1993], oni pate i od nekih nedostataka od kojih je najveći ograničeno područje primjene. S obzirom da se sam jezik mora sastojati od konačnog broja osnovnih građevnih elemenata jasno je da je vrlo teško postići općenitost potrebnu za mogućnost izgradnje različitih optimizacijskih algoritama. U svezi s tim se može primijetiti gotovo ekskluzivna posvećenost navedenih optimizacijskih jezika izgradnji egzaktnih algoritama namijenjenih za rješavanje standardnih mješovito-cjelobrojnih programa. Uzrok tome vjerojatno leži u činjenici da je to vrlo zrelo i matematički dobro proučeno područje optimizacije. Nepostojanje mogućnosti za modeliranje kombinatorijalnih problema i heurističkih algoritama je svakako veliki nedostatak navedenih jezika.

Iako opisana situacija ne daje povoda za preveliki optimizam, u prethodnom izlaganju zanemarena je jedna bitna činjenica. Naime, velika većina postojećih optimizacijskih algoritama je već izgrađena! U svjetlu te činjenice, posao istraživača izgleda jednostavan. Sve što je potrebno napraviti je pronaći implementaciju traženog algoritma i primijeniti je na svoj problem. Prva prepreka u takvom scenariju je pronalaženje i priskrbljivanje postojeće implementacije. Usprkos Internetu i njegovim potencijalima za jednostavno dijeljenje programskog kôda, za većinu algoritama je vrlo jednostavno pronaći samo *opis* načina rada, najčešće u znanstvenom članku kojim je objavljen pronalazak novog algoritma, dok je do same implementacije, odnosno programskog kôda preko kojeg je realiziran algoritam, puno teže doći. Pored toga, samo dobavljanje programskog kôda nije dovoljno. S obzirom da se taj kôd mora prevesti u

izvršni kôd prije provođenja same optimizacije, istraživač ili mora imati pristup istom ili sličnom razvojnom okruženju ili mora posjedovati znanja koja će mu omogućiti da dobavljeni programski kôd prilagodi za prevođenje u svom preferiranom razvojnom okruženju. Imajući u vidu raznolikost razvojnih okruženja i jezika koje istraživači koriste u implementaciji optimizacijskih algoritama, nakon početnog „heureka“ istraživači vrlo često doživljavaju hladan tuš kad shvate da ih prije iskorištavanja nađene implementacije čeka težak posao manualnog prilagođavanja dostupnom razvojnom okruženju (počevši od ručnog podešavanja *make* datoteka kod C i C++ implementacija).

Međutim, čak i kad se uspješno može dobaviti implementacija zanimljivog optimizacijskog algoritma i upogoniti na dostupnom razvojnom okruženju, javlja se novi problem. Naime, vrlo je vjerojatno da postojeća implementacija algoritma ne odgovara točno obliku problema s kojim je istraživač konkretno suočen. Odnosno, potrebno je provesti manje ili više zahtjevno prilagođavanje algoritma konkretnom problemu. Zahtjevnost te prilagodbe ovisi o konkretnoj situaciji. Ona može biti jednostavna, čak i trivijalna, ukoliko je, primjerice, jedino potrebno prilagoditi format datoteke s podacima koja definira problem. Tipičan ovakav slučaj je prilagodba formata datoteke s matricom udaljenosti  kod problema trgovačkog putnika.

Značajno je složeniji slučaj kad se mora prilagoditi reprezentacija varijabli u definiciji optimizacijskog problema. Npr., moguća je situacija u kojoj je istraživaču dostupan programski kôd s implementacijom genetičkog algoritma za optimiranje realne funkcije, ali je u implementaciji predviđeno da postoje samo dvije varijable problema s točno definiranim intervalom vrijednosti. Iako je proširenje takve implementacije na općeniti slučaj s $n$ varijabli relativno jednostavno, situacija se lako može zakomplicirati, npr. zahtjevom za uključivanjem i cjelobrojnih varijabli kao ulaznih veličina problema, što zahtjeva ugradnju novih vrsta operatora što već nije trivijalan zahvat u postojeću implementaciju.

Dodatno značajno ograničenje u primjenjivosti konkretne implementacije optimizacijskog algoritma predstavlja odabir reprezentacije varijabli koja se koristi u toj implementaciji. Realne i cjelobrojne varijable se univerzalno preslikavaju u programske tipove *float* i *int*, ali se postavlja pitanje programskog modeliranja raspona njihovih vrijednosti (koji također ima univerzalno primjenjivano rješenje – korištenje dva broja koji predstavljaju donju i gornju granicu intervala). Kod genetičkih algoritama se pojavljuje i bitovna reprezentacija varijabli, uz mogućnost korištenja Gray-evog kôda za preslikavanje bitovne u cjelobrojnu reprezentaciju što dodatno komplicira prilagodbu dostupne implementacije konkretnom problemu.

Problem se u svoj svojoj veličini pojavljuje kod definiranja reprezentacija za kombinatorijalne strukture. Strukture permutacija, grafova i pridruživanja su matematički vrlo bogate i dopuštaju različite realizacije u programskom kôdu. Prilagodba postojećih algoritama za korištenje drugačijih reprezentacija varijabli zahtijeva duboko ulaženje u strukturu programskog kôda što ne treba shvatiti olako. Problem nepostojeće dokumentacije koji je inherentan svim manjim softverskim projektima, u koju kategoriju spada i većina dostupnih implementacija, samo dodatno otežava situaciju, osim kod nekih rijetkih izuzetaka..

Iako su navedeni problemi ozbiljna prepreka za široko iskorištavanja dostupnih implementacija optimizacijskih algoritama, njihovo rješavanje ipak zahtjeva samo određenu količinu relativno manualnog rada što u većini situacija ne predstavlja nepremostivu prepreku. Pravi problem se javlja kod iskorištavanja postojećih implementacija na rješavanje stvarno složenih problema iz prakse koji, ponajviše zbog specifičnosti svog skupa ulaznih veličina, nisu prikladni za rješavanje postojećim implementacijama. Ovdje su moguće dvije situacije. U prvom slučaju radi se o doista jedinstvenom problemu koji zahtijeva posvećenu implementaciju i tada istraživaču ne preostaje ništa drugo nego tu implementaciju i osobno izgraditi. Jasno je da ova situacija ne zadovoljava osnovnu pretpostavku da već postoji implementacija prikladnog algoritma. Međutim, doista jedinstveni optimizacijski problemi se u praksi relativno rijetko pojavljuju. Puno češći slučaj predstavlja situacija u kojoj je istraživač suočen sa rješavanjem problema koji je sličan problemu za koji već postoji implementacija optimizacijskog algoritma, ali se razlikuje u određenim detaljima.

Tipičan primjer ove situacije je rješavanje problema satnice (engl. *timetabling problem*). Iako je sam problem relativno dobro definiran, u praksi se kod njegovog rješavanja javljaju različite varijacije, poglavito uzrokovane različitim ustrojem obrazovnih institucija za koje se treba definirati satnica. Zbog složenosti problema satnice, samo dio potrebne implementacije se odnosi na izgradnju optimizacijskog algoritma, dok se, vjerojatno i veći dio, odnosi na pripremu podataka za algoritam (modeliranje i dohvaćanje podataka o predmetima, nastavnom planu, grupama studenata/učenika, profesorima/nastavnicima i dvoranama/učionicama) te vizualizaciju nađenih rješenja radi njihove analize i odabira najprihvatljivijeg. U takvoj situaciji postojeće implementacije postaju gotovo neiskoristive jer je u većini slučajeva potencijalno iskoristiv programski kôd posvećen provođenju same optimizacije u velikoj mjeri isprepleten s programskim kodom posvećenim definiranju samog problema, koji je nužno specifičan za slučaj za koji je implementacija originalno i napravljena (npr. podaci se uzimaju iz Informix baze sa određenom strukturom tablica, a vizualizacija rješenja se obavlja preko GUI sučelja izgrađenog korištenjem Java Swing biblioteke).

Dodatnu varijantu predstavlja situacija u kojoj je potrebno riješiti kombinirani problem u smislu da se problem sastoji od više različitih i jednostavnijih potproblema. Ova situacija se vrlo često javlja u poslovnim primjenama gdje se radi optimizacije poslovanja istovremeno mora riješiti nekoliko (pot)problema koji predstavljaju različite aspekte poslovanja. Univerzalan primjer je rješavanje problema transporta robe od skladišta do maloprodajnih trgovina gdje se susreću problem određivanja rute vozila (VRP) i problem određivanja načina pakiranja kamiona (varijanta problema naprtnjače). Iako za svaki od ta dva problema postoje mnogobrojni algoritmi sa pripadnim implementacijama, malo je vjerojatno da će se naći implementacija koja je direktno primjenjiva na takav problem, pogotovo ukoliko se uzme u obzir da i ovdje specifičnost ulaznih podataka koji definiraju problem igra jednako značajnu ulogu.

## 3.1 Različiti „scenariji optimizacije“

Iz prethodnog izlaganja je razvidno da se istraživači na području optimizacije mogu naći u različitim situacijama s obzirom na programsku podršku koja im je dostupna za rješavanje danog optimizacijskog problema. Cilj ovog potpoglavlja je dati podrobniji opis mogućih scenarija kod iskorištavanja programske podrške za rješavanje optimizacijskih problema.

**Gotova aplikativna rješenja – „*off-the-shelf*“ scenarij**

Idealan scenarij za svakog optimizacijskog istraživača čini situacija u kojoj je dostupna gotova i kompletna programska podrška za rješavanje zadanog problema. Nažalost, zbog već opisane raznolikosti optimizacijskih problema, to je i vrlo rijedak scenarij. Pored toga, i sama kvaliteta dotične programske podrške igra značajnu ulogu. Za određena, uglavnom zrela područja optimizacije poput linearnog programiranja, postoji relativno velik broj profesionalnih programskih paketa koji su primjenjivi za rješavanje najsloženijih problema iz pokrivenih područja optimizacije i koji svojom kvalitetom i popratnom dokumentacijom istraživaču značajno olakšavaju posao. Pored takvih aplikativnih rješenja koja svojom cijenom odražavaju kvalitetu i trud utrošen u njihovu izgradnju, prilikom potrage za već postojećom programskom podrškom ne treba zanemariti ni *shareware* rješenja. Iako rješenja bazirana na principu otvorenog kôda ne mogu parirati profesionalnim softverskim paketima, i to poglavito u pogledu dostupne dokumentacije i korisničke potpore, njihova prednost leži u raznovrsnosti dostupnih rješenja što povećava vjerojatnost nalaženja gotovog rješenja koje je primjenjivo na zadani problem.

Jedini (potencijalni) nedostatak primjene posve gotovih aplikativnih rješenja kod rješavanja zadanog optimizacijskog problema predstavlja složenost njihove integracije s drugim IT sustavima koji su relevantni za provođenje optimizacije. Ovaj nedostatak nije prisutan u situaciji kada je jedini cilj istraživača riješiti optimizacijski problem za konkretne vrijednosti njegovih parametara, uz uvjet da je izlaz (format rješenja) koji daje korišten softver zadovoljavajući.

Međutim, kod složenijih primjena u praksi, samo rješavanje optimizacijskog problema najčešće čini samo jedan dio šireg sustava čiji je cilj kvalitetnija potpora poslovnom odlučivanju. Najčešća je situacija u kojoj se ulazni podaci za proces optimizacije generiraju u jednom dijelu sustava, a nakon provedene optimizacije se dobivena rješenja ponovno uvoze u neki drugi (ili možda isti) dio sustava. S obzirom na složenost ugradnje međuprocesne komunikacije i činjenicu da su samo rijetki softveri za optimizaciju predviđeni za tako nešto, komunikacija između optimizatora i ostalih dijelova sustava se najčešće izvodi preko vanjskih datoteka, što slabi integriranost sustava.

**Prilagodba postojeće implementacije**

U određenim slučajevima, moguća strategija prilikom rješavanja danog optimizacijskog problema je prilagođavanje postojeće implementacije. Kako je već opisano, složenost zahtijevanih preinaka je vrlo varijabilna i u principu predstavlja

osnovni kriterij kod procjenjivanja da li je povoljno pristupiti takvom postupku. Očito je da složenost postojeće implementacije algoritma značajno utječe na tu procjenu, te je ovaj scenarij primjenjiv uglavnom kod jednostavnijih problema.

**Korištenje postojeće biblioteke**

Problemi s integracijom gotovih aplikativnih rješenja za optimizaciju u šire IT sustave se mogu efikasno riješiti korištenjem biblioteka s ugrađenim programskim komponentama za provođenje optimizacije. S obzirom da prilikom korištenja biblioteka sav generirani izvršni kôd vezan uz optimizaciju postaje dio aplikacije unutar koje se koristi biblioteka (bilo direktnim uključivanjem kod korištenja statičkih biblioteka, bilo indirektnim uključivanjem vanjskih dinamičkih biblioteka – engl. *dynamicaly loaded libraries*, *dll*), integracija optimizacijskog kôda je potpuna te ga se može koristiti i njime upravljati direktno na programskom nivou izvornog koda.

Presudni faktor kod ovoga scenarija je dostupnost kvalitetnih biblioteka sa ugrađenim optimizacijskim algoritmima. S obzirom na jednostavniju izgradnju, jer za razliku od gotovih aplikativnih rješenja nemaju ugrađeno vizualno (GUI) sučelje, ovakva rješenja su mnogobrojnija u odnosu na kategoriju gotovih aplikativnih rješenja i predstavljaju odličan izbor ukoliko su prikladna za rješavanje danog optimizacijskog problema. Dostupne biblioteke značajno variraju u svojoj složenosti i ugrađenim mogućnostima, a sažeti opis nekih od njih je dan kasnije u poglavlju.

**Korištenje postojećih razvojnih okosnica (engl. *framework*)**

Nešto napredniju inačicu programske biblioteke čini koncept razvojne okosnice (u praksi se često koristi i termin *okvir za razvoj*). Iako se sa stajališta računarske znanosti biblioteka i razvojna okosnica razlikuju po svojim karakteristikama, a u nekim pogledima su i slični, najvažnija razlika leži u mogućnostima proširenja. Pojednostavljeno govoreći, biblioteka predstavlja zatvoreni skup programskih komponenti (bilo funkcija, bilo razreda) koji se može iskoristiti prilikom izgradnje informacijskog sustava. Razvojna okosnica, s druge strane, čini skup programskih komponenata čija je osnovna namjena definiranje *potporne strukture* informacijskog sustava unutar koje korisnik razvojne okosnice ugrađuje svoje programske komponente koje zadovoljavaju sučelja definirana u samoj razvojnoj okosnici. To znači da kontrolu tijeka izvođenja programa ima razvojna okosnica koja preko zadanih sučelja poziva naknadno ugrađene programske komponente za obavljanje određenih zadataka (karakteristika koja je opisana u poznatom *Hollywood* principu: "*Don't call us, we'll call you*" [Vlissides1996]).

Sa stajališta programske podrške za optimizaciju, glavnu karakteristika razvojne okosnice čini mogućnost proširenja s novim programskim komponentama kojima se modeliraju nove vrste problema, algoritama i pripadnih operatora. S obzirom na već opisanu raznolikost koncepata koji se javljaju u domeni optimizacije, većina dostupnih biblioteka u stvari barem djelomično spada pod kategoriju razvojne okosnice jer, u različitoj mjeri, uglavnom pružaju mogućnost ugradnje novih vrsta optimizacijskih komponenti. Ono što ih razlikuje od pravih okvira za razvoj aplikacija jest činjenica da nisu namijenjeni razvoju čitavih informacijskih sustava već su poglavito namijenjeni

omogućavanju efikasne i fleksibilne izgradnje dijela sustava namijenjenog provođenju same optimizacije.

## 3.2 Tražene karakteristike optimizacijskog okruženja

Svi informacijski sustavi, pa tako i oni koji spadaju u kategoriju programske podrške za optimizaciju, se mogu okarakterizirati nekim univerzalnim svojstvima poput jednostavnosti održavanja (engl. *maintainability*) i prijaznosti za korištenje (engl. *user friendliness*). U slučaju programske podrške za optimizaciju možemo definirati dodatni skup relevantnih karakteristika na koji se mora obratiti pažnja prilikom njihove izgradnje. Te karakteristike su okvirno već navedene u uvodu ove disertacije i sljedeće su:

- jednostavnost korištenja

  Jedna od osnovnih karakteristika po kojima se procjenjuje kvaliteta i iskoristivost bilo kakve programske podrške. U današnje vrijeme se jednostavnost (prijaznost) korištenja primarno ocjenjuje kroz kvalitetu grafičkog korisničkog sučelja. Stoga mogućnost rješavanja optimizacijskih problema preko GUI sučelja ima značajnu prednost u odnosu na standardnu *edit-compile-link-execute* paradigmu kod koje „optimizacijsko okruženje" čine editor, prevodilac i linker.

- široka primjenjivost

  Općenitost optimizacijskog okruženja se primarno ocjenjuje kroz mogućnost iskorištavanja za rješavanje širokog skupa različitih optimizacijskih problema.

- ugrađen široki skup optimizacijskih komponenti

  Iako raznolikost optimizacijskih problema i postupaka za njihovo rješavanje negira mogućnost izgradnje „kompletnog" optimizacijskog okruženja, sa stajališta korisnika optimizacijskog okruženja je svejedno povoljno da korišteno okruženje dolazi sa ugrađenim skupom optimizacijskih komponenti, koje su „*out-of-the-box*" primjenjive na neke standardne probleme optimizacije.

- proširivost

  Uzimajući u obzir gore navedenu nemogućnost izgradnje „kompletnog" optimizacijskog okruženja, jasno je da je proširivost s novim (ili novo-izgrađenim) optimizacijskim komponentama vrlo važno svojstvo općenitog optimizacijskog okruženja.

- ugrađena potpora za analizu i vizualizaciju rezultata

  Kod rješavanja optimizacijskih problema nužno je imati mogućnost analize, usporedbe i vizualizacije dobivenih rješenja. Stoga je dodatni zahtjev na optimizacijsko okruženje ugradnja programskih komponentama za vizualizaciju i analizu rezultata koje su generičke u smislu da se mogu iskoristiti za vizualizaciju rezultata dobivenih rješavanjem različitih optimizacijskih problema.

- jednostavnost izgradnje novih komponenti

  Jedan od osnovnih preduvjeta široke iskoristivosti optimizacijskog okruženja je i jednostavnost izgradnje novih optimizacijskih komponenti, i to poglavito u smislu jednostavnosti (odnosno što manje složenosti) primijenjenih tehnika programiranja kod izgradnje tih komponenti.

- efikasnost izvršavanja

  S obzirom na računsku zahtjevnost provođenja optimizacije kod iole složenijih problema, bitno je da optimizacijsko okruženje ugradnjom gore navedenih karakteristika i mogućnosti ne kompromitira efikasnost izračunavanja (odnosno, da to čini u minimalnoj mogućoj mjeri).

U sljedećem odjeljku će se dati detaljniji opis navedenih karakteristika.

## 3.2.1 Jednostavnost korištenja

Iako je jednostavnost korištenja određenog softvera donekle relativna i subjektivna kategorija, kod programske podrške namijenjene optimiranju se s tog stajališta izgrađena rješenja mogu klasificirati u dvije skupine. Prvu skupinu čine programska rješenja koja zahtijevaju korištenje razvojnog okruženja za neki programski jezik, odnosno njihovo iskorištavanje se obavlja programski, na nivou izvornog koda (uz korištenje standardnog *edit-compile-link-execute* ciklusa). U ovu skupinu spadaju različite izgrađene biblioteke i okviri za razvoj i primjeri takvih rješenja su mnogobrojni i široko dostupni. Drugu skupinu čine kompletno izgrađena rješenja kod kojih se sve njihove mogućnosti mogu iskoristiti putem vizualnog sučelja, znači bez potrebe za programiranjem u nekom programskom jeziku.

Iako je jasna prednost situacije u kojoj se problem može riješiti samo odabirom opcija na ekranu računala, isto je tako jasno da takva rješenja pate od problema nedovoljno široke primjenjivosti s obzirom da nije moguća ugradnja novih optimizacijskih komponenti! Razlog tome leži u činjenici da bi ugradnja novih komponenti zahtijevala modifikaciju vizualnog sučelja kako bi te komponente preko njega bile dostupne, što je tehnološki vrlo zahtjevan zadatak.

Proširivanje gotovih rješenja/aplikacija s novim optimizacijskim komponentama također zahtijeva mogućnost dinamičkog rukovanja s programskim komponentama unutar koji su realizirani dani optimizacijski problemi, algoritmi i operatori. Iako te mogućnosti odavno postoje u različitim razvojnim okruženjima i na Windows (*dll* biblioteke) i na UNIX operacijskim sustavima, tehnička složenost njihovog iskorištavanja je priječila njihovu širu primjenu u području optimiranja.

Srećom, pojava najnovije generacije razvojnih okruženja (.NET i Java) je značajno olakšala tehničke aspekte izgradnje takvog optimizacijskog okruženja, što je bio i jedan od glavnih motiva za izradu ove disertacije. Međutim, usprkos značajnom napretku u jednostavnosti izgradnje takvog *dinamički-podesivog* optimizacijskog okruženja, ostaje kruta činjenica da rješavanje novog optimizacijskog problema ili primjena novog optimizacijskog postupka zahtijeva razvoj novih programskih komponenti, znači - programiranje.

Takva situacija je neizbježna s obzirom da proširivost optimizacijskog okruženja igra značajnu ulogu kod određivanja moguće širine primjene. Stoga bi idealno optimizacijsko okruženje trebalo pružiti mogućnost korištenja na dvije razine. U situaciji kada se rješava problem za koji su odgovarajuće potrebne optimizacijske komponente već ugrađene u okruženje, potrebno je omogućiti njihovo potpuno iskorištavanje kroz grafičko sučelje. Na toj razini se rješavanje optimizacijskog problema svodi na odabir konkretnih optimizacijskih komponenti, podešavanje njihovih parametara i analizu dobivenih rezultata, sve kroz vizualno sučelje, bez potrebe za ikakvim programiranjem.

U situaciji kada se rješava optimizacijski problem za koji potrebne optimizacijske komponente nisu već dio optimizacijskog okruženja, potrebno ih je naprije izgraditi, zadovoljavajući pritom zahtjeve koje nameće okruženje kako bi se unutar njega mogle iskoristiti, te dinamički ugraditi (učitati) u okruženje nakon čega one postaju dostupne preko vizualnog sučelja optimizacijskog okruženja.

### 3.2.2 Ugrađeni alati za vizualizaciju i analizu

Kao što je već ranije navedeno, prilikom rješavanja složenih optimizacijskih problema mogućnost vizualizacije i analize rješenja je vrlo važna. S jedne strane, kod određenih problema je važno imati mogućnost vizualizacije dobivenih rješenja jer se time daje kvalitetnija informacija istraživaču tijekom provođenja procesa optimizacije. Tipičan primjer je vizualizacija Pareto plohe kod višekriterijskih problema koja je nužan preduvjet za kvalitetan odabir konačnog rješenja jer se njenom vizualizacijom olakšava pronalaženje optimalnog kompromisa s obzirom na različite kriterije. Drugi mogući primjer je vizualizacija različitih dobivenih rasporeda kod rješavanja problema satnice. Iako se taj problem najčešće modelira kao jednokriterijski problem što bi izbor konačnog rješenja trebalo učiniti trivijalnim, vrlo je teško unutar funkcije cilja ugraditi i modelirati sve aspekte kvalitetnog rasporeda. Stoga je nužno pružiti istraživaču (satničaru) mogućnost vizualne usporedbe različitih rješenja kako bi ih on mogao ocijeniti u skladu sa svojim iskustvom te donijeti konačnu odluku o odabranom najboljem rješenju.

Pored mogućnosti vizualizacije dobivenih rješenja, jasno je da je za istraživača prilikom rješavanja optimizacijskog problema važno imati mogućnost njihove analize i usporedbe. To pogotovo vrijedi kod rješavanja problema pomoću heurističkih optimizacijskih postupaka, odnosno šire govoreći, kod primjene svih optimizacijskih postupaka koji su ovisni o vrijednostima nekih parametara. U takvim situacijama je jedna od vremenski najzahtjevnijih aktivnosti istraživanje ovisnosti kvalitete dobivenih rješenja o vrijednostima parametara algoritma, odnosno nalaženje optimalnih parametara algoritma za dani problem. Mogućnost automatiziranog obavljanja optimizacije za različite vrijednosti parametara algoritama te prikupljanje, vizualizacija i usporedba tako dobivenih rezultata predstavlja značajnu pomoć za istraživača u procesu optimizacije.

### 3.2.3 Proširivost i široka primjenjivost

Ove dvije vrlo bitne karakteristike općenitog optimizacijskog okruženja su usko povezane i to iz već često ponavljanog razloga: s obzirom na raznolikost optimizacijskih

problema i algoritama i nemogućnost izgradnje kompletnog optimizacijskog okruženja, jasno je da je preduvjet za široku primjenjivost izgrađenog optimizacijskog okruženja mogućnost njegovog proširivanja s novim optimizacijskim komponentama.

Međutim, ultimativni cilj izgradnje potpuno proširivog optimizacijskog okruženja unutar kojeg se mogu ugraditi programske komponente kojima se može modelirati bilo kakav optimizacijski problem ili algoritam je vrlo teško dostižan. Razlog tome leži ponajprije u raznolikosti paradigmi izgradnje različitih postupaka optimizacije o čemu je već bilo riječi u drugom poglavlju gdje su detaljno opisani i klasificirani pojedini optimizacijski postupci. Ukoliko usporedimo primjerice, rekurzivne i iterativne algoritme, jasno je da je način njihove izgradnje vrlo različit, što vodi i različitim načinima iskorištavanja i upravljanja s programskim komponentama preko kojih su dotični algoritmi realizirani. Dodatnu razliku nalazimo kod koncepta operatora kao osnovnog elementa nekih optimizacijskih postupaka. Neki optimizacijski postupci poput genetičkih algoritama se izrazito oslanjaju na definirane operatore za obavljanje ključnog dijela optimizacijskog postupka dok neki, poput simpleksnog algoritma ili postupaka dinamičkog programiranja uopće ne koriste taj koncept u svojoj implementaciji. Stoga se kod izgradnje optimizacijskog okruženja potrebno ograničiti na određeni skup podržanih paradigmi oblikovanja optimizacijskih postupaka kako bi se zadržala koherentnost arhitekture i dizajna razvijenog informacijskog sustava.

Sa stajališta dizajna i arhitekture aplikacije kroz koju je realizirano optimizacijsko okruženje, mogućnost proširenja funkcionalnosti aplikacije s novim programskim komponentama se može realizirati na dva načina: statički i dinamički (precizniji engleski termin bi bio *run-time* proširivost).

Pod statičkim proširenjem se podrazumijeva iskorištavanje (vanjskih) programskih komponenti koje su implementirane kao statičke biblioteke (standardno s .lib ili .obj ekstenzijom u slučaju C/C++ programskog jezika). Iskorištavanje novoizgrađenih komponenti zahtjeva ponovno prevođenje aplikacije kako bi ih povezivač (engl. *linker*) ugradio u izvršnu aplikaciju. Sa stajališta prijaznosti korištenja nedostaci takvog rješenja su očigledni jer se od korisnika optimizacijskog okruženja zahtijevaju dodatna tehnička znanja vezana uz postavljanje razvojnog okruženja i provođenje cijelog *edit-compile-link-execute* ciklusa.

Dinamičko (*run-time*) iskorištavanje se oslanja na koncept dinamički učitane biblioteke (*dll*) čije iskorištavanje ne zahtijeva ponovno prevođenje aplikacije koja želi iskoristiti funkcionalnost ugrađenu u biblioteku. U tom slučaju se proces izgradnje dodatnih optimizacijskih programskih komponenti razdvaja (engl. *decouple*) od izgradnje same aplikacije koja predstavlja optimizacijsko okruženje čime se značajno olakšava posao graditeljima novih optimizacijskih komponenti. Sa stajališta arhitekture, takvo se rješenje stoga sastoji od glavne aplikacije koja realizira optimizacijsko okruženje s ugrađenom potrebnom funkcionalnošću i skupa programskih komponenti realiziranih preko *dll*-ova koji predstavljaju različite optimizacijske komponente, i koje se unutar aplikacije mogu iskoristiti te se s njima može upravljati.

Mogućnost definiranja razreda sučelja (engl. *interface class*) koju donosi objektno-orijentirana paradigma predstavlja bitnu, a može se reći i nužnu programsku

tehniku kod izgradnje dinamički iskoristivih programskih komponenti. Da bi vanjska programska komponenta, koja je u .NET razvojnom okruženju realizirana preko biblioteke razreda (engl. *class library*), bila iskoristiva unutar glavne aplikacije, nužan preduvjet je da glavna aplikacija može komunicirati s danom programskom komponentom. To se postiže izvođenjem (nasljeđivanjem) programske komponente (odnosno razreda) iz već postojećeg sučelja čija je definicija poznata i glavnoj aplikaciji i na čiju definiciju se glavna aplikacija oslanja prilikom upravljanja s učitanom programskom komponentom. Razina proširivosti optimizacijskog okruženja se stoga najbolje oslikava u skupu sučelja koje optimizacijsko okruženje definira za pojedine optimizacijske komponente.

### 3.2.4 Ugrađeni skup optimizacijskih komponenti

S obzirom da na području optimizacije postoji veliki broj standardnih problema koji se u raznim oblicima i formulacijama često javljaju u primjenama, dodatni zahtjev na općenito optimizacijsko okruženje je da ono dolazi sa ugrađenim skupom optimizacijskih komponenti za rješavanje takvih široko rasprostranjenih optimizacijskih problema.

Kako je za istraživača na području optimizacije najbolja situacija ona u kojoj postoji gotovo aplikacijsko rješenje za rješavanje zadanog optimizacijskog problema (opisani *off-the-shelf* scenarij) ugradnja široke baze osnovnih optimizacijskih komponenti značajno povećava mogućnost takvog scenarija. Pored toga, otvara se i mogućnost da se pri rješavanju kombiniranih problema iskoriste već postojeće zasebne optimizacijske programske komponente koje se jednostavnim slaganjem mogu primijeniti na optimizaciju zadanog kombiniranog problema.

### 3.2.5 Jednostavnost izgradnje novih komponenti

Nažalost, opisani scenarij se relativno rijetko događa u praksi, te su u većini slučajeva istraživači prisiljeni izgrađivati dodatne komponente potrebne za rješavanje zadanog problema. Složenost izgradnje i ugradnje novih programskih komponenti je pri tome vrlo bitan faktor. Ovdje je bitno naglasiti da se ne misli na složenost same programske implementacije za danu optimizacijsku komponentu (koliko je dana implementacija složena sa stajališta teorije algoritama i struktura podataka) već na složenost uspostavljanja razvojnog okruženja za izgradnju dane komponente i njenu ugradnju u optimizacijsko okruženje.

S obzirom na različite preferencije istraživača u odabiru razvojnog okruženja koje koriste u svakodnevnom radu, idealno optimizacijsko okruženje bi trebalo omogućiti izgradnju novih optimizacijskih komponenti u bilo kojem programskom jeziku. Takvo idealno optimizacijsko okruženje bi, npr. istovremeno podržavalo izgradnju i ugradnju algoritama za simpleks metodu u nekoj verziji Fortrana s oslanjanjem na njegove moćne biblioteke funkcija za manipulaciju s matricama, ugradnju genetičkih algoritama implementiranih u izvornom (engl. *native*) C++u te eventualnu izgradnju vizualizacijskih komponenti u Javi ili C#-u.

Nažalost, postizanje takve razine interoperabilnosti među komponentama izgrađenim u različitim programskim jezicima je tehnički vrlo složen problem, koji izrazito negativno utječe na performance izvođenja. Stoga je većina razvijenih optimizacijskih okruženja izgrađena korištenjem jednog programskog jezika u kojem se moraju graditi komponente kojima se proširuje dano okruženje. Međutim, pojava .NET programske platforme je značajno poboljšala situaciju s obzirom da zajednički sistem tipova (engl. *Common Type System – CTS*) i zajedničko izvršno okruženje (engl. *Common Language Runtime – CLR*) omogućavaju izgradnju programskih komponenti u različitim programskim jezicima (odnosno njihovim .NET varijantama) koje su unutar .NET okruženja posve interoperabilne. S obzirom na brzi rast broja podržanih programskih jezika u .NET okruženju (C++, C#, J#, Basic, Ruby, Python, Smalltalk) moguće se jako približiti idealnom scenariju u kojem istraživač može graditi svoje optimizacijske komponente u programskom jeziku po svom odabiru, a ne po odabiru nametnutom od strane korištenog optimizacijskog okruženja.

### 3.2.6 Efikasnost izračunavanja

Rješavanje složenih optimizacijskih problema koji se javljaju u praksi je u većini slučajeva izračunski vrlo težak zadatak. Stoga je kod izgradnje optimizacijskog okruženja bitno posvetiti dodatnu pažnju maksimiziranju efikasnosti izvršavanja programskog koda kako bi korisnik okruženja u što kraćem vremenu došao do traženih rezultata. Dinamička ugradnja programskih (optimizacijskih) komponenti u ESOP sučelje uvodi dodatni nivo indirekcije u arhitekturu programskog rješenja te se prilikom izgradnje tom aspektu mora posvetiti dodatna pažnja, što će biti opisano u poglavlju 5.6.

## 3.3 Opis postojećih okruženja

U zadnjih desetak godina su se pojavila mnogobrojna programska rješenja za optimizaciju. Razlog te proliferacije ponajviše leži u pojavi objektno-orijentirane paradigme koja je svoj zamah doživjela u 90-tim godinama prošlog stoljeća i koja zahvaljujući svojim karakteristikama predstavlja puno prirodnije okruženje za razvoj optimizacijskih okruženja. Većina tih programskih rješenja je izgrađena kao biblioteka razreda.

Jedna od prvih programskih biblioteka namijenjena razvoju programskih rješenja za optimizaciju pomoću genetičkih algoritama je GALib [ref] koja se pojavila još 1993. godine. Iako se broj izgrađenih rješenja vjerojatno mjeri u stotinama, može se navesti popis rješenja koja su najšire korištena i popraćena odgovarajućim znanstvenim člancima. Pored GALib, na taj popis svakako spadaju i JEO [Arenas2002a], OpenBEAGLE [Gagne2002], EOLib [Keijzer2001], DREAM [Arenas2002b], PISA [Bleuler2003], EASYLocal++, TEA [Emmerich2001], ParadiseEO [Cahon2004], ECJ [Luke2002] i HeuristicLab [Wagner2004].

Kod izgradnje većine tih rješenja su primjenjene programske tehnike na kojima će se bazirati i izgradnja ESOP optimizacijskog okruženja, odnosno njegovog dijela s ugrađenom funkcionalnošću za statičko (engl. *compile-time*) iskorištavanje izgrađenih

programskih komponenti. Stoga će se u ostatku ovog potpoglavlja detaljnije opisati GAlib biblioteka kao tipičan primjer rješenja izgrađenog kao biblioteka razreda i HeuristicLab optimizacijsko okruženje koje predstavlja nešto modernije rješenje, izgrađeno u .NET razvojnom okruženju s ugrađenim mogućnostima vizualizacije.

### 3.3.1 GAlib

GAlib je biblioteka razreda namijenjena rješavanju optimizacijskih problema pomoću genetičkih algoritama. Prilikom korištenja biblioteke, istraživač primarno radi s dva razreda: razredom *genoma* i razredom *genetičkog algoritma*. Genom predstavlja jedno rješenje problema i u sebi sadrži reprezentaciju ulaznih varijabli problema dok se preko razreda genetičkog algoritma definira način obavljanja optimizacije.

Unutar biblioteke je definiran osnovni razred GAGenome i nekoliko njegovih specijalizacija preko kojih su modelirane različite vrste reprezentacija (GABinaryString, GAArray, GAList) a korisnik biblioteke izvođenjem novog razreda iz razreda GAGenome može izgraditi i reprezentacije specifične za svoj problem. U biblioteku su također ugrađena i tri razreda koji predstavljaju osnovne vrste genetičkog algoritma: GASimpleGA, GASteadyStateGA i GAIncrementalGA. Važno je napomenuti da ti razredi imaju ugrađenu funkcionalnost za rukovanje populacijom i ostalim generičkim aspektima pripadnog genetičkog algoritma.

Jednostavan primjer korištenja je dan u sljedećem programskom odsječku u kojem se kao reprezentacija koristi polje bitova a sam genetički algoritam koristi standardno ugrađene operatore koji su dio ugradnje GASimpleGA razreda:

```
float Objective(GAGenome&);

main()
{
    GA1DBinaryStringGenome genome(length, Objective); // create a genome
    GASimpleGA ga(genome); // create the genetic algorithm
    ga.evolve(); // do the evolution
    cout << ga.statistics() << endl; // print out the results
}
float Objective(GAGenome&) {
    // definicija funkcije cilja
}
```

Podešavanje ponašanja genetičkog algoritma se obavlja postavljanjem određenih parametara.

```
ga.populationSize(popsize);
ga.nGenerations(ngen);
ga.pMutation(pmut);
ga.pCrossover(pcross);

GASigmaTruncationScaling sigmaTruncation;
ga.scaling(sigmaTruncation);
```

Definiranje funkcije cilja zahtjeva izgradnju funkcije koja kao parametar prima referencu na GAGenome objekt, obavlja prilagodbu (engl. *cast*) na konkretnu korištenu reprezentaciju i izračunava vrijednost funkcije cilja. Primjer jednostavne funkcije cilja koja kao vrijednost vraća broj bitova koji su postavljeni na 1 je dan u sljedećem programskom odsječku.

```
float Objective(GAGenome & g) {
    GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
    float score=0.0;
    for(int i=0; i<genome.length(); i++)
            score += genome.gene(i);
    return score;
}
```

Funkcija se može definirati kao statička članska funkcija u izvedenom razredu genoma ili se može ugraditi kao samostalna funkcija koja se kao pokazivač predaje postojećim razredima koji predstavljaju razrede *genoma*, kao što je to prikazano u prethodnom primjeru.

GAlib biblioteka poznaje tri vrste operatora: operator inicijalizacije kojim se incijaliziraju vrijednosti genoma, odnosno varijabli reprezentacija, i operatore križanja i mutacije koji predstavljaju standardne operatore genetičkih algoritama. Definiranje operatora koje će genetički algoritam koristiti u svom radu se obavlja predavanjem pokazivača na funkcije preko kojih su ti operatori realizirani. Postojeći razredi genetičkih algoritama dolaze s ugrađenim operatorima za osnovne vrste genoma, ali je korišteni operator vrlo jednostavno promijeniti predavanjem pokazivača na funkciju koja implementirati traženi operator. Iz sljedećeg programskog odsječka se vidi način redefiniranja operatora za inicijalizaciju i obavljanje križanja nad genomom koji reprezentira polje bitova. Ovakva ugradnja podrazumijeva postojanje funkcija MyInitializer i MyCrossover u kojima je implementirana funkcionalnost tih operatora.

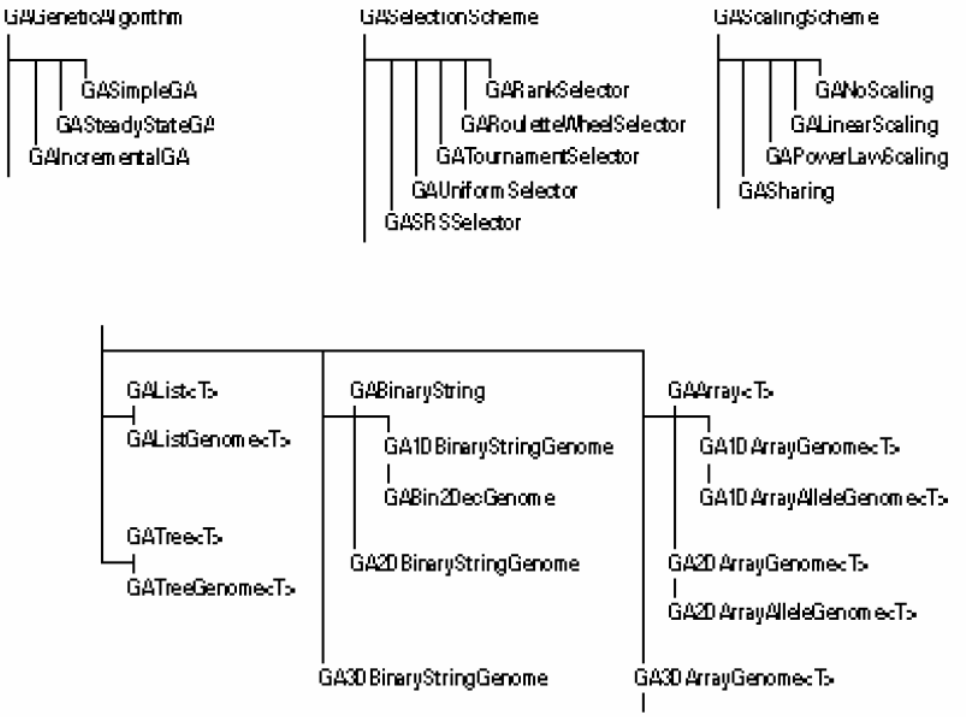
```
GA1DBinaryStringGenome genome(20);
genome.initializer(MyInitializer);
genome.crossover(MyCrossover);
```

Prilikom izgradnje vlastitih genoma, najefikasnije je direktno u razredu podesiti operatore koji će se koristiti kod rada genetičkog algoritma s takvom vrstom genoma.

```
class MyGenome : public GAGenome {
public:
    static void RandomInitializer(GAGenome&);
    static int JuggleCrossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
    static int KillerMutate(GAGenome&, float);
    static float ElementComparator(const GAGenome&, const GAGenome&);
    static float ThresholdObjective(GAGenome&);
public:
    MyGenome() {
            initializer(RandomInitializer);
            crossover(JuggleCrossover);
            mutator(KillerMutate);
```

```
            comparator(ElementComparator);
            evaluator(ThresholdObjective);
        }
        // remainder of class definition here
};
```

Pored operatora križanja i mutacije, za genetički algoritam je bitan i način pretvaranja vrijednosti funkcije cilja u dobrotu rješenja te način odabira jedinki za reprodukciju (operator selekcije). GAlib biblioteka dolazi s predefiniranim ugradnjama funkcija kojima su realizirani odgovarajući načini pridruživanja dobrote i provođenja selekcije, a njihov odabir se također svodi na prenošenje pokazivača na odgovarajuću funkciju objektu genetičkog algoritma. Pored ovih razreda, u GAlib su ugrađeni i razredi za određivanje trajanja optimizacije te četiri razreda koji implementiraju različite varijante generatora slučajnih brojeva. Kompletna hijerarhija razreda ugrađenih u GAlib biblioteku je dana na slici 3.1.




**Slika 3.1 Hijerarhija razreda GAlib biblioteke.**

GAlib biblioteka predstavlja široko korišteno, iako ponešto zastarjelo programsko rješenje za provođenje optimizacije genetičkim algoritmima. Osnovna prednost ove biblioteke je njema široka dostupnost, postojanje dobre dokumentacije i kvaliteta same implementacije koja je provjerena kroz deset godina postojanja i mnogobrojne primjene u praksi.

Međutim, ova biblioteka pati i od nekih problema. Jedna zamjerka proizlazi iz nepostojanja bilo kakvih alata za vizualizaciju dobivenih rješenja, a čije postojanje značajno olakšava rad istraživaču prilikom rješavanja konkretnog optimizacijskog problema. Pored toga, sama izgradnja u klasičnom C++ programskom jeziku ima

određene nedostatke i sa stajališta arhitekture pošto je veliki dio funkcionalnost ugrađen preko globalnih deklaracija koje su razasute kroz različite datoteke zaglavlja. Iako je cijelo programsko rješenje dobro strukturirano, bolje razumijevanje zahtjeva napredno znanje C++ programskog jezika. Uz to, korištenje koncepta pokazivača na funkciju kao osnovnog načina za proširivanje biblioteke s novom funkcionalnošću također nije najsretnije rješenje. Ipak, mora se priznati da je usprkos navedenim nedostacima, GAlib biblioteka kvalitetno programsko rješenje za provođenje optimizacije genetičkim algoritmima, uz uvjet da je korisnik spreman na nešto dublje ulaženje u tehničke detalje C++ programskog jezika.

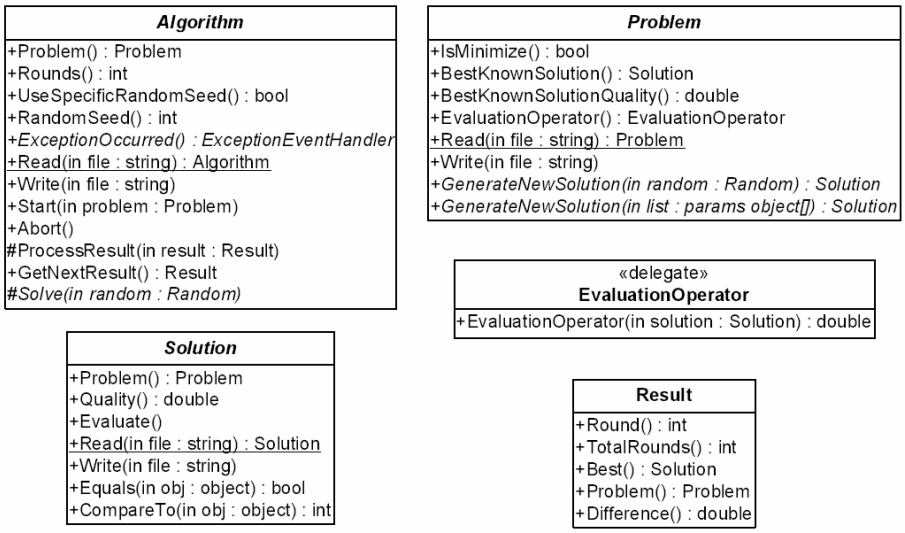
### 3.3.2 HeuristicLab

HeuristicLab, za razliku od GAlib biblioteke, predstavlja moderno optimizacijsko okruženje izgrađeno korištenjem .NET razvojne okoline i C# programskog jezika. HeuristicLab okruženje je koncipirano kao općenito optimizacijsko okruženje namijenjeno ugradnji različitih optimizacijskih postupaka. Za razliku od biblioteke GAlib, i većine ostalih programskih rješenja za optimizaciju navedenih na početku ovog potpoglavlja, kod izgradnje ovog rješenja je od početka velika pažnja posvećena vizualizacijskim aspektima provođenja optimizacije. Nepostojanje vizualnih sučelja kroz koja se može provoditi proces optimizacije je standardni nedostatak većine programskih rješenja za optimizaciju koja su izgrađena kao biblioteke.

Mogućnost ugradnje različitih optimizacijskih postupaka je kod HeuristicLab okruženja postignuta modeliranjem optimizacije pomoću heurističkih algoritama na visokom nivou apstrakcije. Citirajući same autore:

> „Svaki *algoritam* na specifičan način iterativno modificira jedno ili više *rješenja* danog *problema* s ciljem povećanja njihove kvalitete i generira *rezultate* preko kojih obavještavaja druge o napretku optimizacije“

Četiri navedena koncepta: algoritam, problem, rješenje i rezultat su prisutna u svakom procesu optimizacije pomoću heurističkih postupaka i oni su u HeuristicLab okruženju modelirane preko četiri apstraktna razreda: Algorithm, Problem, Solution i Result. Pored toga je kao osnovni koncept dodan i EvaluationOperator delegat čija je osnovna namjena evaluiranje kvalitete rješenja. UML dijagram iz kojeg su vidljive deklaracije tih razreda je dan na slici 3.2.




**Slika 3.2 Dijagram osnovnih razreda HeuristicLab optimizacijskog okruženja**

Apstraktni razred Algorithm realizira općeniti koncept optimizacijskog postupka i ima ugrađenu funkcionalnost zajedničku za sve optimizacijske algoritme. Ta funkcionalnost je vezana uz mogućnost definiranja korištenog generatora slučajnih brojeva, pokretanje procesa optimizacije u vlastitoj dretvi, te rukovanje rezultatima optimizacije. Ovaj razred je namijenjen daljenjem nasljeđivanju od strane razreda koji će implementirati konkretni optimizacijski postupak.

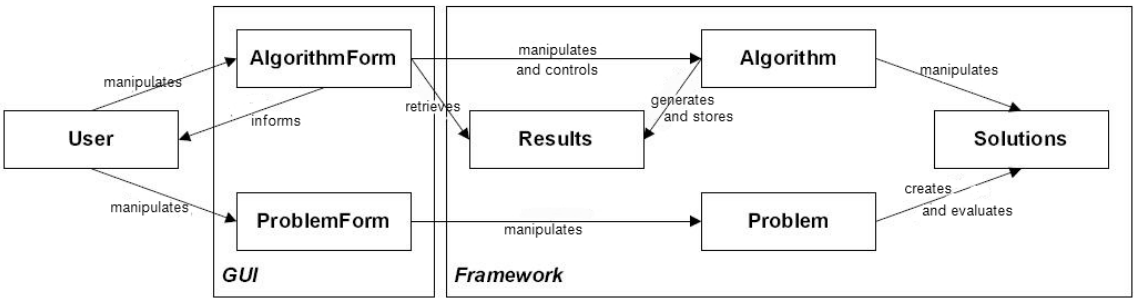
Razred Problem je u implementacijskom pogledu sličan razredu Algorithm, s tom razlikom da je njime predstavljen koncept optimizacijskog problema. U njega je ugrađena funkcionalnost za slučajno generiranje rješenja i dohvaćanje najboljih poznatih rješenja za dani problem. Vrlo važan dio razreda Problem čini referenca na EvaluationOperator delegat preko kojega se izračunava kvaliteta rješenja (u terminologiji genetičkih algoritma se radi o dobroti – fitnessu danog rješenja).

Razred Solution je također apstraktni razred koji predstavlja rješenje optimizacijskog problema. Prilagođavanje za konkretnu reprezentaciju se provodi izvođenjem novog razreda iz razreda Solution. Razred Solution ima referencu na Problem objekt a kvaliteta rješenja se izračunava pozivom funkcije Evaluate() koja za proračun kvalitete rješenja koristi EvaluatonOperator iz objekta problema. CompareTo() funkcija služi za uspoređivanje dva rješenja.

Razred Result sadrži konačni rezultat optimizacije preko reference Best. Pored toga sadrži i podatke o trenutnoj iteraciji (Rounds) i ukupnom broju iteracija koji se treba provesti (TotalRounds). Difference predstavlja razliku u kvaliteti između trenutnog najboljeg rješenja i najboljeg uopće poznatog rješenja, koje se dobije preko reference na Problem objekt koji sadrži taj podatak ukoliko je on poznat.

Izvođenjem konkretnih razreda iz navedenih apstraktnih razreda te njihovim međusobnim povezivanjem se mogu izgraditi vrlo različiti optimizacijski postupci. Međutim, HeuristicLab ide i dalje od toga, ugrađujući infrastrukturu potrebnu za upravljanje procesom optimizacije kroz vizualno sučelje. To je postignuto preko

koncepta forme koja predstavlja vizualno sučelje prema konkretnim objektima koji predstavljaju probleme i algoritme. Definirana su dva apstraktna razreda, AlgorithmForm i ProblemForm, koji imaju ugrađenu osnovnu funkcionalnost i iz kojih će graditelji konkretnih objekata algoritma i problema izvesti nove forme za upravljanje s novoizgrađenim optimizacijskim komponentama. Forme imaju reference na odgovarajuće objekte problema i algoritma, a povratna komunikacija se izvodi preko Results razreda. Algoritam u Results objekt stavlja rezultate optimizacije, taj objekt se stavlja u FIFO red iz kojeg ga dohvaća AlgorithmForm-a te vizualizira na ekranu. Na ovaj način je postignuto razdvajanje prezentacijskog sloja od sloja u kojem su konkretni objekti koji se koriste za provođenje optimizacije. Međudjelovanje objekata koji sudjeluju u optimizaciji je prikazano na slici 3.3.

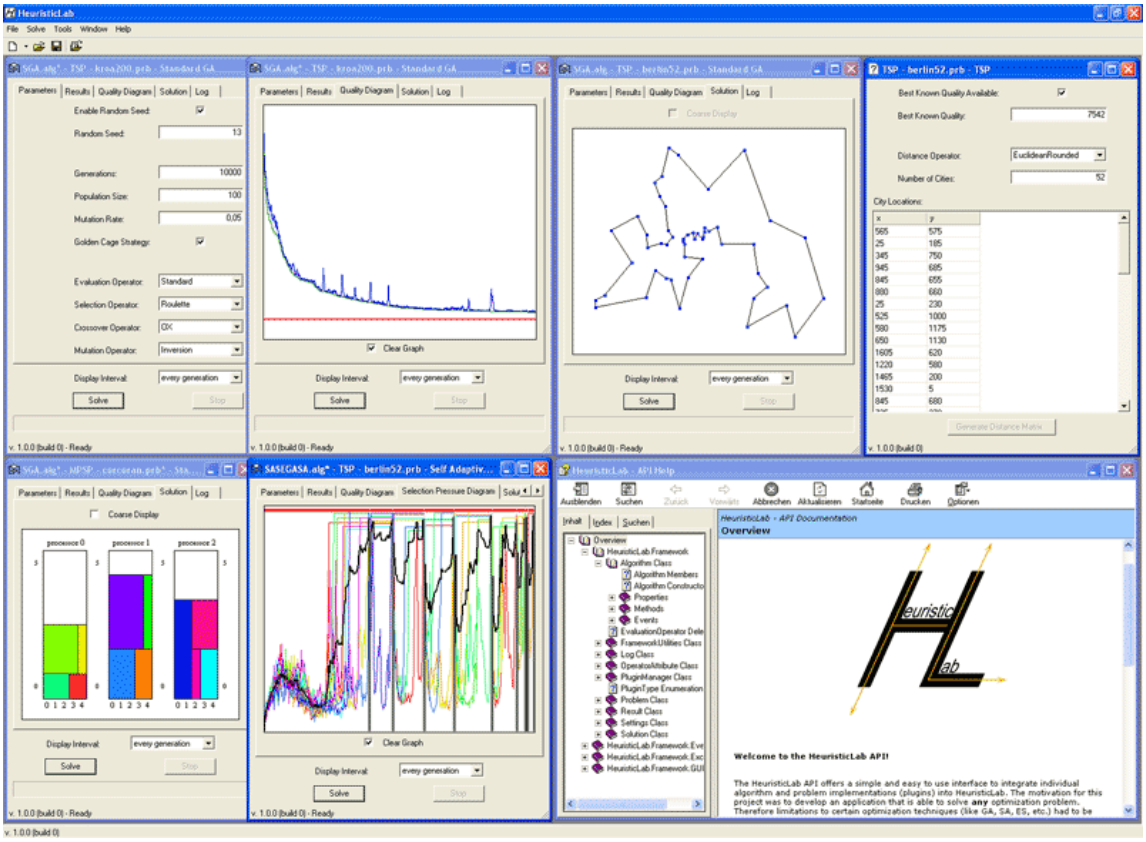



**Slika 3.3 Međudjelovanje osnovnih razreda u HeuristicLab okruženju**

Specifičnost HeuristicLab okruženja je način na koji je u njemu realiziran koncept operatora koji optimizacijski algoritmi koriste za pretraživanje prostora rješenja. Operatori su specifični za problem, odnosno za reprezentaciju problema (skup ulaznih varijabli nad kojima je problem definiran). Kod izgradnje heurističkih algoritama jedan od osnovnih problema je iznalaženje načina za razdvajanje koncepta operatora, kojemu je poznata konkretna reprezentacija što njegovu implementaciju čini ovisnom o vrsti problema koji se rješava, i koncepta algoritma koji samo koristi operator u procesu optimizacije. U HeuristicLab okruženju je koncept operatora modeliran korištenjem delegata. Delegat, u terminologiji .NET okruženja, je tip koji u biti predstavlja metodu (funkciju). Detaljan opis koncepta delegata u .NET okruženju se može naći u [Richter2002] a pojednostavljeno se može reći da delegat predstavlja enkapsulaciju funkcije kao zasebnog objekta. Odnosno, delegat je ekvivalent pokazivača na funkciju u standardnoj proceduralnoj paradigmi. U moderne objektno-orijentirane jezike je pojam delegata uveden kako bi se izbjegli nedostaci korištenja pokazivača na funkcije.

S tehničkog stajališta, delegat definira tipove parametara koji se trebaju predati funkciji i tip povratnog parametra. Instanci delegata se može pridružiti bilo kakva funkcija koja zadovoljava definiranu deklaraciju i s delegatom se može operarati kao s bilo kojom drugom varijablom. To, primjerice, znači da će razred koji će predstavljati ugradnju genetičkog algoritma imati dvije članske varijable koje će predstavljati delegate i koje će se inicijalizirati s odgovarajućim implementacijama operatora križanja i mutacije.

Međutim, tu se javlja problem pronalaženja implementacija odgovarajućih funkcija, preko kojih su realizirani pojedini operatori, u različitim bibliotekama razreda (dll). U HeuristicLab okruženju je taj problem riješen korištenjem atributa. Atributi predstavljaju novinu uvedenu u .NET razvojnom okruženju i pomoću njih pisac izvornog koda razreda može u metapodatke biblioteke u kojoj se nalazi razred ugraditi neke svoje specifične podatke. Na primjer, svaka metoda u razredu koja predstavlja operator križanja ima pridružen atribut „CROSSOVER“. Korištenjem funkcionalnosti ugrađene u .NET razvojno okruženje se tijekom izvršavanja programa na osnovu pridružene vrijednosti atributa ta optimizacijska komponenta može identificirati kao operator križanja i u skladu s time iskoristiti. Postavljanjem ovakve arhitekture je omogućeno iskorištavanje operatora ugrađenih u različite biblioteke razreda (u terminologiji HeuristicLab okruženja se takva biblioteka razreda naziva *plug-in*).



**Slika 3.4 Slika ekranskog zaslona HeuristicLab optimizacijskog okruženja**

HeuristicLab dolazi s velikim skupom ugrađenih *plug-in*-ova za rješavanje različitih optimizacijskih problema: GA (*Generic Genetic Algorithm*), SSGA (*Steady-State Genetic Algorithm*), IslandGA (*Coarse Grained Parallel Genetic Algorithm*), GP (*Genetic Programming*), ES (*Evolution Strategy*), SA (*Simulated Annealing*), PSO (*Particle Swarm Optimization*), ACO (*Ant Colony Optimization*), TS (*Tabu Search*), TSP (*Traveling Salesman Problem*), JSSP (*Job Shop Scheduling Problem*), MPSP (*Multiprocessor Scheduling Problem*). Prikaz zaslona HeuristicLab okruženja u akciji je dan na slici 3.4.

HeuristicLab okruženje predstavlja značajan napredak u odnosu na optimizacijska okruženja koja su izgrađena kao biblioteke razreda. U osnovama svoje arhitekture HeuristicLab je vrlo sličan ESOP optimizacijskom okruženju. Korištenjem naprednih mogućnosti .NET razvojnog okruženja je definirana vrlo fleksibilna arhitektura koja podržava ugradnju različitih problema i optimizacijskih postupaka.

Međutim, upravo u korištenju tih naprednih mogućnosti leži možda i najveći nedostatak. Naime, koncepti delegata i atributa su napredni koncepti OO paradigme i potpuno razumijevanje njihovog rada zahtjeva dubinsko ulaženje u arhitekturu .NET okruženja. Iako se ugradnja optimizacijskih komponenti uz korištenje delegata i atributa može svesti na metodu „kuharice“, znači na jednostavno ugrađivanje bez potpunog razumijevanja, korištenje takvog načina proširivanja s novim optimizacijskim komponentama može odbiti istraživača od primjene ovog optimizacijskog okruženja.

Pored toga, direktno povezivanje optimizacijskih komponenti s odgovarajućim vizualizacijskim komponentama (odnosno formama) u određenim situacijama također može predstavljati nedostatak, poglavito u situaciji kad izgrađena programska komponenta za provođenje optimizacije predstavlja samo jedan dio šireg informacijskog sustava.

# 4. Konceptualni model domene optimizacije

Osnovni cilj prethodna tri poglavlja je bio pružiti uvid u područje optimizacije, i to sa stajališta izgradnje općenitog programskog okruženja za provođenje optimizacije. Klasifikacija optimizacijskih problema i postupaka koja je opisana u prva dva poglavlja, zajedno s opisom traženih karakteristika optimizacijskog okruženja predstavlja osnovu nad kojom će se (iz)graditi ESOP optimizacijsko okruženje. U jeziku metodologija za razvoj softvera, prva dva poglavlja predstavljaju analizu domene problema (engl. *domain analysis*) dok je u trećem dana specifikacija zahtjeva za dani sistem (engl. *requirements specification*).

Sljedeći korak je sama izgradnja ESOP optimizacijskog okruženja koja je provedena u nekoliko iteracija u skladu s pravilima dizajna zasnovanog na analizi domene (engl. *domain driven design*). U ovom poglavlju će se opisati konceptualni model domene programske podrške za optimizaciju, a opis same izgradnje ESOP okruženja je tema sljedećeg poglavlja. Kako bi opis izgradnje ESOP optimizacijskog okruženja bio što jasniji, najprije će se dati sažeti prikaz primijenjene metodologije razvoja programske podrške.

## 4.1 Metodologije razvoja softvera

Kod izgradnje bilo kakvog informacijskog sustava od primarne je važnosti odabir korištenog procesa razvoja programske podrške (engl. *software development process*) kojim je definirana metodologija izgradnje informacijskog sustava. Metodologiju razvoja softvera možemo definirati kao „kolekciju procedura, tehnika, alata i dokumentacijskih pomagala, potkrijepljenih filozofijom, koji potpomažu izgradnju informacijskog sustava“ [Avison1995]. Osnovni ciljevi definiranja metodologije su [Fertalj2006]:

- omogućiti sustavni postupak razvoja kojem će se moći pratiti napredak
- uspostaviti komunikaciju između sudionika uključenih u izgradnju IS (poslovodstvo, korisnici, analitičari, programeri …)
- osigurati skup tehnika koji će omogućiti da se zadaci izvršavaju na standardne i provjerene načine
- osigurati učinkovit nadzor sa ciljem uočavanja pogrešaka u ranim fazama
- omogućiti elastične promjene poslovanja i tehnologije (npr. odvajanjem analize i oblikovanja)
- uokviriti razvojnu strategiju kojom će se ukloniti *ad hoc* rješavanje problema
- odrediti ili ukazati kada i u kojoj mjeri je potrebno uključivanje korisnika, te poticati i omogućiti uključivanje korisnika kada se za to ukaže potreba
- osigurati da se dovoljno pažnje posveti analizi poslovanja, čime će se osigurati izrada sustava koji odgovara poslovanju i zahtjevima korisnika

Postojeće metodologije pristupaju ostvarivanju navedenih ciljeva na različite načine, a samo područje računarske znanosti koje se njima bavi prolazi kroz česte promjene, uzrokovane ponajviše sve većom složenošću IT sustava koji se razvijaju, ali i napretkom na području informacijske tehnologije (pojava proceduralne, strukturne i objektno-orijentirane paradigme, pojava Interneta i komunikacijskih mogućnosti).

Iako među postojećim metodologijama postoje značajne razlike u pristupu izgradnji informacijskog sustava, mogu se definirati i njihove zajedničke karakteristike koje proizlaze iz samog problema izgradnje informacijskog sustava. Osnovna pitanja kojima se bavi područje metodologije razvoja softvera su: „što treba izgraditi ?“ i „kako to izgraditi ?“. Iz ova dva pitanja direktno proizlaze i dvije osnovne faze izgradnje informacijskog sustava koje su u različitim oblicima prisutne u svim postojećim metodologijama. Prvu fazu čini skup aktivnosti povezan s provođenjem analize problema, dok drugu fazu čini skup aktivnosti vezanih uz samu izgradnju (implementaciju) informacijskog sustava. Skup aktivnosti, njihov redoslijed i međuovisnosti koje pojedine metodologije propisuju za svaku od ovih faza su čimbenici po kojima se postojeće metodologije razlikuju.

## 4.1.1 Osnovne aktivnosti u procesu razvoja softvera

Informacijski sustavi se značajno razlikuju po složenosti izgradnje. Jednostavni „sustavi“ se sastoje od jedne aplikacije koju jedan programer izgradi u nekoliko dana ili tjedana. Složeni informacijski sustavi se često sastoje od nekoliko (desetaka) milijuna linija programskog kôda na čijoj su izradi radili cijeli timovi programera kroz nekoliko godina. Jasno je da tako različita složenost izgradnje povlači za sobom i različite pristupe izgradnji. Međutim, prilikom izgradnje bilo kakvog informacijskog sustava, postoji određeni skup aktivnosti koje se moraju provesti bez obzira na njegovu složenost. Te aktivnosti se ugrubo mogu grupirati u tri različite faze razvoja informacijskog sustava: fazu analize, fazu oblikovanja (dizajna), i fazu izgradnje (implementacije). Svaka od navedenih faza ima svoju ulogu u procesu razvoja informacijskog sustava, a ovisno o složenosti sustava koji se razvija varira i relativni značaj svake od njih. Detaljniji prikaz procesa razvoja informacijskih sustava se može naći u [Maciaszek2001], a izvrstan prikaz moderne objektno-orijentirane metodologije bazirane na UML-u (*Unified Modeling Language*) i obrascima oblikovanja (engl. *design patterns*) se može naći u [Larman2005].

### 4.1.1.1 Faza analize

U fazi analize fokus je na zahtjevima (engl. *requirements*) koje sistem mora zadovoljavati. Osnovni cilj analize je određivanje skupa zahtjeva postavljenih na sistem (engl. *requirements determination phase*) te njihova precizna specifikacija (engl. *requirements specification phase*) [Maciaszek2001]. U ovoj fazi se definiraju funkcionalnosti sustava i modeli podataka, a razmatraju se i prikupljaju i nefunkcionalni zahtjevi te ograničenja koja sustav mora zadovoljavati.

#### 4.1.1.2 Faza oblikovanja

Fazu oblikovanja se obično dijeli u dvije veće podfaze: arhitekturno i detaljno oblikovanje. Arhitekturno oblikovanje podrazumijeva opis sistema na višoj razini apstrakcije, odnosno razlaganje sustava na skup modula u kojima su grupirane određene skupine funkcionalnosti sistema, dok se u fazi detaljnog oblikovanja razrađuju i definiraju unutrašnji detalji svakog modula. U fazi oblikovanja se razmatraju različiti utjecaji na skalabilnost razvijenog sustava, a posebna pažnja se posvećuje izgradnji sustava koji će biti što jednostavniji za razumijevanje i kasnije održavanje (engl. *maintainability*).

#### 4.1.1.3 Faza izgradnje

U fazi izgradnje (implementacije) se obavlja kodiranje danog sustava u skladu sa specifikacijama izrađenim u ranijim fazama. Najčešće se koriste iterativni i inkrementalni procesi, a provodi se i testiranje ugrađenog programskog kôda. Vrlo često se pokazuje potreba za povratnom vezom (engl. *round-trip engineering*) između faza oblikovanja i izgradnje kako bi se unutar sustava što bolje obuhvatili zahtjevi korisnika i saznanja do kojih se dolazi u kasnijim fazama izgradnje sustava.

### 4.1.2 Modeli razvoja softvera

U prethodnom odjeljku su opisane pojedine faze razvoja informacijskog sustava. Međutim, nije rečeno ništa o redoslijedu njihovog izvođenja, odnosno o interakcijama među različitim fazama. Određivanje redoslijeda aktivnosti kod razvoja informacijskog sustava i njihovih međuovisnosti čini važnu komponentu svake metodologije razvoja softvera. Iako svaka od mnogobrojnih postojećih metodologija u tom pogledu ima svojih specifičnosti, postoje dva osnovna modela koji su široko korišteni i dobro ilustriraju dva fundamentalno različita pristupa izgradnji informacijskih sustava. To su vodopadni (engl. *waterfall*) i iterativni model razvoja informacijskih sustava.

Osnovna karakteristika vodopadnog modela je slijedno provođenje faza analize, oblikovanja, izgradnje i testiranja informacijskog sustava koji se izgrađuje. Odnosno, svaka od faza se provodi u potpunosti, nakon čega se prelazi u sljedeću fazu. Iako je sa stajališta upravljanja procesom razvoja programske podrške to vrlo zahvalan model, u novije vrijeme se polako napušta te se prednost daje iterativnom modelu razvoja. Osnovni nedostatak vodopadnog modela razvoja leži u mogućnosti pojavljivanja i uočavanja pogrešaka u kasnijim fazama razvoja, kada je provođenje promjene u sustavu vrlo skupo, odnosno u nemogućnosti reakcije na promijenjene zahtjeve korisnika.

Stoga su u posljednje vrijeme na popularnosti dobile iterativne metode razvoja, od kojih su primjeri *Rational Unified Process* [Jacobson1999], agilne metodologije [Martin2003] i tehnike ekstremnog programiranja [Beck1999]. Osnovna karakteristika iterativnog modela razvoja jest da se informacijski sustav razvija kroz niz iteracija, s time da se u svakoj iteraciji prolazi kroz sve opisane faze razvoja (znači analizu, oblikovanje, izgradnju i testiranje). Navedene metode se donekle razlikuju u pojedinim detaljima, ali im je zajednička centralna uloga testiranja u procesu razvoja programske podrške i mogućnost brze reakcije na promijenjene zahtjeve korisnika. U skladu s time

kod navedenih metodologija značajnu ulogu imaju tehnike prilagođavanja (engl. *refactoring*) programskog kôda kako bi arhitektura i oblikovanje sustava uvijek bili u skladu sa zahtjevima krajnjih korisnika [Fowler1999].

### 4.1.3 Dizajn pokretan domenom

Pojava objektno-orijentirane paradigme je iz osnova promijenila proces izgradnje programske podrške. Mnoge postojeće metodologija su utemeljene na proceduralnoj paradigmi koja podatke i funkcije koje nad njima operiraju promatra kao dva odvojena (iako inherentno povezana) aspekta izgradnje informacijskog sustava. Tome su i prilagođene tehnike modeliranja koje te metodologije koriste: dijagrami toka podataka (engl. *Data Flow Diagrams*), logički modeli podataka (engl. *Logical Data Model*) i drugi. Objektno-orijentirana paradigma se, s druge strane, bazira na konceptu razreda / objekta koji sjedinjuje ta dva aspekta. Osnovna prednost takvog pristupa je mogućnost programskog modeliranja elemenata danog informacijskog sustava direktno u terminima domene problema kojoj je taj informacijski sustav namijenjen. Razredi / objekti imaju definirano stanje, određeno vrijednostima članskih varijabli razreda, i ponašanje koje je definirano javnim sučeljem razreda koje čine svi javni članovi razreda. Direktno preslikavanje koncepata iz domene problema u odgovarajući skup razreda / objekata omogućava bolje razumijevanje danog informacijskog sustava i olakšava komunikaciju između sistem analitičara koji poznaje domenu problema i graditelja informacijskog sustava (engl. *developer*) koji tu domenu problema mora softverski modelirati i izgraditi.

Međutim, korištenje objektno-orijentirane paradigme samo djelomično olakšava razvoj složenih informacijskih sustava. Razlog tome je činjenica da složenost izgradnje informacijskog sustava samo djelomično proizlazi iz tehničke složenosti njegove implementacije. Najznačajniju komponentu složenosti izgradnje informacijskog sustava čini složenost same domene problema kojoj je informacijski sustav namijenjen. Naravno, informacijski sustavi kod izgradnje zahtijevaju rješavanje često i vrlo složenih tehničkih problema. Omogućavanje klijent-server načina rada kod distribuiranih aplikacija, izgradnja skalabilnih web-aplikacija s podrškom za broj korisnika koji se mjeri u tisućama ili milijunima, ugradnja višenitnog izvršavanja s ciljem postizanja što boljeg odziva informacijskog sustava samo su neki od mogućih tehničkih izazova s kojima se graditelji informacijskih sustava svakodnevno suočavaju. Međutim, to su ipak samo tehnički problemi čije je rješavanje moguće standardnim inženjerskim tehnikama koje se najčešće sastoje u pronalaženju već gotovih sličnih rješenja i njihovom prilagođavanju danoj situaciji.

Za razliku od tih tehničkih aspekata izgradnje sustava koji se uvelike mogu klasificirati i koji su tema mnogobrojnih što specijalističkih tečajeva a što standardnih predmeta na računarski orijentiranim fakultetima, problemi povezani sa složenošću domene za koju se gradi informacijski sustav ne podliježu tako jednostavnim tehnikama rješavanja.

Jedna od najnovijih filozofija razvoja informacijskih sustava je i dizajn pokretan domenom [Evans2004] (korištenjem termina filozofija a ne metodologija se želi naglasiti činjenica da tehnike koje obuhvaća nisu u tolikoj mjeri formalizirane). Osnovna

karakteristika ovog načina razvoja informacijskih sustava jest da se primarni naglasak stavlja na izradu modela domene koji predstavlja „rigorozno organiziranu i selektivnu apstrakciju znanja“ [Evans2004] o toj domeni. Bitno je naglasiti da izrađeni model domene ne mora nužno biti u obliku dijagrama niti mora predstavljati doslovnu repliku stvarnog svijeta koji se njime modelira. Njegova uloga kod razvoja korištenjem dizajna pokretanog domenom je višestruka:

1. Model predstavlja osnovu za izradu dizajna informacijskog sustava

Između modela domene i konačne implementacije sustava postoji intimna veza koja model čini relevantnim i osigurava da analiza koja je pratila njegovu izradu bude primjenjiva i na konačni produkt razvoja, program koji se izvršava. Jedan od najvećih problema mnogih metodologija razvoja softvera jest nepovezanost rezultata analize i konačne implementacije. Uzrok toga je jednostavan: do dubinskog shvaćanja domene problema se najčešće dolazi tek u fazi izgradnje informacijskog sustava. Za potvrdu ove jednostavne činjenice je dovoljno pogledati mnogobrojne dokumente koji su generirani u fazama analize kod razvoja nekog informacijskog sustava i koji s krajnjom implementacijom nemaju nikakve veze. U najboljem slučaju završe skupljajući prašinu na nekoj polici dok u najgorem slučaju nekoga tko ih kasnije proučava vode na pogrešan put u razumijevanju izgrađenog sustava.

Dizajn pokretan domenom razrješava taj problem povezivanjem (engl. *binding*) modela i implementacije koji se tijekom razvoja informacijskog sustava paralelno razvijaju sa stjecanjem sve dubljeg uvida u domenu problema. Ili, citirajući Erica Evansa, „dizajn pokretan domenom zahtijeva izradu modela koje ne samo da pomaže u ranoj analizi problema, već predstavlja samu osnovu za dizajn i implementaciju [informacijskog sustava]“.

2. Model čini kostur jezika komunikacije za sve članove razvojnog tima

Zbog vezivanja modela i implementacija, graditelji sustava mogu razgovarati u jeziku domene problema i komunicirati direktno sa ekspertima iz te domene bez prevođenja pojmova. A kako je taj jezik baziran na samom modelu, članovi tima mogu koristiti ljudima urođene lingvističke sposobnosti za daljnje poboljšavanje i razrađivanje samog modela.

3. Model predstavlja pročišćeno znanje o domeni problema

Model predstavlja dogovoreni način za strukturiranje znanja o domeni i razlučivanje elemenata koji su od primarnog interesa u toj domeni. Model određuje način razmišljanja o domeni tako što definira skup pojmova, razbija ih na pojedine koncepte i ustanovljava njihovu povezanost. Zajednički jezik omogućava graditeljima i ekspertima domene da efikasno surađuju u svođenju informacija o domeni u prikladan oblik a veza ustanovljena između modela i implementacije omogućava da se iskustva skupljena u ranim fazama izgradnje iskoriste kao ulazi za daljnje poboljšavanje samog modela.

## 4.2 Model domene programske podrške za optimizaciju

U ovom potpoglavlju će se opisati izgrađen konceptualni model domene programske podrške za optimizaciju koji, u skladu sa navedenim principima dizajna pokretanog domenom, predstavlja osnovu za izgradnju ESOP optimizacijskog okruženja. Ovdje dani opis predstavlja konačni rezultat nekoliko iteracija razvoja ESOP okruženja kroz koje su opisani koncepti i njihove interakcije razrađivani i poboljšavani s ciljem što boljeg zadovoljavanja u uvodu navedenih zahtjeva na općenito optimizacijsko okruženje. Prije pristupanja opisu samog konceptualnog modela, potrebno je detaljnije opisati ulogu koju će taj model imati u izgradnji ESOP optimizacijskog okruženja.

### 4.2.1 Značaj konceptualnog modela za izgradnju ESOP-a

Jedna od osnovnih postavki dizajna pokretanog domenom jest ustanovljavanje direktne veze između modela domene i konkretne izgradnje informacijskog sustava. Većina postojećih (uglavnom starijih) metodologija razdvaja faze analize, dizajna i implementacije informacijskog sustava u zasebne aktivnosti u procesu izgradnje. Kako je već opisano, takav pristup u velikom broju slučajeva vodi u situacije u kojoj artefakti iz već provedenih faza izgradnje (specifikacije, dokumenti sistemske analize, UML dijagrami dizajna i arhitekture) u odnosu na zahtjeve trenutne faze izgradnje sustava postaju „mrtvo slovo na papiru“. Odnosno, zbog sve boljeg uvida u domenu problema za koju se gradi informacijski sustav rezultati prethodnih faza postaju irelevantni te graditeljima sustava ne preostaje ništa drugo nego da ih u svom daljnjem radu zanemare.

Jedan od načina rješavanja te situacije jest primjena već opisanih modernih metodologija iterativnog razvoja (agilne metodologije, tehnike ekstremnog programiranja) u kojima se klasični vodopadni model razvoja razbija na nekoliko iteracija. Unutar svake iteracije se zatim provodi cijeli ciklus razvoja koji obuhvaća faze analize, dizajna i implementacije, a problem mijenjanja zahtjeva zbog sve boljeg uvida u sustav se rješava intenzivnom primjenom tehnika za prilagodbu dizajna sustava (engl. *refactoring*).

Dizajn pokretan domenom kao filozofija razvoja također spada u iterativne metodologije. Međutim, za razliku od postojećih metodologija dizajn pokretan domenom u određenoj mjeri sjedinjava sve opisane faze procesa razvoja softvera u jedinstvenu fazu. To je omogućeno centralnom ulogom modela domene koji predstavlja okvir unutar kojega se obavljaju i faza analize i faza dizajna i faza implementacije. Zahvaljujući jedinstvenom jeziku koji je baziran upravo na (stalno evoluirajućem) modelu domene značajno je olakšana komunikacija između sistem analitičara, dizajnera i programera. Time je omogućena brza povratna veza između različitih aktivnosti jer se koncepti i njihove međusobne interakcije koje su sistem analitičari definirali u domeni direktno programski realiziraju preslikavanjem identificiranih koncepata u elemente objektno-orijentirane paradigme (razredi, objekti, sučelja), uz korištenje pripadnih programskih tehnika (apstrakcija, učahurivanje, polimorfizam / nasljeđivanje).

Stoga tijekom razvoja ne postoje zasebni dokumenti u kojima bi se opisivala analiza problema i definirani dizajn programskog rješenja, već se cjelokupni razvoj informacijskog sustava odvija u kontekstu izgrađenog modela domene. Svaki od sudionika u procesu razvoja ima svoj pogled na taj model, u skladu sa svojom specijalnošću. Sistem analitičari su primarno fokusirani na pronalaženje relevantnih koncepata, definiranje njihovih (apstraktnih) karakteristika i istraživanje načina njihove interakcije. Sistem dizajneri mapiraju identificirane koncepte u objektni model i pri tome stječu dodatni uvid u sam problem. Postojanje jedinstvenog modela domene im omogućuje da u direktnoj interakciji sa sistem analitičarima dodatno razrađuju izgrađeni model. Programeri u konačnici grade programske komponente definirane u objektnom modelu i tijekom same izgradnje u suradnji i sa sistem analitičarima i s dizajnerima dodatno razrađuju model, u skladu s tehničkim detaljima koji se pojavljuju tek u fazi kodiranja. Bitna je karakteristika dizajna pokretanog domenom da se sve navedene aktivnosti odvijaju paralelno čime se izbjegava klasični problem kasnih promjena u zahtjevima koje, pogotovo u vodopadnom modelu razvoja, iz temelja ruše karakteristike već izgrađenog dijela sustava.

U kontekstu dizajna pokretanog domenom razdvajanje opisa izgrađenog ESOP optimizacijskog okruženja u dva dijela (četvrto poglavlje opisuje konceptualni model dok je sama izgradnja prikazana u petom poglavlju) donekle ide protiv osnovnih principa primijenjene metodologije. Međutim, to razdvajanje je primarno uvjetovano zahtjevima na strukturu disertacije koja predstavlja opis konačnog rezultata istraživanja, odnosno izgrađenog ESOP okruženja.

Elementi konceptualnog modela koji će biti opisani u ostatku ovog poglavlja stoga predstavljaju konkretne programske elemente koji postoje i u konačnoj implementaciji ESOP sustava. Većina opisanih koncepata će u izgrađenom sustavu biti realizirana preko razreda sučelja kojima će se obuhvatiti apstraktne karakteristike pripadajućih koncepata i koji će omogućiti daljnju nadgradnju sustava ugrađivanjem konkretnih razreda/objekata koji će se izvoditi iz definiranih sučelja.

### 4.2.2 Osnovna analiza

Teorijska razmatranja provedena u prvom i drugom poglavlju predstavljaju osnovu na kojoj će se izgraditi model domene programske podrške za optimizaciju. S obzirom da je definirani model domene prilično složen, u ovom odjeljku će se najprije dati globalni pregled domene problema, dok će se u kasnijim odjeljcima detaljnije opisati pojedini koncepti i njihove interakcije.

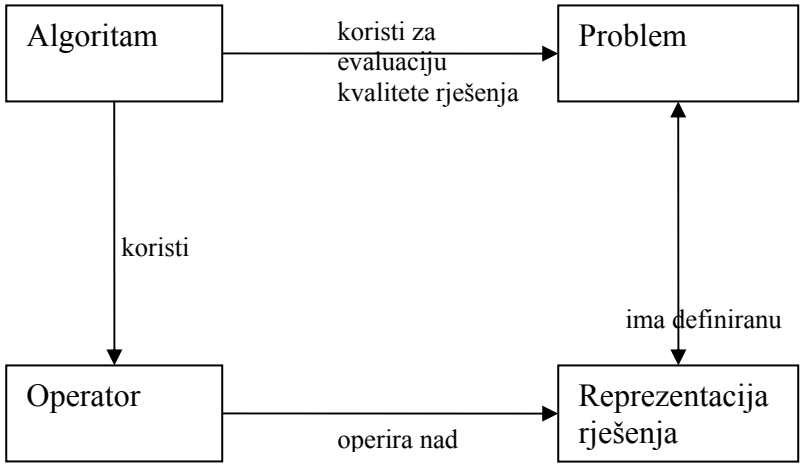
Izgradnja modela kao prvi korak zahtijeva definiranje osnovnih elemenata koji ga čine. U kontekstu programske podrške za optimizaciju možemo identificirati dva osnovna elementa modela u domeni programske podrške za optimizaciju: 1) *optimizacijski problem* i 2) *optimizacijski postupak* (*algoritam*) koji je primijenjen kod rješavanja danog problema.

Daljnjom razradom dolazimo do još dva koncepta koji dodatno opisuju domenu problema. Vezano uz optimizacijski problem, a naslanjajući se na klasifikaciju provedenu u drugom poglavlju, možemo identificirati koncept skupa ulaznih varijabli

problema odnosno skup ulaznih veličina problema čije se optimalne vrijednosti žele pronaći u procesu optimizacije. S obzirom da se taj skup varijabli u programskoj implementaciji mora modelirati (reprezentirati) preko dostupnih programskih tipova i da taj skup varijabli zajedno s njihovim vrijednostima predstavlja jedno rješenje optimizacijskog problema, dolazimo do novog elementa modela kojeg nazivamo *reprezentacijom rješenja*.

Drugi bitan element modela je vezan uz koncept optimizacijskog postupka, odnosno algoritma kojim je on realiziran. Iz opisa heurističkih metoda optimizacije, koje su primarni fokus ESOP optimizacijskog okruženja, proizlazi pojam *operatora* koji predstavlja integralni dio algoritma i koji djeluje nad zadanom reprezentacijom rješenja s određenim ciljem (koji je uvijek na neki način povezan s nalaženjem što boljeg rješenja danog optimizacijskog problema).

Sumiranjem navedenih karakteristika modela domene u cjelinu, možemo definirati sljedeći globalni konceptualni model domene programske podrške za optimizaciju.




**Slika 4.1 Prikaz osnovnih elemenata konceptualnog modela**

Opis pojedinih elemenata danog konceptualnog modela domene u kojem će se preko opisa uloga, odgovornosti i kolaboracija (engl. *roles, responsibilities, collaborations*) opisati njihove karakteristike je tema sljedećih nekoliko odjeljaka (za detaljan opis izgradnje objektnog modela baziranog na ulogama, odgovornostima i kolaboracijama objekata vidjeti [Wirfs-Brock2002]).

## 4.2.3 Koncept optimizacijskog problema

Osnovne karakteristike koncepta optimizacijskog problema, sa stajališta izgradnje općenitog optimizacijskog okruženja, proizlaze iz teorijske klasifikacije provedene u prvom poglavlju ove disertacije gdje su identificirana četiri kriterija za klasifikaciju optimizacijskih problema:

1) broj i vrsta funkcija cilja koje definiraju kriterije optimalnosti danog rješenja
2) broj i vrsta varijabli koje predstavljaju ulazne veličine problema
3) prisutnost postavljenih ograničenja u optimizacijskom problemu
4) postojanje parametara problema

Od ova četiri kriterija, od posebne su važnosti prvi i treći kriterij koji zahtijevaju daljnju razradu koncepta optimizacijskog problema. Kako je u prvom poglavlju već opisano, po prvom kriteriju optimizacijske probleme dijelimo na jednokriterijske i višekriterijske. Zbog značajne različitosti u izgradnji optimizacijskih postupaka namijenjenih rješavanju jednokriterijskih i višekriterijskih problema, nužno je u izgrađeni model uvesti dva zasebna koncepta koji predstavljaju respektivne vrste optimizacijskih problema. Budući da je za jezik imenovanja programskih elemenata kod izgradnje ESOP okruženja odabran engleski, ova dva koncepta su u skladu sa skraćenicama engleskih termina *single-objective* i *multi-objective* imenovana kao IProblemSO i IProblemMO.

U pogledu trećeg navedenog kriterija, optimizacijske probleme dijelimo na one koji nemaju definiranih ograničenja i one koji takva ograničenja imaju. S obzirom da i u ovom slučaju imamo značajne razlike u izgradnji i primjeni algoritama na probleme s ograničenjima u odnosu na one koji ih nemaju, ponovno je potrebno dodatno razraditi model uvođenjem novog koncepta koji će predstavljati probleme koji imaju definirana ograničenja – IProblemWC (gdje WC potječe od skraćenice engleskog termina *with constraints*). Uvođenjem ova tri koncepta koji predstavljaju specijalizaciju generalnog koncepta IProblem su pokrivene različite vrste optimizacijskih problema koji se javljaju u praksi.

Sljedeći korak je identifikacija zajedničkih karakteristika navedenih koncepata a koje proizlaze iz same definicije optimizacijskog problema. Jedna od osnovnih karakteristika svih optimizacijskih problema je da za određeni skup ulaza (odnosno vrijednosti ulaznih veličina – varijabli koje predstavljaju jedno rješenje optimizacijskog problema) na osnovu postavljenih kriterija optimalnosti (modeliranih preko funkcija cilja) izračunava skup vrijednosti izlaza koji određuju optimalnost danog rješenja. U model je stoga potrebno dodati dva nova koncepta – IProblemInput koji predstavlja „ulaz“ optimizacijskog problema, i IProblemOutput koji predstavlja njegov „izlaz“, te dodatnu odgovornost koncepta IProblem da na neki način zna provesti preslikavanje vrijednosti iz IProblemInput u IProblemOutput.

Ulaz za optimizacijski problem čini skup vrijednosti varijabli nad kojima je problem definiran. Pojedinačnu varijablu problema modeliramo konceptom IDesignVariable i opis tog koncepta je tema sljedećeg odjeljka. IProblemInput predstavlja skup svih varijabli problema iz čega automatski proizlaze i odgovornosti vezane uz kreiranje i održavanje tog skupa. Međutim, da bi IProblemInput mogao efikasno obavljati tu funkciju, nužan je preduvjet da poznaje strukturu ulaza problema, odnosno tipove varijabli koje se nalaze u tom skupu. Tipovi varijabli su definirani konkretnim optimizacijskim problemom te ovdje možemo ustanoviti interakciju između koncepata IProblem i IProblemInput. IProblem zna nad kakvim tipovima varijabli je definiran, dok IProblemInput predstavlja skup *konkretnih* reprezentacija tih varijabli.

Stoga uvodimo novi koncept IProblemInputDescriptor koji predstavlja opis karakteristika ulaznih varijabli problema. Kako IProblem ima informaciju o vrstama varijabli nad kojima je definiran, prirodno slijedi povezivanje ova dva koncepta na način da svaki IProblem ima pridružen i odgovarajući IProblemInputDescriptor. Kako je skup tipova varijabli bitan i za druge koncepte u domeni (poglavito IProblemInput koji je odgovoran za upravljanje skupom konkretnih varijabli problema koje se po tipu moraju slagati sa definicijom sadržanom u IProblemInputDescriptor konceptu), iz toga proizlazi dodatna odgovornost za IProblem a to je da mora pružiti mogućnost dohvata sadržanog opisa problema.

Izlaz optimizacijskog problema je modeliran konceptom IProblemOutput i njegova osnovna uloga je da sadrži podatke o vrijednostima izlaza izračunatim na osnovu vrijednosti ulaznih varijabli. Kako se izlaz optimizacijskog problema razlikuje ovisno o njegovoj vrsti, definirane su i četiri specijalizacije ovog koncepta:

- IProblemOutputSO – izlaz jednokriterijskog problema bez ograničenja (sadrži podatak o izračunatoj vrijednosti jedne funkcije cilja)
- IProblemOutputSOWC – izlaz jednokriterijskog problema s ograničenjima (sadrži vrijednost funkcije cilja i vrijednosti ograničenja)
- IProblemOutputMO – izlaz višekriterijskog problema bez ograničenja (sadrži izračunate vrijednosti svih funkcija cilja)
- IProblemOutputMOWC – izlaz višekriterijskog problema s ograničenjima (sadrži izračunate vrijednosti funkcija cilja i vrijednosti ograničenja)

### 4.2.4 Reprezentacije varijabli

Varijabla optimizacijskog problema predstavlja jednu ulaznu veličinu nad kojom je problem definiran. Prilikom klasifikacije varijabli problema u prvom poglavlju, opisano je nekoliko vrsta varijabli: realne varijable, cjelobrojne varijable, nizovi realnih i cjelobrojnih varijabli, varijable koje predstavljaju matematičke strukture permutacije ili pridruživanja. Svaka od tih vrsta varijabli ima svoj jedinstveni skup karakteristika koji proizlazi iz matematičkog koncepta koji varijabla predstavlja. Pitanje je kakve zajedničke karakteristike imaju sve navedene vrste varijabli.

Nažalost, raznolikost vrsta varijabli uvelike ograničava taj skup karakteristika. Stoga je koncept IDesignVariable koji predstavlja generalizaciju koncepta ulazne varijable problema relativno siromašan te se mogu definirati tri njegove odgovornosti:

1) varijabla problema se mora moći klonirati (odnosno, stvoriti svoju istovjetnu kopiju)
2) varijabla mora znati slučajno generirati svoju vrijednost
3) varijabla zna kreirati tekstualnu (string) reprezentaciju svojeg sadržaja

Konkretne karakteristike pojedinih vrsta varijabli problema će biti modelirane u konkretnijim konceptima koji će biti specijalizacije koncepta IDesignVariable. Tako će koncept realne varijable biti modeliran konceptom IReal koji će modelirati dodatne

karakteristike specifične za realnu varijablu: činjenicu da se takvoj varijabli može postaviti i dohvatiti vrijednost te karakteristiku realne varijable da ima definiran interval dopuštenih vrijednosti koje može imati. Opis karakteristika svih vrsta varijabli koje su ugrađene u ESOP optimizacijsko okruženje će biti dan u sljedećem poglavlju.

Povezano s varijablama problema, potrebno se osvrnuti na interakciju koje te varijable imaju s konceptima vezanim uz optimizacijski problem. Već je ustanovljena interakcija između ulazne varijable i IProblemInput koncepta koji sadrži skup tih ulaznih varijabli. Međutim, postoji i veza sa IProjectInputDescriptor konceptom koji sadrži informacije o vrstama varijabli nad kojima je definiran optimizacijski problem. Zbog toga je potrebno uvesti novi koncept IDesignVariableDescriptor koji će modelirati koncept vrste ulazne varijable. Bitno je naglasiti razliku između IDesignVariable i IDesignVariableDescriptor koncepata. IDesignVariable predstavlja konkretnu instancu varijable problema koja ima određenu vrijednost i koja se koristi za proračunavanje vrijednosti izlaza. IDesignVariableDescriptor s druge strane modelira koncept vrste varijable problema i kao takvog ga iskorištava IProblemInputDescriptor za opis skupa tipova ulaznih varijabli danog problema.

Razlog opisane dvojnosti leži u zahtjevu za proširivim optimizacijskim okruženjem. Naime, realizacija gotovo svake vrsta varijable problema u konkretan programski objekt se može izvesti na više načina. Primjerice, realna varijabla s rasponom [0,10] se može programski modelirati preko obične *float* varijable uz pripadne podatke koji definiraju interval. Međutim, isti koncept se može reprezentirati i preko polja bitova gdje se diskretizacijom dane realne varijable ona prevodi u cjelobrojni oblik koji se zatim prevodi u odgovarajuću bitovnu reprezentaciju. Ovakva reprezentacija realne varijable je prirodna za genetičke algoritme i mogućnost paralelnog korištenja i jedne i druge vrste realizacije je nužna za općenito optimizacijsko okruženje.

### 4.2.5 Koncept algoritma

Koncept optimizacijskog postupka predstavlja jedan od osnovnih koncepata u modelu domene programske podrške za optimizaciju. Uloga optimizacijskog postupka u domeni programske podrške za optimizaciju je jasna: on „zna“ naći rješenje danog optimizacijskog problema. Međutim, zbog raznolikosti načina rada različitih optimizacijskih postupaka i njihove pripadne implementacije, unutar generalnog koncepta optimizacijskog postupka se neće moći modelirati svi detalji pojedinih konkretnih optimizacijskih postupaka već će oni biti delegirani na konkretne programske realizacije koje će predstavljati specijalizaciju općenitog koncepta optimizacijskog postupka.

Ipak, usprkos tome, može se definirati skup zajedničkih karakteristika svih (za ESOP relevantnih) optimizacijskih postupaka koji proizlazi iz definiranog područja primjene ESOP-a. Već je navedeno da je ESOP okruženje prvenstveno namijenjeno iskorištavanju *iterativnih heurističkih* postupaka za rješavanje različitih optimizacijskih problema. Koncept iterativnog algoritma (imenovan IIterativeAlgorithm) stoga predstavlja općeniti koncept optimizacijskog postupka u ESOP optimizacijskom okruženju. Osnovna karakteristika iterativnog algoritma jest da se cijeli postupak

optimizacije obavlja u određenom broju iteracija. Iz toga proizlazi i osnovna odgovornost koncepta iterativnog algoritma a to je da zna obaviti jednu iteraciju danog optimizacijskog postupka. Ovdje je bitno naglasiti da se pretpostavlja da su sve iteracije algoritma jednake u pogledu programske realizacije, odnosno da odvijanje svake iteracije algoritma podrazumijeva izvršavanje istog slijeda instrukcija.

Uvođenje koncepta iteracije optimizacijskog postupka je bitno iz dva razloga. Prvi razlog je povezan uz definiranje trajanja optimizacije. Naime, zbog činjenice da je za većinu realnih optimizacijskih problema optimalno rješenje nepoznato i da heuristički algoritmi uglavnom nemaju dobro definiran kraj izvođenja, za razliku od egzaktnih algoritama kod kojih je trajanje određeno brojem egzaktnih koraka koje algoritam izvodi do nalaženja konačnog optimalnog rješenja, kod primjene heurističkog optimizacijskog algoritma na dani problem je potrebno točno odrediti trajanje provođenja optimizacije. Najčešće se trajanje optimizacije određuje ili preko definiranja broja iteracija koje će algoritam obaviti u potrazi za rješenjem ili preko zadavanja vremenskog trajanja optimizacije. S obzirom da se mogu definirati i drugi načini određivanja trajanja optimizacije (npr. provođenje optimizacije sve dok se ne obavi određeni broj evaluacija funkcija cilja, što je bitna karakteristika kod uspoređivanja različitih algoritama), u model ESOP okruženja je uveden i koncept IOptimizationTerminator. Njegova osnovna odgovornost je da definira kriterij završetka optimizacije. Odnosno, IOptimizationTerminator daje informaciju o tome da li je u trenutnoj iteraciji zadovoljen kriterij završetka optimizacije.

Drugi razlog važnosti koncepta iteracije leži u mogućnosti upravljanja tijekom izvođenja optimizacijskog postupka. Naime, definiranje koraka algoritma koji se izvode u jednoj njegovoj iteraciji nije dovoljno za potpuno opisivanje algoritma. Kod velike većine optimizacijskih postupaka postoji i određeni postupak inicijalizacije algoritma tijekom kojega se uspostavljaju potrebne programske strukture koje će se koristiti tijekom izvođenja. Stoga se kao dodatni zahtjev na koncept optimizacijskog algoritma uvodi i odgovornost za provođenje inicijalizacije cijelog postupka. Ovdje je bitno napomenuti da koncept IIterativeAlgorithm samo definira postojanje takve odgovornosti. Implementacija konkretnih operacija koje će se obavljati u procesu inicijalizacije je obveza konkretnih izgrađenih algoritama koji će specijalizirati koncept IIterativeAlgorithm.

Još jedna važna karakteristika koncepta optimizacijskog postupka proizlazi iz činjenice da prilikom realizacije danog postupka putem računalnog algoritma tom algoritmu mora biti poznat problem koji se rješava. Iz toga proizlazi i veza među koncepatima IIterativeAlgorithm i IProblem. Odnosno, IIterativeAlgorithm mora imati mogućnost postavljanja (određivanja) optimizacijskog problema na koji se želi primjeniti. Pored te veze, postoji i veza između algoritma i IProblemInput koncepta. Podatak o odabranoj reprezentaciji ulaznih varijabli problema (što određuje IProblemInput) je za algoritam važan jer će algoritam djelovati upravo nad tom reprezentacijom tijekom potrage za optimalnim rješenjem problema. Iz toga proizlazi da IIterativeAlgorithm mora poznavati, odnosno imati referencu na IProblemInput koncept.

Završna karakteristika koncepta optimizacijskog postupka je vezana uz koncept *rezultata optimizacije* (IOptimizationResult). Sam rezultat optimizacije kao koncept iz modela domene će se detaljnije obraditi u jednom od sljedećih odjeljaka. Ono što je bitno za optimizacijski postupak je da nakon završetka optimizacije mora postojati mogućnost dohvaćanja konačnog rezultata što predstavlja još jednu odgovornost koncepta IIterativeAlgorithm.

### 4.2.6 Koncept operatora

U generalnom pregledu modela domene definiran je koncept operatora optimizacijskog postupka koji proizlazi iz načina rada heurističkih algoritama. Za razliku od egzaktnih algoritama, većina heurističkih algoritama se bazira na principu iterativnog poboljšavanja rješenja. Odnosno, kreće se od nekog početnog rješenja (koje je u većini slučajeva slučajno generirano) te se zatim određenim postupkom pokušava naći bolje rješenje. U tom postupku pronalaženja boljeg rješenja, važnu ulogu imaju operatori danog optimizacijskog postupka.

Operatori se prirodno javljaju kod optimizacije genetičkim algoritmima koji se oslanjaju na operatore križanja i mutacije kao osnovne mehanizme za nalaženje rješenja. Međutim, mogu se definirati i drugi operatori. Primjerice, operator lokalnog pretraživanja koji pretražuje okolinu trenutnog rješenja s ciljem nalaženja boljeg rješenja, ili operator pridruživanja dobrote (engl. *fitness assignment*) koji jedinkama u populaciji genetičkog algoritma pridružuje fitnes.

U razvijenom modelu domene, operator je predstavljem konceptom IOperator. Međutim, s obzirom da su operatori primarno karakterizirani odgovornošću za obavljanje jedne operacije i da su te operacije po potrebnoj programskoj strukturi dosta različite te se ne može definirati skup zajedničkih karakteristika svih operatora, koncept IOperator u modelu predstavlja samo identifikator tipa čija je osnovna namjena da služi kao generalni koncept za sve vrste operatora. Odnosno, karakteristike pojedinih vrsta operatora će biti modelirane u specijalizacijama koncepta IOperator (IFitnessAssignment-Operator, ICrossover, IMutation, ...) a opisi pojedinih vrsta će biti dani u sljedećem poglavlju.

### 4.2.7 Koncept rješenja

Dosada opisani koncepti predstavljaju osnovne elemente domene koji su direktno povezani sa odgovarajućim teorijskim konceptima iz domene problema. Međutim, prilikom programske realizacije pokazuje se potreba za uvođenjem još jednog koncepta, a to je *rješenje* problema (ISolution).

Iako se na prvi pogled taj koncept čini suvišnim, s obzirom da je rješenje problema u stvari definirano vrijednostima ulaznih varijabli (IProblemInput), kod programske realizacije algoritma taj koncept nije dovoljan. Razlog tome leži u činjenici da IProblemInput modelira samo vrijednosti ulaza optimizacijskog problema dok je optimizacijskom algoritmu za efikasan rad potreban i podatak o pripadajućim izlazima, odnosno vrijednostima funkcija cilja i vrijednostima ograničenja, za dane vrijednosti ulaznih varijabli.

Koncept ISolution stoga predstavlja kombinaciju (engleski termin koji se koristi u literaturi jest *mix-in*) koncepata IProblemInput i IProblemOutput i predstavlja osnovni element modela nad kojim operiraju algoritmi. Osnovna odgovornost ISolution koncepta u modelu jest da referencira odgovarajuće IProblemInput i IProblemOutput elemente i da pruža mogućnost pristupa podacima koji su u njima definirani.

Međutim, odmah se postavlja i pitanje sinkronizacije vrijednosti ulaza i izlaza problema sadržanih u ISolution. Logičan je zahtjev da te vrijednosti budu u skladu, odnosno da u svakom trenutku vrijednosti izlaza odgovaraju postavljenim vrijednostima ulaza. Budući da ISolution mora pružiti mogućnost promjene podataka o ulaznim vrijednostima varijabli (što je jedna od osnovnih funkcionalnosti potrebnih kod izgradnje optimizacijskog postupka), a da je odgovornost za proračunavanje izlaza na osnovu ulaza pridijeljena konceptu optimizacijskog problema (IProblem), sinkronizacija je postignuta međudjelovanjem koncepata ISolution i IProblem. Odgovornosti za ažuriranje sadržanih vrijednosti je dodjeljena konceptu ISolution a kako proračun vrijednosti izlaza za dani ulaz zna obaviti optimizacijski problem, nužno je da ISolution ima referencu na IProblem kako bi mogao iskoristiti tu mogućnost.

Stoga je prilikom svake promjene vrijednosti varijabli (koju će gotovo uvijek obavljati optimizacijski algoritam) potrebno nad pripadnim rješenjem obaviti i ažuriranje vrijednosti izlaza. Odnosno, ispravno bi bilo reći da je potrebno obavijestiti ISolution da su vrijednosti varijabli promijenjene i da je potrebno obaviti ažuriranje.

S obzirom da postoji više specijalizacija IProblem koncepta, ovisno o vrsti optimizacijskog problema, definiran je i odgovarajući skup specijalizacija ISolution koncepta. U skladu s vrstom problema čije rješenje predstavljaju, definirane su sljedeće specijalizacije: ISolutionSO, ISolutionMO, ISolutionWC.

### 4.2.8 Koncept rezultata optimizacije

Sam rezultat optimizacije je ono što istraživača u konačnici najviše zanima i u modelu domene je on predstavljen konceptom IOptimizationResult. S obzirom na različiti karakter rezultata jednokriterijske i višekriterijske optimizacije (jedno globalno optimalno rješenje vs. skup Pareto rješenja) jasno je da će se u model morati ugraditi i odgovarajući koncepti koji će preciznije modelirati rezultate za pojedine vrste optimizacija.

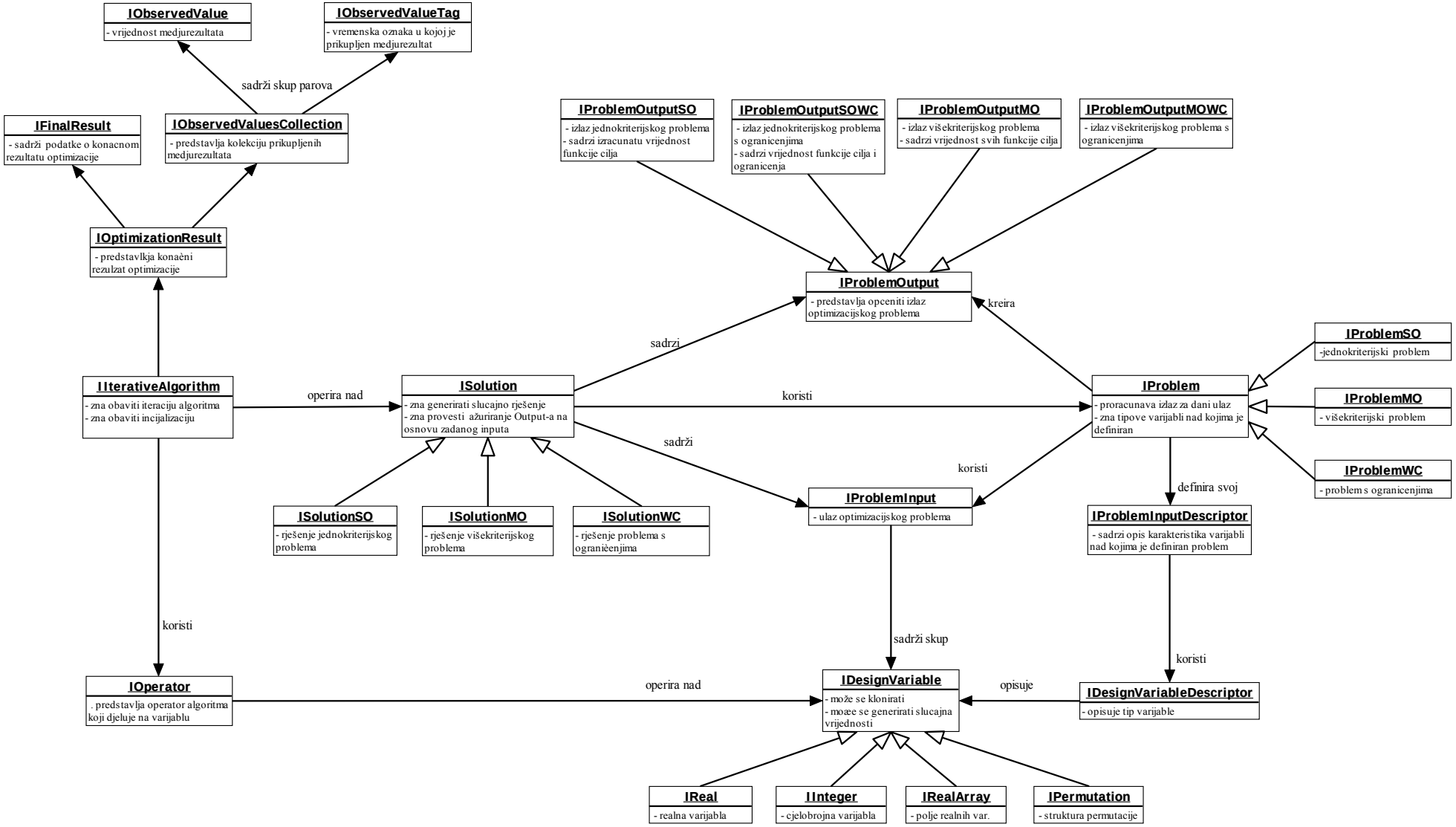
Međutim, kod provođenja optimizacije pomoću heurističkih algoritama je vrlo važno imati podatak i o međurezultatima koji su generirani tijekom procesa optimizacije, poglavito kako bi se mogao preciznije pratiti rad samog algoritma. Kako je skup potencijalno zanimljivih međurezultata vrlo širok, potrebno je omogućiti praćenje različitih vrsta međurezultata.

Stoga je model proširen definiranjem dodatnih koncepata *konačnog rezultata* (IFinalResult) i *kolekcije međurezultata* (IObservedValuesCollection). IFinalResult predstavlja općeniti koncept završnog rezultata optimizacije dok IObservedValuesCollection predstavlja općenitu kolekciju (bilo kakvih) međurezultata.

Kako bi se modelirale konkretne karakteristike različitih vrsta rješenja, u model će se ugraditi i dodatni elementi koji će predstavljati specijalizaciju IFinalResult.

IOptimizationResult koji modelira koncept cjelokupnog rezultata optimizacije stoga preuzima odgovornost za pružanje informacije o konačnom rezultatu i o skupu generiranih međurezultata. Detaljan opis načina ugradnje će biti dan u sljedećem poglavlju kod opisivanja izgrađenog objektnog modela ESOP optimizacijskog okruženja.

Na slici 4.2. je prikazan kompletni dijagram izgrađenog konceptualnog modela.




**Slika 4.2 Izrađeni konceptualni model domene programske podrške za optimizaciju**

# 5. Izgradnja optimizacijskog okruženja ESOP

Konceptualni model opisan u prethodnom poglavlju predstavlja rezultat analize domene programske podrške za optimizaciju. U njemu su sadržani osnovni koncepti prisutni u toj domeni i modelirane njihove međusobne interakcije. U skladu s postavkama dizajna pokretanog domenom, sljedeći cilj je izgradnja objektnog modela koji će predstavljati konkretnu programsku realizaciju danog konceptualnog modela.

Zahvaljujući mogućnostima apstrakcije unutar objektno-orijentirane paradigme mapiranje konceptualnog modela u konkretni objektni model će biti relativno jednostavno. Međutim, izgradnja ESOP okruženja će ipak zahtijevati dodatnu razradu opisanog modela, jer definirani konceptualni model predstavlja samo analizu domene. Kod konkretne izgradnje izvršne aplikacije potrebno je rasvijetliti mnogobrojne druge aspekte vezane uz tu izgradnju, a koji nisu adresirani tijekom izrade konceptualnog modela.

Prvi zadatak u toj izgradnji je određivanje arhitekture programskog rješenja. Arhitektura informacijskog sustava primarno proizlazi iz zahtjeva postavljenih na sam sustav te je stoga najprije potrebno analizirati kakav utjecaj će zahtjevi postavljeni na ESOP optimizacijsko okruženje imati na njegovu arhitekturu.

## 5.1 Utjecaj zahtjeva na izgradnju ESOP-a

U trećem poglavlju je detaljno opisan skup zahtjeva na općenito optimizacijsko okruženje. Od navedenih zahtjeva, dva su od velike važnosti kod definiranja arhitekture sustava jer način njihove realizacije, odnosno zadovoljavanja tih zahtjeva, uvelike utječe na arhitekturu. Ta dva zahtjeva su: omogućavanje jednostavnog korištenja optimizacijskog okruženja i mogućnost njegovog proširivanja.

### 5.1.1 Jednostavnost korištenja

Jednostavnost korištenja je jedan od osnovnih zahtjeva na svaki informacijski sustav. Međutim, u slučaju razvoja općenitog okruženja za optimizaciju taj zahtjev nije lako zadovoljiti. Osnovni razlog tome leži u širini domene problema. Kako je već opisano u trećem poglavlju, općenito optimizacijsko okruženje bi trebalo pružati mogućnost primjene različitih optimizacijskih postupaka na široki skup optimizacijskih problema. Različitost među optimizacijskim problemima i postupcima, međutim, povlači i značajne razlike u njihovoj izgradnji. Kako je u današnje vrijeme pojam jednostavnog korištenja aplikacije neizostavno povezan sa mogućnošću rada s aplikacijom preko GUI sučelja postavlja se pitanje kako omogućiti vizualno iskorištavanje tako različitih programskih komponenti.

Tehnička složenost omogućavanja takve funkcionalnost je osnovni razlog zašto većina postojećih okruženja za optimizaciju ne pruža takvu mogućnost. Odnosno, većina ih je izgrađena kao biblioteka razreda čije iskorištavanje zahtjeva

određenu količinu programiranja u nekom razvojnom okruženju. Takva rješenja imaju jednu veliku prednost i jednu veliku manu.

Prednost takvog rješenja leži u širokoj mogućnosti iskorištavanja ugrađenih programskih komponenti (razreda). Naime, korištenje postojeće biblioteke razreda na nivou izvornog kôda pruža istraživaču mogućnost potpune kontrole. Ugrađene programske komponente se mogu iskorištavati i kombinirati na različite načine, a vrlo često postoji i mogućnost njihovog razrađivanja (u smislu redefiniranja određene funkcionalnosti) i ugrađivanja novih komponenti.

Nažalost, iz toga proizlazi i velika mana, a to je da istraživač mora biti vješt u korištenju danog razvojnog okruženja, odnosno, mora poznavati pripadni programski jezik i odgovarajuće tehnike programiranja. Iako se mora priznati da su graditelji većine postojećih biblioteka uložili veliki trud u olakšavanje njihovog korištenja, tako da u nekim slučajevima rješavanje jednostavnih optimizacijskih problema zahtjeva pisanje svega nekoliko desetaka linija programskog koda, pojavljuje se i dodatni problem. S rastom složenosti biblioteke raste i složenost njenog iskorištavanja koje tada zahtjeva značajan trud kod podešavanja razvojnog okruženja. Podešavanje raznih staza do odgovarajućih direktorija, definiranje #include datoteka, korištenje makroa za odabir pojedinih dijelova funkcionalnosti (*#define*, *#if*, *#elif* konstrukti u C++) su aktivnosti koje čak i iskusne programere mogu odbiti od korištenja dane biblioteke. Ukoliko se uzme u obzir složenost modernih razvojnih okruženja koja nameće specijalizaciju i odabir preferabilnog okruženja, problem postaje još teži.

Optimizacijsko okruženje koje pruža mogućnost rada kroz GUI sučelje očigledno rješava opisani nedostatak. Međutim, istraživač za to mora platiti i određenu cijenu jer se gubi mogućnost potpune kontrole. Naime, optimizacijsko okruženje s GUI sučeljem mora unaprijed definirati načine korištenja pojedinih programskih komponenti i načine njihovog kombiniranja te je prilikom definiranja cjelokupnog konteksta optimizacije (problem + primjenjeni algoritam) istraživač ograničen predefiniranim karakteristikama vizualnog sučelja.

Iz navedenih razloga se zahtjev za jednostavnosti korištenja mora donekle modificirati u smislu da bi općenito optimizacijsko okruženje u idealnom slučaju moralo pružati mogućnost korištenja na oba navedena načina. Odnosno, za napredne korisnike pružiti mogućnost iskorištavanja ugrađene funkcionalnosti na nivou izvornog koda, ali istovremeno omogućiti i jednostavno korištenje kroz vizualno sučelje, bez potrebe za dodatnim programiranjem.

### 5.1.2 Proširivost

Drugi zahtjev koji ima značajan utjecaj na arhitekturu optimizacijskog okruženja je povezan uz njegovu proširivost pod čime podrazumijevamo mogućnost ugradnje novoizgrađenih optimizacijskih komponenata u optimizacijsko okruženje.

Postizanje proširivosti je jednostavnije u optimizacijskim okruženjima koja su izgrađena kao biblioteke razreda, jer u tom slučaju graditelj novih komponenti ima mogućnost interakcije s već ugrađenim komponentama na nivou izvornog koda. Međutim, skup programskih tehnika kojima se omogućava proširivost izgrađenog sustava je isti i u slučaju optimizacijskog okruženja s GUI sučeljem i bazira se na konceptu sučelja (enlg. *interface*) u objektno-orijentiranoj paradigmi.

Sučelja, odnosno preciznije rečeno razredi sučelja (engl. *interface class*), u OO paradigmi predstavljaju jedan od osnovnih elemenata i omogućavaju korištenje *design by contract* [Mayer1988] paradigme razvoja. Pojednostavljeno rečeno, razredi sučelja definiraju (deklariraju) skup operacija čija će se konkretna implementacija ugraditi u izvedenim razredima. Izvođenjem konkretnog razreda iz razreda sučelja konkretni razredi preuzimaju obvezu implementacije svih metoda deklariranih u sučelju čime se uspostavlja *je-vrsta-od* (engl. *is-a-kind-of*) hijerarhijski odnos između razreda sučelja i konkretnog izvedenog razreda. U OO terminologiji se kaže da konkretni razred *zadovoljava* definirano sučelje.

Osnovna prednost korištenja sučelja je da omogućavaju iskorištavanje izgrađenih komponenti po principu crne kutije. Odnosno, sve dok izgrađena komponenta (razred) zadovoljava zadano sučelje ostatak programa njom zna upravljati i s njom može komunicirati (preko definiranog sučelja). Dodatno svojstvo korištenja sučelja, a koje fundamentalno omogućava proširivost izgrađenih sustava s novim komponentama, jest da te komponente mogu biti naknadno izgrađene. Da bi se izgrađeni sustav mogao proširiti, potrebno je samo definirati određeni skup sučelja koja će se kasnije realizirati u izgrađenim programskim komponentama.

U kontekstu definiranog konceptualnog modela to znači da će većina definiranih koncepata biti upravo programski realizirana preko razreda sučelja. Odgovornosti i karakteristike pojedinih elemenata modela će biti modelirane preko skupa čistih virtualnih funkcija (engl. *pure virtual function*) odgovarajućeg sučelja a konkretna realizacija će biti ugrađena u razredima izvedenim iz tih sučelja.

Bitan detalj kod iskorištavanja razreda koji zadovoljavaju sučelja je vezan uz njihovo instanciranje. Instanciranje razreda podrazumijeva kreiranje objekta tog tipa. Međutim, da bi se mogao kreirati objekt definicija pripadnog razreda mora biti poznata. Ovdje do izražaja dolazi razlika iskorištavanja izgrađenih razreda na statičkom (engl. *compile-time*) odnosno dinamičkom (engl. *run-time*) nivou.

Statičko iskorištavanje izgrađenih programskih komponenti ponajprije podrazumijeva njihovo iskorištavanje u vidu biblioteke razreda. Za program koji se gradi to prvenstveno znači da su prilikom prevođenja poznati svi tipovi koji su definirani unutar tih biblioteka (primarno razredi, ali i ostali programski konstrukti: enumeracije, konstante, ...). Tako izgrađeni program nakon prevođenja i povezivanja predstavlja završenu (i monolitnu) cjelinu. Jasno je da su u tom slučaju mogućnosti proširenja ograničene. Ili, preciznije rečeno, takav program je moguće naknadno proširiti dodatnim komponentama, ali to zahtijeva modifikacije u izvornom kodu programa i njegovo ponovno prevođenje i povezivanje.

Dinamičko iskorištavanje izgrađenih programskih komponenti se oslanja na pojam dinamički učitane biblioteke (engl. *dynamically loaded library* - *dll*). Osnovna razlika u odnosu na statičke biblioteke razreda (npr. lib datoteke kod C++a) leži u tome da dll-ovi ne zahtjevaju povezivanje s izgrađenom aplikacijom u jednu monolitnu cjelinu. Oni postoje kao posebna vrsta datoteka koje dana aplikacija po potrebi učitava u svoj izvršni kontekst (engl. *execution context*). Međutim, bitno je naglasiti da se i dinamički učitane biblioteke mogu koristiti na statički način. Ovo zahtijeva dodatno pojašnjenje.

Statičko iskorištavanje komponente znači da je njena definicija poznata prilikom prevođenja programa. Kod korištenja statičkih biblioteka razreda to znači

da će prevodilac provjeriti slaganje tipova između programa i komponente (koja prema van izlaže skup .h datoteka, u slučaju C++a), te ukoliko je to slaganje potpuno kreirati monolitnu izvršnu aplikaciju. Preciznije, povezivač će generirati jednu .exe datoteku u koju će biti ugrađen cjelokupan programski kôd aplikacije.

Statičko iskorištavanje dinamički učitane biblioteke (*dll*) je donekle drugačije, ali primarno sa stajališta povezivača. Naime, ukoliko se žele iskoristiti programski elementi ugrađeni u *dll* (npr. pozvati funkciju), prilikom prevođenja programa i u ovom slučaju njihove deklaracije moraju biti poznate. Stoga je jedina razlika u odnosu na statičke biblioteke razreda/funkcija u tome što povezivač ne spaja sav izvršni kôd u jednu cjelinu, već je on razdijeljen na izvršnu aplikaciju (exe) i skup dinamičkih biblioteka (*dll*). Korišteni *dll*-ovi će prilikom pokretanja aplikacije biti učitani u njen izvršni kontekst te će zajedno s aplikacijom opet predstavljati monolitnu cjelinu.

Kod općenitog optimizacijskog okruženja trebalo bi imati mogućnost dinamičkog iskorištavanja programskih komponenti, u smislu da se u izvršni kontekst aplikacije mogu učitati i programske komponente koje nisu bile predviđene i izgrađene u vrijeme izgradnje same aplikacije. Kod standardnih dll-ova koji su primarno predstavljali nakupinu samostalnih funkcija takva mogućnost je također postojala, ali je zahtijevala poštivanje predefiniranih deklaracija funkcija. Značajna karakteristika .NET razvojnog okruženja jest mogućnost definiranja pravih dinamičkih biblioteka razreda, gdje je unutar te biblioteke sadržana cjelokupna definicija razreda koja prilikom učitavanja u izvršni kontekst postaje dostupna aplikaciji. Govoreći tehničkim žargonom – razred/objekt preživljava prelazak granice između dll-a i .exe aplikacije. Stoga razredi ugrađeni u .NET biblioteke razreda postaju dostupni u svoj svojoj punoći unutar konteksta aplikacije te se shodno tome mogu i programski iskorištavati.

Dodatna vrlo važna karakteristika .NET okruženja jest i njegova podrška za dinamičko otkrivanje sadržaja biblioteka razreda (*reflection* mehanizam). Naime, da bi se razredi ugrađeni u programske komponente mogli dinamički iskorištavati, potrebno ih je unutar dll-ova pronaći te saznati sve informacije potrebne da bi se mogao instancirati takav tip objekta. Funkcionalnost ugrađena u .Reflection prostoru imena (engl. *namespace*) .NET okruženja je krucijalna za izgradnju ESOP razvojnog okruženja a detaljni opis tog aspekta izgradnje dan je u potpoglavlju 5.6.

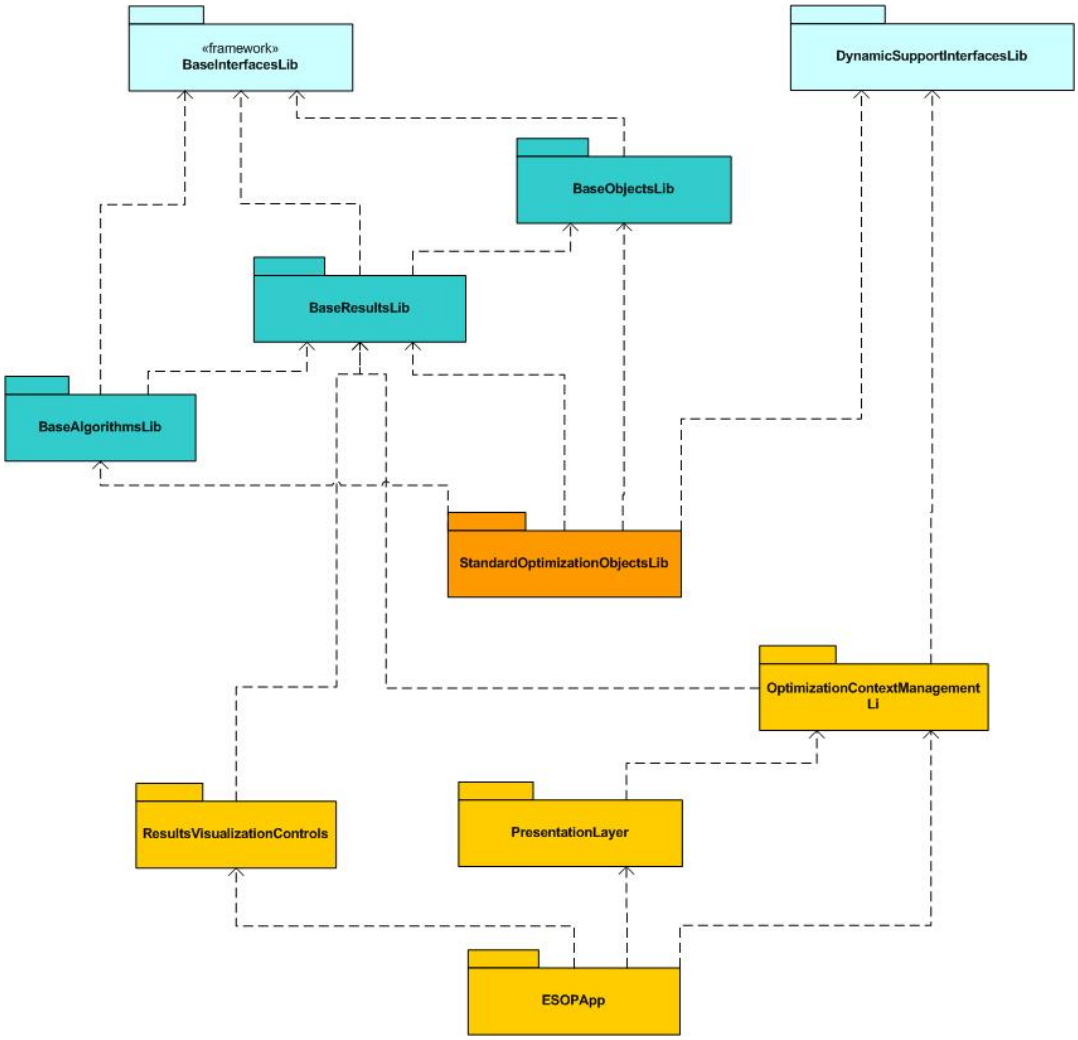
## 5.2 Arhitektura ESOP-a

Iz navedenog opisa utjecaja postavljenih zahtjeva na ESOP okruženje proizlazi i njegova arhitektura. Kako bi se omogućila oba navedena načina iskorištavanja optimizacijskih programskih komponenti, unutar arhitekture ESOP okruženja su definirana dva sloja programskih komponenti (engl. *layer*).

Prvi sloj čini skup projekata, u biti biblioteka razreda, koji definiraju ESOP razvojnu okosnicu. Unutar tog sloja se nalazi izgrađen skup razreda sučelja i pripadnih programskih komponenti koji je namijenjen statičkom iskorištavanju kako je opisano u prethodnom poglavlju. Iskorištavanje tih programskih komponenti zahtjeva izgradnju posebne aplikacije unutar koje će ugrađeni razredi / objekti koristiti na nivou izvornog kôda. Osnovni element tog sloja čini biblioteka BaseInterfacesLib unutar koje su definirani razredi sučelja koji realiziraju opisani

konceptualni model. Pored te biblioteke su unutar ovog sloja izgrađene i tri dodatne biblioteke razreda (BaseObjectsLib, BaseResultsLib i BaseAlgorithmsLib) unutar kojih su realizirani razredi koji predstavljaju konkretne objekte/koncepte prisutne u domeni programske podrške za optimizaciju.

Drugi sloj čini skup projekata preko kojeg je izgrađeno izvršno (engl. *run-time*) ESOP okruženje. Osnovni element tog sloja čini ESOPApp aplikacija koja predstavlja ljusku s GUI sučeljem unutar koje se izgrađene programske komponente mogu koristiti na dinamički način. Odnosno, ESOPApp aplikacija omogućava dinamičko učitavanje izgrađenih optimizacijskih komponenti i manipulaciju s njima preko vizualnog sučelja. Pored toga su unutar ovog sloja ugrađene i komponente za vizualizaciju rezultata (ResultVisualizationControls biblioteka), i skup komponenti koji omogućava dinamičko definiranje optimizacijskog programa (OptimizationContext-ManagementLib) i njegovu GUI prezentaciju (PresentationLayerLib)




**Slika 5.1 Arhitektura ESOP optimizacijskog okruženja**

Vrlo bitan element arhitekture je i biblioteka DynamicSupportInterfacesLib koja predstavlja most između statičkog i dinamičkog dijela ESOP okruženja. Njena osnovna zadaća je da omogući jednostavno iskorištavanje i statički izgrađenih programskih optimizacijskih komponenti unutar izgrađene ESOP ljuske (ESOPApp aplikacije). Taj cilj je ostvaren definiranjem dodatnog skupa razreda sučelja preko kojih će ESOPApp aplikacija koristiti statičke programske komponente. To znači da će mogućnost iskorištavanja izgrađenih programskih komponenti unutar ESOP ljuske zahtijevati od graditelja tih komponenti tehnički jednostavno nasljeđivanje dodatnog skupa sučelja, preko kojih će ESOPApp aplikacija s njima komunicirati. Ukoliko se izgrađene optimizacijske komponente ne žele koristiti unutar run-time ESOP okruženja taj skup sučelja se može jednostavno zanemariti.

Još se postavlja pitanje gdje se unutar opisanog arhitekturnog modela nalaze izgrađene konkretne optimizacijske komponente - problemi, algoritmi, operatori. One predstavljaju zaseban dio arhitekture i grade se kao standardni *dll*-ovi u NET razvojnom okruženju s time da je prilikom njihove izgradnje jedino potrebno referencirati *dll*-ove iz ESOP framework-a. Unutar razvijenog ESOP okruženja je skup već ugrađenih optimizacijskih komponenti realiziran preko StandardOptimi-zationObjectsLib biblioteke razreda, ali je moguće (i očekivano) izgraditi i nove biblioteke s ugrađenim novim optimizacijskim komonentama. Bitno je naglasiti da sve optimizacijske komponente izgrađene unutar StandardOptimizationObjectsLib biblioteke zadovoljavaju sučelja definirana u DynamicSupportInterfacesLib biblioteci što automatski omogućava njihovo korištenje unutar run-time ESOP okruženja.

## 5.3 Opis infrastrukture ESOP okruženja

U ovom potpoglavlju će se opisati aspekti izgradnje statičkog dijela ESOP optimizacijskog okruženja koje omogućava izgradnju optimizacijskih komponenti predviđenih za iskorištavanje na nivou izvornog kôda. Opis je podijeljen u tri dijela. U prvom dijelu će se dati opis skupa osnovnih sučelja i razreda koji čine osnovu ESOP okruženja. Prikazat će se programska realizacija konceptualnog modela preko odgovarajućeg skupa razreda sučelja i izgradnja konkretnih razreda koji su prisutni u domeni programske podrške za optimizaciju.

Nakon toga slijedi opis ugrađenih programskih komponenti za optimizaciju. Opisati će se izgradnja razreda koji će predstavljati konkretne optimizacijske probleme, algoritme, vrste varijabli i operatore. U završnom dijelu potpoglavlja će se opisati i skup izgrađenih pomoćnih programskih komponenti, primarno namijenjenih vizualizaciji rezultata optimizacije.

### 5.3.1 Skup osnovnih sučelja i razreda

Osnovu ESOP okruženja čini sljedeći skup biblioteka razreda: BaseInterfacesLib, BaseObjectsLib, BaseAlgorithmsLib i BaseResultsLib. Razredi ugrađeni u ovim bibliotekama predstavljaju realizacije koncepata iz konceptualnog modela. Iako skup definiranih razreda predstavlja jednu cjelinu, iz arhitekturnih razloga su razdvojeni u različite biblioteke kako bi se razdvojili pojedini aspekti funkcionalnosti koje izgrađeni razredi implementiraju.

#### 5.3.1.1 Preslikavanje konceptualnog modela u programsku realizaciju

Prije samog opisa navedenih biblioteka razreda potrebno je preciznije opisati vezu između koncepata iz razvijenog konceptualnog modela i razreda sučelja. Razredi sučelja u objektno-orijentiranoj paradigmi se po svojim karakteristikama značajno razlikuju od „običnih“ razreda (u OO terminologiji se koristi termin konkretan razred). Konkretni razredi predstavljaju predloške iz kojih se instanciraju objekti tog razreda, a svaki objekt ima skup članskih varijabli i članskih funkcija koji je definiran u pripadnom razredu. U tom pogledu su svi objekti instancirani iz jednog razreda jednaki. Ono po čemu se razlikuju je *stanje* koje je definirano *vrijednostima* članskih varijabli objekta / razreda. Objekti se razlikuju i po ponašanju u smislu da će se tijek izvođenja članskih funkcija pozvanih nad razredom razlikovati od objekta do objekta, ovisno o njegovom stanju.

Osnovna karakteristika po kojoj se razredi sučelja razlikuju od konkretnih razreda je da se ne mogu instancirati konkretni objekti iz razreda sučelja. Razredi sučelja predstavljaju *apstraktne* razrede (engl. *abstract class*) koji su primarno namijenjeni za daljnje nasljeđivanje.

Ovdje je potrebna mala digresija, uvjetovana tehničkim razlikama u pojmu razreda sučelja u standardnom C++ programskom jeziku i .NET razvojnom okruženju. Naime, u C++u razred sučelja ne postoji kao zasebni element programskog jezika. Razred sučelja u C++u je standardan razred, ali uz dodatnu karakteristiku da ima definiranu barem jednu *čistu virtualnu funkciju*. Čiste virtualne

funkcije su u stvari virtualne članske funkcije koje predstavljaju *deklaraciju* funkcije čija će konkretna implementacija biti dana u izvedenom (naslijeđenom) razredu. Preciznije rečeno, svaki razred izveden iz razreda sučelja mora ugraditi definiciju svih čistih virtualnih funkcija kako bi se mogli instancirati objekti tog razreda.

U .NET razvojnom okruženju je pojam razreda sučelja uveden kao novi programski element na način da se korištenjem ključne riječi *interface class* kod deklaracije razreda dani razred proglašava razredom sučelja. Sintaksna razlika se najbolje može ilustrirati primjerom:

Standardni C++

```
class IMojeSucelje
{
public:
        virtual void        Funkcija() = 0;
};
```

C++/CLI

```
public interface class IMojeSucelje
{
public:
        virtual void        Funkcija() = 0;
};
```

C#

```
public interface IMojeSucelje
{
        public    virtual void        Funkcija();
};
```

Pored sintaksne razlike, između razreda sučelja u standardnom C++u i razredima sučelja u .NET okruženju postoji i semantička razlika. Naime, kako razred sučelja u C++u predstavlja običan razred (koji je primarno karakteriziran postojanjem čistih virtualnih funkcija), iz toga proizlazi da takvi razredi mogu imati i karakteristike običnih razreda. Odnosno, razred sučelja u C++u može sadržavati i članske varijable i članske funkcije s *pripadnom implementacijom*. Iako se na prvi pogled čini besmisleno da razred sučelja ima članske varijable s obzirom da se ne mogu instancirati objekti tog razreda, bitno je naglasiti da će te članske varijable izvođenjem (naslijeđivanjem) konkretnog razreda iz razreda sučelja postati dio definicije konkretnog razreda.

Za razliku od standardnog C++a, u .NET razvojnom okruženju je pojam razreda sučelja definiran na čišći način. Razredi sučelja u .NET okruženju *ne mogu* imati članske varijable (znači ne mogu imati definirano stanje) već samo deklaracije virtualnih funkcija kojima će izvedeni razredi dati svoje značenje.

Razredi sučelja predstavljaju izuzetno važan element objektno-orijentirane paradigme, koji je kod izgradnje ESOP optimizacijskog okruženja iskorišten do maksimuma. Njihova važnost proizlazi iz mogućnosti da se preko definiranja (u stvari deklariranja) određenog skupa virtualnih članskih funkcija modelira skup odgovornosti koncepta koji taj razred sučelja predstavlja.

Ukoliko uzmemo za primjer koncept IProblem, koji predstavlja koncept općenitog optimizacijskog problema, za njega je kao jedna od osnovnih odgovornosti

definirana njegova mogućnost da na osnovu danog ulaza problema može izračunati odgovarajući skup izlaza (vrijednosti funkcije/a cilja). S obzirom da dano izračunavanje ovisi o konkretnoj vrsti optimizacijskog problema, postavlja se pitanje kako prisiliti konkretne realizacije koncepta IProblem da realiziraju tu mogućnost. Odgovor na ovo pitanje se najbolje može vidjeti iz konkretnog primjera u izvornom kôdu:

```
public interface class IProblem
{
public:
        virtual    void      CalcProblemOutput(IProblemInput ^inPI,
                                                IProblemOutput ^outPO) = 0;
        // ... ostatak deklaracije
};
public ref class DeJong_F1_TestProblem : public IProblem
{
public:
     virtual   void  CalcProblemOutput( IProblemInput ^inPI,
                                        IProblemOutput ^outPO) override
    {
        IRealArray     ^arrX = dynamic_cast<IRealArray ^>(inPI->GetDesignVariable(0));

        double sum = 0;
        for( int i=0; i<3; i++ )
        {
            double x = arrX->getValue(i);
            sum += (x - 1) * (x - 1);
        }

        ProblemOutputSO      ^outVal = dynamic_cast<ProblemOutputSO ^>(outPO);
        outVal->setObjectiveValue(sum);

    }
    // ... ostatak definicije razreda
};

void NekaFunkcija(IProblem ^inProblem)
{
        inProblem.CalcProblemOutput( ... );
};

void GlavnaFunkcija()
{
        IProblem ^problem = gcnew DeJong_F1_TestProblem();
        // ...
        NekaFunkcija(problem);
}
```

*Programski odsječak 1: Primjer realizacije sučelja u konkretnom razredu.*

U gornjem programskom odsječku je definiran razred sučelja IProblem i konkretan razred DeJong_F1_TestProblem koji predstavlja jedan od optimizacijskih problema iz DeJong-ovog skupa testnih problema. Također su definirane i dvije funkcije za ilustraciju načina kreiranja odgovarajućih objekata i pozivanja njihovih članskih funkcija.

Osnovna opservacija jest sljedeća: svaki konkretni razred izveden iz razreda sučelja IProblem mora imati ugradnju (implementaciju) funkcije CalcProblemOutput, koja po prototipu točno odgovara deklaraciji u razredu sučelja.

Kako se svi konkretni razredi koji predstavljaju (različite) optimizacijske probleme u ESOP optimizacijskom okruženju moraju izvesti iz razreda sučelja IProblem (ili neke od njegovih specijalizacija), na taj način je postignuto da svi ti konkretni razredi imaju ugrađenu funkcionalnost za proračun izlaza problema na osnovu danog ulaza. Time je apstraktno definirana odgovornost koncepta optimizacijskog algoritma prevedena u konkretnu programsku realizaciju. Odnosno, odgovornost definirana za pojedini koncept (ono što taj koncept „zna“) se modelira odgovarajućom članskom funkcijom u pripadnom razredu sučelja (koji u programskoj realizaciji predstavlja taj koncept).

Kako bi mapiranje razvijenog konceptualnog modela u konkretnu programsku realizaciju bilo što jasnije, potrebno je opisati i značenje ostalih opisanih karakteristika koncepta i njihovih relacija. Često korištena relacija među konceptima je njihovo međusobno „poznavanje“. Npr. kod opisa koncepta IIterativeAlgorithm je navedeno da mu mora biti poznat optimizacijski problem koji se rješava. U konkretnoj programskoj realizaciji koncepta IIterativeAlgorithm to znači da on mora imati referencu na konkretni objekt koji predstavlja dani optimizacijski problem. S obzirom da u .NET okruženju razredi sučelja ne mogu imati članske varijable, navedena karakteristika koncepta algoritma je u razredu sučelja IIterativeAlgorithm modelirana definiranjem para *get/set* funkcija koje za dani objekt algoritma postavljaju ili vraćaju referencu na objekt optimizacijskog problema.

Druga često korištena relacija među konceptima je „korištenje“. Npr. IIterativeAlgorithm *koristi* IOperator tijekom provođenja optimizacije, i *operira nad* ISolution konceptom koji predstavlja jedno rješenje danog optimizacijskog problema. U oba slučaja to znači da IIterativeAlgorithm mora imati referencu na odgovarajući objekt operatora (izveden iz IOperator sučelja), odnosno objekt rješenja (izveden iz ISolution sučelja). Iz već spomenutog razloga (nemogućnost postojanja članske varijable, koja bi predstavljala referencu unutar razreda sučelja) ovakva vrsta relacija će također zahtijevati modeliranje preko odgovarajućih virtualnih članskih funkcija za postavljenje ili dohvaćanje referenci na odgovarajuće objekte.

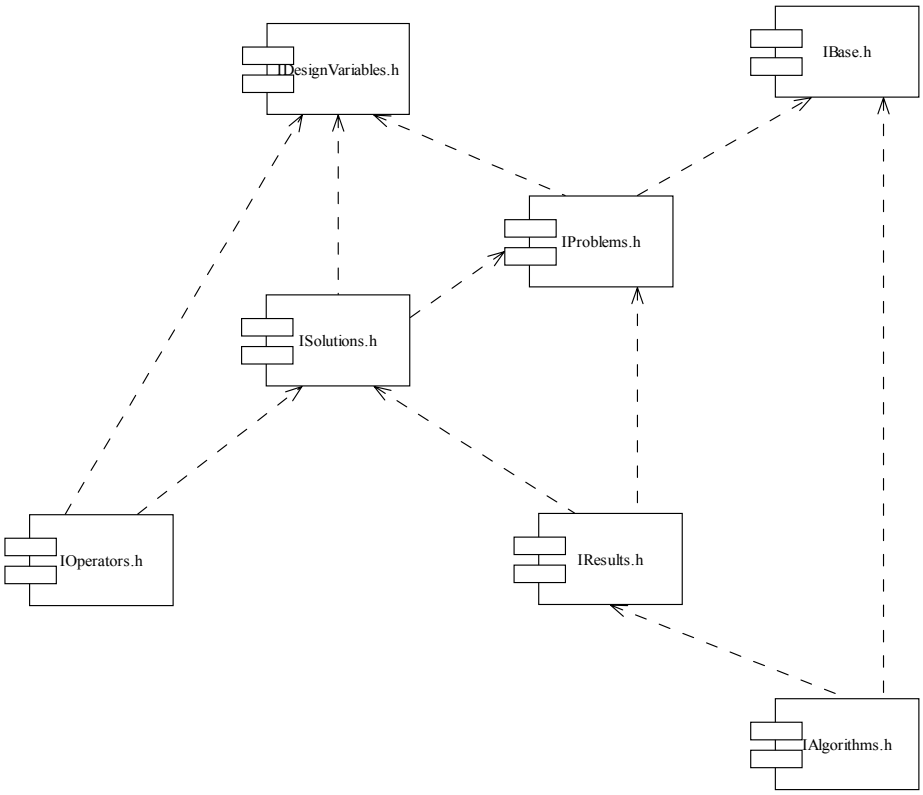
Za kraj je potrebno opisati i značenje specijalizacije koncepta. U konkretnoj programskoj realizaciji to znači da će se razred sučelja koji odgovara specijaliziranom konceptu izvesti iz razreda sučelja koji odgovara generalnom konceptu. U definiranom konceptualnom dijagramu primjer takve relacije imamo između koncepata IProblem i njegovih specijalizacija IProblemSO, IProblemMO i IProblemWC. Osnovni koncept IProblem definira skup karakteristika i odgovornosti koje su zajedničke za sve optimizacijske probleme. Specijalizirani koncepti izvođenjem iz razreda sučelja IProblem preuzimaju i njegovu cjelokupnu definiciju, s time da dodaju karakteristike i odgovornosti specifične za koncept koji modeliraju (respektivno, jednokriterijski problem, višekriterijski problem i problem s ograničenjima). Uspostavljanjem hijerarhije koncepata, odnosno razreda sučelja, se omogućava preciznije razrađivanje modela domene i točnije modeliranje odgovarajući koncepata koji se u toj domeni pojavljuju.

U sljedećim odjeljcima slijedi detaljni opis programske realizacije biblioteka razreda koje predstavljaju osnovu ESOP optimizacijskog okruženja.

#### 5.3.1.2 BaseInterfacesLib

Ova biblioteka razreda predstavlja dušu ESOP razvojnog okruženja jer je u njoj ugrađen skup razreda sučelja koji se direktno oslanjaju na definirani konceptualni model. Definirane razrede sučelja možemo podijeliti u dvije skupine. U prvoj skupini su razredi sučelja koji predstavljaju osnovne točke proširivanja ESOP okruženja. Oni su primarno namijenjeni nasljeđivanju od strane konkretnih programskih optimizacijskih komponenti / razreda i u tu skupinu spadaju IProblem, IDesignVariable, IIterativeAlgorithm i IOperator. Bitno je naglasiti da je u ESOP okruženje ugrađen i skup standardnih optimizacijskih komponenata koji se oslanja na navedene razrede sučelja, ali je moguće i definirati nove.

U drugoj skupini se nalaze razredi sučelja koji unutar ESOP okruženja imaju svoju generičku implementaciju i koji nisu namijenjeni daljnjem nasljeđivanju, odnosno redefiniranju njihovog ponašanja. U tu skupinu spadaju ISolution, IFinalResult i ostali razredi sučelja povezani s konceptom rezultata. Razlog zbog kojega u ESOP okruženje nije potrebno ugrađivati nove komponente bazirane na tim sučeljima je što je sva potrebna funkcionalnost već ugrađena u generičkim implementacijama ugrađenim u ESOP. Specijalizacije navedenih sučelja primarno proizlaze iz potrebe za modeliranjem različitih vrsta optimizacijskih problema (SO, SOWC, MO i MOWC), a kako je taj skup konačan, jer pokriva sve klase optimizacijskih problema podržane od strane ESOP-a, nema potrebe za omogućavanjem daljnjeg proširivanja.




**Slika 5.2 Dijagram fizičke međuovisnosti datoteka zaglavlja u BaseInterfacesLib biblioteci**

Sama biblioteka BaseInterfacesLib je izgrađena u C++/CLI programskom jeziku (varijanta C++ s podrškom za *managed* objekte u .NET okruženju) i nju sačinjava skup C++ datoteka zaglavlja u kojima su dane deklaracije sadržanih razreda sučelja. Razredi su u pojedine datoteke grupirano po funkcionalnosti, a dijagram međuovisnosti datoteka je dan na slici 5.2.

#### 5.3.1.2.1 Razredi i sučelja povezani s definiranjem optimizacijskog problema

U konceptualnom modelu smo vezano uz optimizacijski problem definirali sljedećih nekoliko koncepata:

- IProblem – predstavlja općeniti optimizacijski problem (postoje i definirane specijalizacije: IProblemSO – jednokriterijski problem, IProblemMO – višekriterijski problem, IProblemWC – problem s definiranim ograničenjima)
- IProblemInput – predstavlja ulaz za optimizacijski problem, odnosno sadrži skup ulaznih varijabli problema
- IProblemInputDescriptor – sadrži podatke o tipovima varijabli nad kojima je definiran problem
- IProblemOutput – predstavlja izlaz optimizacijskog problema, odnosno sadrži skup vrijednosti koje su izračunate za zadane vrijednosti ulaznih varijabli (vrijednosti funkcije/a cilja i vrijednosti ograničenja). Postoje definirane četiri specijalizacije ovog koncepta (koje su dio BaseObjectsLib biblioteke): ProblemOutputSO, ProblemOutputSOWC, ProblemOutputMO, ProblemOutput-MOWC i koje predstavljaju izlaz odgovarajuće vrste optimizacijskog problema

Kako je ranije opisano, navedeni koncepti su reprezentirani razredima sučelja kod kojih je skup uloga i dogovornosti danog koncepta modeliran preko odgovarajućeg skupa čistih virtualnih funkcija.

**interface IProblem**

Razred sučelja koji predstavlja općeniti koncept optimizacijskog problema. Sve programske komponente koje predstavljaju optimizacijski problem zadovoljavaju ovo sučelje ili neku od njegovih specijalizacija.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO); | Izvedeni razredi će unutar svoje realizacije ove funkcije ugraditi programski kôd za izračunavanje izlaza problema na osnovu zadanog ulaza. |
| IProblemInputDescriptor^ getProblemInputDescription(); | Preko ove funkcije će objekt vratiti opis skupa ulaznih varijabli problema. Taj opis je dio IProblemInputDescriptor-a, i preko ove funkcije objekt problema daje referencu na taj pripadni objekt. |

**interface IProblemSO : IProblem**

Ovaj razred sučelja predstavlja specijalizaciju optimizacijskog problema koji ima definiran samo jedan kriterij optimalnosti (funkciju cilja). Uvodi se enumeracija EObjectiveType koja ima definirana dva člana: MAX i MIN, koji respektivno označavaju da se radi o problemu maksimizacije, odnosno minimizacije funkcije cilja.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| EObjectiveType getObjectiveType(); | Ova funkcija vraća cilj optimizacije (min / max) |
| String^ getObjectiveName(); | Vraća naziv funkcije cilja. |

**interface IProblemMO : IProblem**

Razred sučelja koji predstavlja višekriterijski optimizacijski problem.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| int getNumObjectives(); | Vraća broj funkcija cilja za dani višekriterijski problem |
| EObjectiveType getObjectiveType(int ObjFuncInd); | Vraća cilj optimizacije za svaku pojedinu funkciju cilja (max / min). |
| String^ getObjectiveName(int ObjFuncInd); | Isto kao gore. |

**interface IProblemWC : IProblem**

Razred sučelja koji predstavlja problem s ograničenjima. Bitno je naglasiti da su unutar ESOP okruženja sva ograničenja tipa g(x) <= 0, te je stoga važan samo podatak o broju ograničenja (njihove vrijednosti za dane vrijednosti ulaznih varijabli se izračunavaju pomoću CalcProblemOutput() funkcije).

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| int getNumConstraints(); | Vraća broj definiranih ograničenja u optimizacijskom problemu. |
| String^ getConstraintName(int i); | Vraća naziv odgovarajućeg ograničenja. Bitno je samo kod tekstualnog ispisivanja rezultata i nema veze s funkcionalnošću za optimizaciju. |

**interface IProblemInput**

Razred sučelja koji predstavlja skup ulaznih varijabli problema. Osnovna funkcionalnost razreda je vezana uz definiranje konkretnih tipova varijabli koje će predstavljati odabranu reprezentaciju ulaznih veličina optimizacijskog problema. Razred ima i podatak o opisu problema (IProblemInputDescriptor) kako bi prilikom specificiranja pojedine ulazne varijable korištenjem funkcije AssignDesignVariable() mogao provjeriti slaganje tipova.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| void setProblemInputDesc( IProblemInputDescriptor ^inPID); | Postavljanje IProblemInputDescritor objekta. |
| IProblemInputDescriptor^ getProblemInputDesc(); | Dohvaćanje IProblemInputDescriptor objekta. |
| void AssignDesignVariable( int DVIndex, IDesignVariable ^inDV); | Funkcija pomoću koje se za ulaznu veličinu problema (s indeksom DVIndex), postavlja konkretna vrsta objekta koji će reprezentirati tu veličinu / varijablu. |
| IDesignVariable^ GetDesignVariable(int DVIndex); | Dohvaćanje postavljene varijable |
| IProblemInput^ Clone(); | Kloniranje sadržaja cijelog objekta. |
| void GenerateRandom(Random ^rndGen); | Generiranje slučajnih vrijednosti za sve varijable ulaza. |

**interface IProblemInputDescriptor**

Razred sučelja koji opisuje tipove varijabli nad kojima je definiran optimizacijski problem. Tip svake varijable je opisan preko odgovarajućeg IDesignVariableDescriptor objekta. Kreiranje konkretnih objekata koji zadovoljavaju ovo sučelje i definiranje njihovog sadržaja obavlja optimizacijski problem.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| IProblemInputDescriptor^ Clone(); | Kloniranje sadržaja cijelog objekta. |
| int getNumDesignVar(); | Vraća broj ulaznih varijabli nad kojima je definiran optimizacijski problem. |
| void setNumDesignVar(int inNumDV); | Postavljanje broj ulaznih varijabli. |
| void setDesignVariableDesc(int DVIndex, IDesignVariableDescriptor ^inDVDesc); | Postavljanje Descriptor objekta za ulaznu varijablu. |
| IDesignVariableDescriptor^ getDesignVariableDesc(int DVIndex); | Dohvaćanje Descriptor objekta za ulaznu varijablu. |

**interface IProblemOutput**

Razred sučelja koji predstavlja izlaz optimizacijskog problema. Definira samo globalne odgovornosti, primjenjive na svaku vrstu problema, dok će konkretne odgovornosti biti definirane u izvedenim razredima sučelja.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| IProblemOutput^ Clone(); | Kloniranje sadržaja objekta. |
| String^ GetStringDesc(); | Funkcija vraća string reprezentaciju sadržaja objekta (odnosno, odgovarajućih izlaznih vrijednosti). |

**interface IProblemOutputSO : IProblemOutput**

Razred sučelja koji predstavlja izlaz jednokriterijskog optimizacijskog problema. Sadrži definiran skup funkcije za dohvaćanje i postavljanje izračunate vrijednosti funkcije cilja.

| *Definirana članska funkcija* | |
|---|---|
| void | setObjectiveValue(double inVal); |
| double | getObjectiveValue(); |

**interface IProblemOutputSOWC : IProblemOutputSO**

Razred sučelja koji predstavlja izlaz jednokriterijskog optimizacijskog problema s ograničenjima. Pored mogućnosti postavljanja i dohvaćanja izračunate vrijednosti funkcije cilja koju naslijeđuje iz IProblemOutputSO, definira skup funkcija za manipuliranje s izračunatim vrijednostima ograničenja.

| *Definirana članska funkcija* | |
|---|---|
| void | setNumConstraints(int inNumConstraints); |
| int | getNumConstraints(); |
| double | getConstraintValue(int Ind); |
| void | setConstraintValue(int Ind, double Val); |

**interface IProblemOutputMO : IProblemOutput**

Razred sučelja koji predstavlja izlaz višekriterijskog optimizacijskog problema (bez ograničenja). Za raliku od IProblemOutputSO sučelja, kod višekriterijskog problema se pojavljuje i broj funkcija cilja kao važan parametar, a kod operiranja s vrijednostima funkcija cilja je pomoću indeksa potrebno naznačiti o kojoj se funkciji cilja radi.

| *Definirana članska funkcija* | |
|---|---|
| void | setNumObjectives(int inNumObjectives); |
| int | getNumObjectives(); |
| double | getObjectiveValue(int Ind) |
| void | setObjectiveValue(int Ind, double Val); |

**interface IProblemOutputMOWC : IProblemOutputMO**

Razred sučelja koji predstavlja izlaz višekriterijskog optimizacijskog problema s ograničenjima. Definira skup funkcija koje rade s vrijednostima funkcija cilja, i skup funkcija koje rade s vrijednostima ograničenja.

| *Definirana članska funkcija* | |
|---|---|
| void | setNumConstraints(int inNumConstraints); |
| int | getNumConstraints(); |
| double | getConstraintValue(int Ind); |
| void | setConstraintValue(int Ind, double Val); |

#### 5.3.1.2.2 Razredi i sučelja za modeliranje varijabli problema

Osnovni koncept vezan uz modeliranje (ulaznih) varijabli problema je IDesignVariable. IDesignVariable je razred sučelja koji će naslijediti svaki konkretni razred kojim se modelira određena vrsta ulazne varijable problema. Kako različite vrste varijabli imaju različite karakteristike, unutar razreda IDesignVariable su definirane samo općenite odgovornosti primjenjive na svaku vrstu varijable, a karakteristike pojedinih varijabli su definirane u specijaliziranim razredima sučelja.

**interface IDesignVariableDescriptor**

Razred sučelja koji predstavlja opis ulazne varijable problema. Objekti ovog tipa će se koristiti unutar razreda IProblemInputDescriptor za opis karakteristika ulaznih varijabli optimizacijskog problema. Isto kao i za koncept ulazne varijable, i ovdje je definiran skup dodatnih razreda koji stvarno opisuju tražene karakteristike ulazne varijable problema (a koji je implementiran u biblioteci BaseObjectsLib)

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| IDesignVariableDescriptor^ Clone() ; | Kloniranje sadržaja objekta. |
| Type^ getType() = 0; | Preko ove funkcije će konkretni izvedeni objekti vratiti tip (u smislu .NET Type objekta) IDesignVariable sučelja kojeg dani descriptor opisuje. |

**interface IDesignVariable**

Razred sučelja koji predstavlja (jednu) varijablu optimizacijskog problema. S obzirom na različitost karakteristika pojedinih vrsta ulaznih varijabli problema, ovaj razred sučelja predstavlja osnovu iz kojega će se izvesti dodatni razredi sučelja koji će preciznije modelirati karakteristike pojedinih vrsta varijabli. Stoga je i skup definiranih članskih funkcija unutar ovog sučelja mali, te su one povezane s modeliranjem općenitih odgovornosti svake ulazne varijable problema.

| *Definirana članska funkcija* | *Opis modelirane odgovornosti* |
|---|---|
| IDesignVariable ^ Clone(); | Kloniranje objekta. |
| void GenerateRandom(Random ^rndGen); | Slučajno generiranje vrijednosti varijable. |
| String^ GetValueStringRepresentation(); | Izvedeni razredi će preko ove funkcije vraćati string reprezentaciju vrijednosti varijable |

**interface IBitArray : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu koja predstavlja polje bitova. Osnovna karakteristika varijable je broj bitova, a u sučelju su definirane i funkcije za dohvaćanje, odnosno postavljanje vrijednosti pojedinog bita.

| *Definirana članska funkcija* | |
|---|---|
| int | getBitNum(); |
| void | setBitNum(int inBitNum); |
| bool | getValue(int BitInd); |
| void | setValue(int BitInd, bool inVal); |

**interface IInteger : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu cjelobrojnog tipa. Bitne karakteristike varijable (koje su modelirane članskim funkcijama razreda sučelja) su interval dopuštenih vrijednosti, te sama vrijednost koju sadrži varijabla.

| *Definirana članska funkcija* | |
|---|---|
| void | getBounds(int ^Low, int ^Up); |
| void | setBounds(int Low, int Up); |
| int | getValue(); |
| void | setValue(int inVal); |

**interface IIntegerArray : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu koja predstavlja polje cjelobrojnih varijabli. Bitna karakteristika definiranog polja je da sadrži iste cjelobrojne varijable (odnosno, varijable istih karakteristika). Iz toga proizlazi da sve varijable imaju isti interval dopuštenih vrijednosti, a kod postavljanja odnosno dohvaćanja vrijednosti odgovarajuće varijable je potrebno navesti i njen indeks u polju.

| *Definirana članska funkcija* | |
|---|---|
| int | getNumVar(); |
| void | setNumvar(int inNumVar); |
| void | getBounds(int ^Low, int ^Up); |
| void | setBounds(int Low, int Up); |
| int | getValue(int VarInd); |
| void | setValue(int VarInd, int inVal); |

**interface IReal : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu realnog tipa. Po sadržaju je ovo sučelje slično (gotovo istovjetno) sa sučelje IInteger, s razlikom što se umjesto tipa podatka *int*, koristi *double*.

| *Definirana članska funkcija* | |
|---|---|
| void | getBounds(double ^Low, double ^Up); |
| void | setBounds(double Low, double Up); |
| double | getValue(); |
| void | setValue(double inVal), |

**interface IRealArray : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu koja predstavlja polje realnih brojeva. Sve navedeno za IIntegerArray vrijedi i za IRealArray, uz razliku u tipu podataka (*double* vs. *int*).

| *Definirana članska funkcija* | |
|---|---|
| int | getNumVar(); |
| void | setNumvar(int inNumVar); |
| void | getBounds(double ^Low, double ^Up); |
| void | setBounds(double Low, double Up); |
| double | getValue(int VarInd); |
| void | setValue(int VarInd, double inVal); |

**interface IPermutation : IDesignVariable**

Razred sučelja koji modelira ulaznu varijablu koja predstavlja matematičku strukturu permutaciju *N* elemenata. Sučelje definira mogućnost određivanja kardinalnosti pripadnog skupa permutacija, a osnovna funkcije je getNth() preko koje se dobija *n*-ti element permutacije.

| *Definirana članska funkcija* | |
|---|---|
| int | getNum(); |
| void | setNum(int inNum); |
| int | getNth(int i); |
| void | swap(int i, int j); |

### 5.3.1.2.3 Razredi i sučelja za modeliranje algoritama

Koncept iterativnog algoritma koji je u konceptualnom modelu predstavljen elementom IIterativeAlgorithm je programski realiziran definiranjem sučelja istog imena. To sučelje moraju zadovoljavati svi razredi koji implementiraju konkretni optimizacijski postupak. Kod definiranja koncepta iterativnog algoritma su definirane sljedeće njegove karakteristike / odgovornosti:

- mogućnost izvođenja jedne iteracije postupka
- mogućnost provođenja inicijalizacijskog koraka algoritma
- postojanje podatka (reference) o optimizacijskom problemu koji se rješava
- pružanje informaciju o postignutom rezultatu optimizacije

IIterativeAlgorithm razred sučelja ima centralnu ulogu u ESOP optimizacijskom okruženju i njegova deklaracija je sljedeća:

```
public interface class IIterativeAlgorithm
{
public:
        virtual   int       Initialize() = 0;
        virtual   int       PerformIteration() = 0;
        virtual void        Run() = 0;

        virtual   void      SetOptTerminator(IOptimizationTerminator^ inOptTerm) = 0;

        virtual void                SetProblem(IProblem ^inProblem) = 0;
        virtual IProblem^           GetProblem() = 0;

        virtual void        SetProblemInput(IProblemInput ^inPV) = 0;

        virtual void        AddObserver( IIntermediateTimer ^Timer,
                                         IObservedValueSaver ^Saver) = 0;

        virtual IOptimizationResult^            GetResultRef() = 0;
};
```

*Programski odsječak 2: Deklaracija razreda IIterativeAlgorithm*

Prve tri funkcije modeliraju osnovnu funkcionalnost iterativnog algoritma. Odnosno, izvedeni razredi koji će predstavljati konkretnu realizaciju određenog optimizacijskog postupka će definiranjem funkcija Initialize() i PerformIteration() ugraditi funkcionalnost specifičnu za taj algoritam.

Preko funkcije SetOptTerminator() se konkretnom objektu algoritma može postaviti odgovarajući objekt koji zadovoljava sučelje IOptimizationTerminator i koji će definirati trajanje optimizacije, a sličnu funkcionalnost pružaju i funkcije SetProblem() i SetProblemInput() pomoću kojih se postavljaju reference na objekt problema koji se rješava, odnosno na objekt koji predstavlja definirani skup ulazih varijabli problema.

Funkcionalnost vezana uz definiranje praćenih međurezultata je modelirana funkcijom AddObserver(). Pomoću te funkcije se objektu algoritma može postaviti objekt IObservedValueSaver koji će pratiti traženu vrstu međurezultata. S obzirom da je za realizaciju te funkcionalnosti vrlo važan i skup ugrađenih generičkih razreda u ESOP, detaljan opis te realizacije će se dati kod opisa BaseAlgorithmsLib biblioteke razreda.

Zadnja funkcija koja je dio sučelja IIterativeAlgorithm je funkcija GetResultsRef() preko koje se može dobiti referenca na IOptimizationResult objekt koji sadržava konačni rezultat optimizacije (zajedno s generiranim međurezultatima).

#### 5.3.1.2.4 Razredi i sučelja za modeliranje rezultata optimizacije

U prethodnom poglavlju smo identificirali koncept IOptimizationResult kao osnovni koncept koji predstavlja završni rezultat optimizacije. U skladu s tim je i definiran razred sučelja IOptimizationResult kojim je programski realiziran opisani

koncept. Kao što je već navedeno, vezano uz rezultat optimizacije imamo dva aspekta. Prvi je povezan s pojmom završnog rezultata optimizacije koji predstavlja konačno (najbolje) nađeno rješenje danog optimizacijskog problema, dok je drugi povezan s pojmom međurezultata koji su generirani tijekom samog provođenja optimizacije.

Koncept završnog rješenja je modeliran uvođenjem razreda sučelja IFinalResult, dok je za pohranjivanje međurezultata definiran razred sučelja IObservedValuesCollection. Međusobna interakcija definiranih razreda sučelja se najbolje vidi iz same deklaracije:

```
public interface class IObservedValue
{
        virtual String^     getStringDesc() = 0;
        virtual Type^       getType() = 0;
};
public interface class IObservedValueTag
{
        virtual int         getIterNum() = 0;
        virtual double      getTimeMs() = 0;
};
public interface class IObservedValuesCollection
{
public:
        property String^            Name;
        property int                NumValues  { int  get(); }
        property Type^              ValueType   { Type^ get(); }

        String^             GetStringRepr() = 0;
        void                Add(IObservedValue^   Value, IObservedValueTag^  inTag) = 0;
};
public interface class IFinalResult
{
        virtual IProblemOutput^   GetProblemOutputValues() = 0;
        virtual IProblemInput^    GetFoundInputValues() = 0;

        virtual String^             GetResultAsString() = 0;
};
public interface class IOptimizationResult
{
        virtual String^             GetResultAsString() = 0;

        virtual   void              SetFinalResult(IFinalResult^   inRes) = 0;
        virtual   IFinalResult^     GetFinalResult() = 0;

        virtual void        AddNewObsCollection(IObservedValuesCollection^  inColl) = 0;
};
```

*Programski odsječak 3: Deklaracija razreda sučelja povezanih s konceptom rezultata i međurezultata.*

Važno je napomenuti da će ovdje definirani razredi sučelja dobiti i svoje konkretne realizacije koje su ugrađene u biblioteku BaseResultsLib.

IOptimizationResult razred sučelja ima dvije osnovne uloge. Prva uloga je da ima podatak o konačnom rezultatu. Taj podatak se u objekt IOptimizationResult (odnosno izvedeni konkretni objekt koji će biti nazvan OptimizationResult koji je ugrađen u BaseResultsLib biblioteci) ugrađuje odnosno dohvaća pozivima funkcija

SetFinalResult() i GetFinalResult(). Druga uloga je da sadrži određeni skup kolekcija međurezultata. S obzirom da je opis funkcionalnosti vezane uz prikupljanje međurezultata opisan u jednom od sljedećih poglavlja (5.3.1.4. u kojem se opisuje BaseAlgorithmsLib biblioteka), ovdje je dovoljno pojasniti ulogu funkcije AddNewObsCollection(). Ta funkcija je, kao što joj i ime kaže, namijenjena dodavanju nove kolekcije međurezultata u listu međurezultata koju održava IOptimizationResults razred.

Važan je još i koncept same kolekcije predstavljene razredom sučelja IObservedValuesCollection. S obzirom da je jedan od ciljeva izgradnje funkcionalnosti vezane uz međurezultate bio postizavanje otvorenosti skupa različitih vrsti međurezultata koji se mogu prikupljati (odnosno, da se mogu definirati i nove vrste međurezultata), jasno je da izgrađeni objekt koji će predstavljati kolekciju međurezultata mora biti generički (odnosno, mora biti tako izgrađen da se može iskoristiti za pohranjivanje različitih tipova vrijednosti). Ta generičnost je postignuta definiranjem novog razreda sučelja IObservedValue, koji će biti bazni razred za sve vrste podataka (međurezultata) koji se žele pratiti. Uloga razreda sučelja IObservedValueTag je da opiše trenutak u kojem je prikupljen rezultat.

#### 5.3.1.2.5 Razredi i sučelja za modeliranje rješenja

U konceptualni model domene programske podrške za optimizaciju uveden je koncept rješenja (ISolution) kako bi postojao jedinstveni objekt nad kojim će operirati algoritmi. Kako je već ranije opisano, osnovni razlog je potreba algoritma da prilikom provođenja optimizacije zna podatke i o trenutnim vrijednostima ulaznih varijabli i o odgovarajućim (izračunatim) vrijednostima funkcija cilja (i ograničenja). Koncept rješenja je programski realiziran preko razreda sučelja ISolution a skup članskih funkcija preko koji su modelirane odgovornosti i karakteristike ISolution koncepta je sljedeći:

```
public interface class      ISolution
{
public:
        virtual ISolution^          Clone() = 0;
        virtual void                GenerateRandom(Random ^rndGen) = 0;
        virtual void                UpdateProblemOutput() = 0;
        virtual String^             GetStringRepr() = 0;

        virtual void                setProblem(IProblem ^inProblem) = 0;
        virtual IProblem^           getProblem() = 0;

        virtual void                setProblemInput(IProblemInput ^inPV) = 0;
        virtual IProblemInput^      getProblemInput() = 0;

        virtual IProblemOutput^     getProblemOutput() = 0;
};
```

*Programski odsječak 4: Deklaracija razreda sučelja ISolution.*

Osnovna funkcija u ovom sučelju je funkcija UpdateProblemOutput() čija je uloga ažuriranje sadržanih podataka o izlazu problema (IProblemOutput) u ovisnosti o postavljenim vrijednostima ulaznih varijabli problema (IProblemInput), koristeći pri tom referencu na optimizacijski problem koji se rješava (a koja je postavljena pozivom funkcije SetProblem()).

S obzirom da smo definirali različite vrste problem a koji koriste različita IProblemOutput sučelja za sadržavanje izračunatog izlaza, definirane su i tri specijalizacije ISolution razreda sučelja, koje su u skladu sa specijalizacijama razreda sučelja IProblem.

### interface ISolutionSO : ISolution

Razred sučelja koji predstavlja objekt rješenja kod optimizacije problema s jednom funkcijom cilja. Osnovna karakteristika tog objekta je da zna trenutno postignutu vrijednost funkcije cilja.

| *Definirana članska funkcija* | |
|---|---|
| double | getCurrObjectiveValue(); |

### interface ISolutionMO : ISolution

Razred sučelja koji predstavlja objekt rješenja kod višekriterijske optimizacije.

| *Definirana članska funkcija* | |
|---|---|
| int | getNumObjectives(); |
| double | getCurrObjectiveValue(int ObjIndex); |

### interface ISolutionWC : ISolution

Razred sučelja koji predstavlja objekt rješenja kod rješavanje optimizacijskog problema s ograničenjima. Iako ograničenja mogu imati i jednokriterijski i višekriterijski problemi, funkcionalnost potrebna za rukovanje ograničenjima je ista u oba slučaja, te je stoga definirano samo jedno sučelje koje će se kombinirati zajedno s ISolutionSO i ISolutionMO sučeljima kod definicije konkretnih objekata koji će predstavljati koncept rješenja optimizacijskog problema. Konkretni implementirani objekti vezani uz koncept rješenja su opisani u BaseObjectsLib biblioteci.

| *Definirana članska funkcija* | |
|---|---|
| int | getNumConstraints(); |
| int | getNumViolatedConstraints(); |
| double | getAmountViolatedConstraints(); |
| double | getCurrConstraintValue(int CIndex); |
| bool | isFeasible(); |

### interface ISolutionContainer

Pored zahtjeva za uvođenjem koncepta (jednog) rješenja, kod izgradnje pojedinih vrsta optimizacijskih algoritama se pokazuje i potreba za kreiranjem kolekcije (skupa) rješenja. Taj zahtjev je najizraženiji kod evolucijskih algoritama koji operiraju nad populacijom jedinki / rješenja, ali je prisutan i kod drugih vrsta optimizacije (npr. kod optimizacijskih postupaka baziranih na inteligenciji jata). Stoga je definirano sučelje ISolutionContainer koje predstavlja skup pojedinačnih rješenja. Definirano sučelje pomoću ugrađenih članskih funkcija modelira standardni

skup operacija pomoću kojih se pruža klasična funkcionalnost kolekcije podataka, i kojima nije potreban dodatan opis.

| *Definirana članska funkcija* | |
|---|---|
| void | Initialize(int inSize, ISolution ^inSol); |
| void | Randomize(Random ^rndGen); |
| void | Clear(); |
| String^ | GetStringRepr(); |
| int | getSize(); |
| ISolution^ | getSolution(int Index); |
| void | pushbackNewSolution(ISolution ^NewInd); |
| void | insertSolution(int Index, ISolution ^inInd); |
| void | replaceSolution(int IndToReplace, ISolution ^newIndividual); |
| void | removeSolution(int IndToRemove); |

#### 5.3.1.2.6 Razredi i sučelja za modeliranje operatora

Operatori predstavljaju građevne blokove heurističkih optimizacijskih postupaka unutar definiranog konceptualnog modela domene. Moglo bi se reći da je ESOP optimizacijsko okruženje postavljeno tako da se upravo u operatorima zbiva većina akcije povezane s provođenjem optimizacije.

U datoteci IOperators.h su definirana sučelja za određeni skup operatora koji je standardno ugrađen u ESOP okruženje. Međutim, taj skup nije zatvoren te ga je moguće proširiti definiranjem novih vrsta operatora. U nastavku odjeljka slijedi opis pojedinih razreda sučelje kojima su modelirani odgovarajući operatori.

**IFitnessAssignment operator**

Osnovna uloga ovog operatora je pridruživanje dobrote pojedinim jedinkama u populaciji jedinki nad kojom operira genetički algoritam. Deklaracija ovog razreda sučelja sadrži samo jednu člansku funkciju, kojom je modelirana odgovornost pridruživanja dobrote jedinki iz populacije. Ta deklaracija je sljedeća:

```
public interface class IFitnessAssignmentOperator : IOperator
{
public:
        virtual    void      Assign(  ISolutionContainer ^inPopulation,
                                      IProblem ^inProblem,
                                      array<double>  ^outFitness) = 0;
};
```

*Programski odsječak 5: Deklaracija sučelja za operatore pridruživanja dobrote.*

Razredi / objekti koji će predstavljati konkretne izgradnje operatora pridruživana dobrote (i čija će izgradnja biti opisana u potpoglavlju 5.3.2.) će ugraditi svoju definiciju ove funkcije. Značenje parametara ove funkcije je sljedeće:

- inPopulation – populacija jedinki (u stvari objekata rješenja) kojima se želi definirati dobrota

- inProblem – optimizacijski problem koji se rješava. Referenca na ovaj objekt je potrebna jer se prilikom pridruživanja dobrote mora znati kakav se optimizacijski problem rješava
- outFitness – izlazni parametar funkcije - polje realnih brojeva (s brojem elemenata koji je jednak broju jedinki u populaciji) koji predstavljaju izračunatu dobrotu odgovarajuće jedinke

Zadovoljavanjem ovog sučelja prilikom izgradnje konkretnih razreda operatora se omogućava njihovo iskorištavanje kod izgradnje različitih vrsta genetičkih algoritama u kojima se navedeni operator koristi, a definirano sučelje je dovoljno općenito da podržava izgradnju vrlo širokog skupa operatora pridruživanja dobrote.

### IFitnessBasedSelectionOperator

Ovim razredom sučelja je modeliran operator selekcije, prisutan kod svih evolucijskih algoritama (kako kod genetičkih algoritama, tako i kod evolucijskih strategija i genetičkog programiranja). Osnovna uloga operatora selekcije je da iz populacije jedinki selektira odgovarajući broj jedinki po nekom kriteriju, a bazirano na vrijednosti dobrote pojedinih jedinki. Ta odgovornost je modelirana članskom funkcijom Select(), a deklaracija razreda sučelja je sljedeća:

```
public interface class IFitnessBasedSelectionOperator : IOperator
{
public:
        virtual   void      Select(   array<double> ^vecSolFitness,
                                      array<int> ^vecOutSelInd,
                                      int inReqNumOfSelInd) = 0;
};
```

*Programski odsječak 6: Deklaracija razreda sučelja za operator selekcije*

Konkretna implementacija načina provođenja selekcije (npr. RouletteWheel ili StohasticUniversalSampling) je dio izvedenih razreda koji zadovoljavaju opisano sučelje. Parametri funkcije Select() su jasni: u polju vecSolFitness se predaje niz vrijednosti koje predstavljaju dobrotu svake pojedinačne jedinke i unutar funkcije je potrebno odabrati inReqNumOfSelInd jedinki i njihove indekse vratiti preko polja vecOutSelInd.

### ICrossoverOperator

Operator križanja predstavlja jedan od dva osnovna operatora kojima genetički algoritmi istražuju prostor rješenja. Njegova uloga je da na osnovu karakteristika postojećih jedinki kreira nove jedinke, koje će nastati križanjem karakteristika jedinki roditelja. Iako se mogu definirati i operatori križanja višeg reda (engl. *multi-parent multi-child*), uobičajeno je da se na osnovu dvije jedinke roditelja kreira jedno ili dvoje jedinki djece. U skladu s time su definirana i dva razreda sučelja sa sljedećom deklaracijom:

```
public interface class ICrossoverOperator2To1 : public IDesignVariableOperator
{
public:
        virtual   void Recombine( IDesignVariable ^Parent1,
                                  IDesignVariable ^Parent2,
                                  IDesignVariable ^pChild) = 0;
};
```

```
public interface class ICrossoverOperator2To2 : public IDesignVariableOperator
{
public:
	virtual	void Recombine( IDesignVariable^Parent1,
				IDesignVariable ^Parent2,
				IDesignVariable ^pChild1,
				IDesignVariable ^pChild2) = 0;
};
```

*Programski odsječak 7: Deklaracija operatora križanja*

Implementacija funkcije Recombine() u izvedenom razredu definira ponašanje operatora, odnosno način provođenja križanja dvije jedinki. Ovdje je bitno uočiti da izgradnja operatora križanja ovisi o tome nad kakvom vrstom varijabli operira. S obzirom da preko parametara funkcija prima jedinke kao IDesignVariable, unutar implementacije funkcije Recombine() će biti potrebno obaviti konverziju (engl. *cast*) na objekt odgovarajućeg tipa.

### IMutationOperator

Operator mutacije je drugi operator koji genetički algoritmi koriste za istraživanje prostora rješenja. Njegova uloga je da mutacijom postojeće jedinke proizvede novu jedinku koja će biti slična staroj, uz određenu (slučajnu) količinu različitosti. Deklaracija sučelja je sljedeća:

```
public interface class IMutationOp : public IDesignVariableOperator
{
public:
	virtual	void Mutate(IDesignVariable ^Child) = 0;
};
```

*Programski odsječak 8: Deklaracija operatora mutacije*

Uloga funkcije Mutate() je da nad primljenim objektom obavi mutaciju. Konkretan način obavljanja te mutacije se implementira u razredima izvedenim iz ovog sučelja.

### INeighbourhoodGeneratorOperator

Ovaj operator se koristi za generiranje novog rješenja iz okoline trenutnog rješenja. Operatori izvedeni iz ovog sučelja se koriste kod algoritama lokalnog pretraživanja i simuliranog kaljenja, a deklaracija razreda je sljedeća:

```
public interface class INeighbourhoodGeneratorOperator : public IDesignVariableOperator
{
public:
	void	GenerateNeighbour(IDesignVariable ^CurrSol, IDesignVariable ^GeneratedSol) = 0;
};
```

*Programski odsječak 9: Deklaracija operatora kreiranja susjednog rješenja*

Kod ugradnje funkcije GenerateNeighbour(), izvedeni razredi moraju ugraditi funkcionalnost za pronalaženje novog rješenja iz lokalne okoline rješenja predanog preko CurrSol ulaznog parametra.

### 5.3.1.3 BaseObjectsLib

Biblioteka razreda BaseInterfacesLib sadrži definirane razrede sučelja koji predstavljaju programske realizacije pojedinih koncepata iz razvijenog konceptualnog modela domene programske podrške za optimizaciju. Kako je već opisano, određeni broj navedenih koncepata ima i svoje generičke implementacije koje su uvijek iste i shodno tome neovisne o konkretnom optimizacijskom problemu i algoritmu iskorištenom za njegovo rješavanje.

S obzirom da postoji značajan broj takvih objekata, njihova ugradnja je razdvojena u tri biblioteke. Skup takvih objekata koji je povezan s konceptom rezultata je ugrađen je BaseResultsLib biblioteku, u biblioteci BaseAlgorithmsLib su ugrađeni konkretni razredi vezani uz koncept optimizacijskog algoritma dok su u biblioteci BaseObjectsLib ugrađeni objekti vezani uz koncepte problema, varijable problema i rješenja.

#### 5.3.1.3.1 Objekti povezani s konceptom problema

Vezano uz koncept optimizacijskog problema je definiran sljedeći skup sučelja: IProblem (+ SO, SOWC, MO i MOWC izvedenice), IProblemInput, IProblemInputDescriptor i IProblemOutput (+ SO, SOWC, MO i MOWC izvedenice). Iz navedene liste, samo je razred sučelja IProblem namijenjen proširivanju unutar ESOP-a, dok su realizacije konkretnih objekata izvedenih iz ostalih sučelja *generičke* (odnosno, jednom su definirane unutar ESOP-a te nema potrebe za definiranjem novih specijalizacija). Slijedi opis pojedinačnih realizacija.

***ProblemInputDescriptor***

Ovaj razred predstavlja realizaciju sučelja IProblemInputDescriptor. Osnovna karakteristika ovog razreda je da sadrži člansku varijablu koja je polje referenci na objekte tipa IDesignVariableDescriptor. Objekti referencirani u tom polju opisuju karakteristike ulaznih varijabli problema. Realizacija funkcija definiranih u sučelju IProblemInputDescriptor je jednostavna i svodi se na manipulaciju poljem _arrDesignVarDesc i njegovim sadržajem.

```
public ref class ProblemInputDescriptor : public IProblemInputDescriptor
{
public:
        ProblemInputDescriptor(int inNumDesignVar);

        virtual IProblemInputDescriptor^        Clone();
        virtual String^     GetStringDesc()   ;

        virtual int         getNumDesignVar();
        virtual void        setNumDesignVar(int inNum);

        virtual void        setDesignVariableDesc( int DVIndex,
                                                   IDesignVariableDescriptor ^inDVDesc);
        virtual IDesignVariableDescriptor^  getDesignVariableDesc(int DVIndex);

private:
        array<IDesignVariableDescriptor ^>    ^_arrDesignVarDesc;
};
```

*Programski odsječak 10: Deklaracija razreda ProblemInputDescriptor*

### *ProblemInput*

Ovaj razred predstavlja konkretnu programsku realizaciju sučelja IProblemInput. U skladu s definiranim odgovornostima, sadrži listu referenci na odabrane tipove ulaznih varijabli, a isto tako sadrži i referencu na objekt ProblemInputDescriptor.

Najvažnija članska funkcija je AssignDesignVariable() pomoću koje se definira konkretna vrsta objekta koji će se koristiti kao reprezentacija za i-tu ulaznu veličinu optimizacijskog problema. Važnost poznavanja odgovarajućeg IProblemInputDescriptor objekta proizlazi iz činjenice da se prilikom pridruživanja pomoću AssignDesignVariable() funkcije provjerava i slaganje tipova varijabli, odnosno, može li se predani konkretni objekt izveden iz IDesignVariable doista iskoristiti za reprezentiranje odgovarajuće ulazne veličine optimizacijskog problema.

```
public ref class ProblemInput : public IProblemInput
{
public:
        ProblemInput();
        ProblemInput(IProblemInputDescriptor ^inPID);

        virtual IProblemInput^      Clone();
        virtual void                GenerateRandom(Random ^rndGen);
        virtual String^             GetStringDesc()  ;

        virtual void                            setProblemInputDesc(IProblemInputDescriptor ^inPID);
        virtual IProblemInputDescriptor^    getProblemInputDesc();

        virtual void                AssignDesignVariable(int DVIndex, IDesignVariable ^inDV);
        virtual  IDesignVariable^   GetDesignVariable(int DVIndex);

private:
        IProblemInputDescriptor^        _refPIDesc;
        array<IDesignVariable ^> ^      _arrDesignVariables;
};
```

*Programski odsječak 11: Deklaracija razreda ProblemInput*

### *ProblemOutput razredi*

Izgrađeni ProblemOutput razredi predstavljaju generičke realizacije odgovarajućih IProblemOutput razreda sučelja i njihova implementacija direktno proizlazi iz tih sučelja. Primarna razlika je u tome što ovdje opisani konkretni razredi imaju definirane članske varijable koje sadrže podatke o izračunatom izlazu optimizacijskog problema.

S obzirom da su deklaracije konkretnih razreda uglavnom istovjetne s deklaracijama odgovarajućih razreda sučelja (osim što su naravno čiste virtualne funkcije sada postaje obične članske funkcije s implementacijom), navest će se kompletna deklaracija samo za jedan od četiri izgrađena razreda, a za ostale će se prikazati samo novouvedeni detalji izgradnje. Deklaracije izgrađenih konkretnih razreda su dane u sljedećem programskom odsječku:

```
public ref class ProblemOutputSO : public IProblemOutputSO
{
public:
        ProblemOutputSO();
        ProblemOutputSO(ProblemOutputSO ^copy);

        // skup funkcija koje realiziraju definirano sučelje
        virtual IProblemOutput^   Clone();
        virtual   String^                       GetStringDesc();

        void              setObjectiveValue(double inVal)
        double            getObjectiveValue();

private:
        double            _ObjectiveValue;
};

public ref class ProblemOutputSOWC : public IProblemOutputSOWC
{
// ... dio razreda koji predstavlja implementaciju sučelja
private:
        double                _ObjectiveValue;
        array<double> ^   _arrConstraintValues;
};

public ref class ProblemOutputMO : public IProblemOutputMO
{
// ... dio razreda koji predstavlja implementaciju sučelja
private:
        array<double> ^    _arrObjectiveValues;
};

public ref class ProblemOutputMOWC : public IProblemOutputMOWC
{
// ... dio razreda koji predstavlja implementaciju sučelja
private:
        array<double> ^    _arrObjectiveValues;
        array<double> ^    _arrConstraintValues;
};
```

*Programski odsječak 12: Deklaracija ProblemOutput razreda*

Iz navedenih deklaracija vidimo da je kod svake konkretne realizacije, u razred dodan odgovarajući skup podataka koji predstavlja podatke generirane kod izračunavanja izlaza optimizacijskog problema. Bitno je napomenuti da svi izgrađeni razredi i dalje zadovoljavaju osnovno sučelje IProblemOutput te se s njima u ostatku aplikacije može manipulirati preko reference na to sučelje, bez obzira na to o kakvom se konkretno objektu radi.

## *Apstraktno definirani razredi problema*

Kako je već navedeno u uvodu u ovaj odjeljak, razred sučelja IProblem (i njegove specijalizacije IProblemSO, IProblemMO, IProblemWC) je namijenjem daljnjem proširivanju u ESOP razvojnom okruženju. Odnosno, konkretni razredi izvedeni iz ovog sučelja će predstavljati pojedinačne konkretne optimizacijske probleme. Međutim, kod izgradnje tih konkretnih razreda se uočava određeni skup

funkcionalnosti koji je generički, odnosno isti za sve razrede / objekte. Kako bi se izbjeglo da se u svakom konkretnom razredu mora iznova ugrađivati ta funkcionalnost, definiran je skup apstraktnih razreda koji zadovoljavaju odgovarajuća sučelja izvedena iz IProblem i koji imaju generičku ugradnju dane funkcionalnosti. Ti apstraktni razredi sada predstavljaju osnovu za daljnje naslijeđivanje.

Te zajedničke karakteristike su vezane uz definiranje broja i vrste (min/max) funkcija cilja, te broja ograničenja u problemu. Kratkom analizom se mogu utvrditi sljedeće činjenice:

- jednokriterijski optimizacijski problemi moraju definirati da li se radi o maksimizaciji ili minimizaciji funkcije cilja
- višekriterijski problemi moraju definirati broj funkcija cilja, i vrstu svake pojedinačne funkcije cilja (min/max)
- problemi s ograničenjima moraju definirati broj ograničenja (s obzirom da je osnovna pretpostavka da su sva ograničenja tipa g(x) <= 0)
- postoje jednokriterijski problemi s ograničenjima i višekriterijski problemi s ograničenjima

Stoga su izgrađena četiri apstraktna razreda od kojih svaki predstavlja određenu vrstu optimizacijskog problema. Sljede najvažniji dijelovi njihovih deklaracija:

```
public ref class ProblemSO abstract : public IProblemSO
{
public:
        ProblemSO(EObjectiveType inObjType);

        virtual   EObjectiveType    getObjectiveType();
protected:
        EObjectiveType              _enProblemType;
};

public ref class ProblemSOWC abstract : public IProblemWC
{
public:
        ProblemSOWC(EObjectiveType inObjType, int inNumConstraints)

        virtual   int               getNumConstraints()
        virtual    EObjectiveType   getObjectiveType();
protected:
        int                         _NumConstraints;
        EObjectiveType              _enProblemType;
};
public ref class ProblemMO abstract : public IProblemMO
{
public:
        ProblemMO(int inNumObjectives);

        virtual   int                 getNumObjectives();
        void                          setObjectiveType(int ObjFuncInd, EObjectiveType inType);
        virtual    EObjectiveType     getObjectiveType(int ObjFuncInd);
protected:
        array<EObjectiveType>^    _enProblemType;
};
```

```
public ref class ProblemMOWC abstract : public IProblemMO, IProblemWC
{
public:
        ProblemMOWC(int inNumObjectives, int inNumConstraints);

        virtual  int                getNumObjectives();
        void                        setObjectiveType(int ObjFuncInd, EObjectiveType inType);
        virtual  EObjectiveType     getObjectiveType(int ObjFuncInd);

        virtual  int                getNumConstraints();
protected:
        int                         _NumConstraints;
        array<EObjectiveType>^      _enProblemType;
};
```

*Programski odsječak 13: Deklaracija apstraktnih Problem razreda*

Izgradnjom ovih apstraktnih razreda je značajno pojednostavljena ugradnja konkretnih optimizacijskih problema. Sve što je potrebno ugraditi je implementacija funkcije CalcProblemOutput(), u kojoj će se definirati način izračunavanja vrijednosti funkcija cilja i ograničenja, i funkcije getProblemInputDescriptor() koja vraća ProblemInputDescriptor objekt u kojem je su opisane karakteristike skupa varijabli nad kojima je definiran problem. Iako te funkcije nisu navedene u gornjoj deklaraciji, one su u opisanim razredima prisutne implicitno, s obzirom da su naslijeđene iz baznih razreda sučelja.

#### 5.3.1.3.2 Objekti povezani s konceptom ulazne varijable

Prilikom razmatranja razreda sučelja koji su povezani s konceptom ulazne varijable problema, definirali smo dva osnovna razreda sučelja: IDesignVariable i IDesignVariableDescriptor. IDesignVariable predstavlja sučelje prema konkretnim razredima koji će predstavljati tipove ulaznih varijabli problema, dok IDesignVariableDescriptor predstavlja sučelje prema razredima koji opisuju karakteristike pojedinih vrsta ulaznih varijabli.

S obzirom da je unutar ESOP optimizacijskog okruženja standardno ugrađen određen broj vrsta ulaznih varijabli (Real, Integer, RealArray, Permutation) koje zadovoljavaju sučelja izvedena iz IDesignVariable (IReal, IInteger, IPermutation, ...), pokazalo se potrebno ugraditi i odgovarajuće objekte izvedene iz IDesignVariableDescriptor sučelja koji će opisivati karakteristike tih standardno ugrađenih varijabli. Kako je već opisano kod opisa koncepta IDesignVariableDescriptor, uloga svakog *descriptor* razreda je da potpuno opiše svojstva pripadne vrste varijable. Detalji implementacije su vidljivi na primjeru RealArray_DVDescription objekta čija je deklaracija sljedeća:

```
public ref class RealArray_DVDescription : public IDesignVariableDescriptor
{
public:
        RealArray_DVDescription(int inNumElem, double inLow, double inUp)
        virtual             IDesignVariableDescriptor^          Clone(),

        virtual String^     getName() { return "Real"; }
        virtual Type^       getType() { return IRealArray::typeid; }
```

```
        // ... skup get/set funkcija
private:
        int     _NumElem;
        double  _Low, _Up;
};
```

*Programski odsječak 14: Deklaracija RealArray_DVDescription razreda*

Pored ovog razreda su još definirani i sljedeći razredi za opis varijabli:

- BitArray_DVDescription – opisuje BitArray varijablu. Osnovni podatak je broj bitova.
- Integer_DVDescription – opisuje cjelobrojnu varijablu. Sadrži podatke o donjoj i gornjoj granici raspona dopuštenih vrijednosti.
- Real_DVDescription – opisuje realnu varijablu. Sadrži podatke o donjoj i gornjoj granici raspona dopuštenih vrijednosti.
- IntegerArray_DVDescription – opisuje karakteristike polja cjelobrojnih varijabli. Sadrži podatak o broju elemenata u polju i raspon dopuštenih vrijednosti.
- Permutation_DVDescription – opisuje karakteristike varijable koja predstavlja permutaciju. Sadrži podatak o broju elemenata skupa koji se premutiraju.

#### 5.3.1.3.3 Razredi za realizaciju sučelja ISolution

Skup izgrađenih konkretnih razreda koji zadovoljavaju sučelja vezana uz ISolution koncept predstavlja završni dio BaseObjectsLib biblioteke. Osnovna uloga ISolution koncepta je da predstavlja miješani (*mix-in*) razred koji će kombinirati tri koncepta: sam optimizacijski problem, ulaz problema i izračunati izlaz za dani ulaz. Kako je već opisano, njegova uloga je da posluži optimizacijskim algoritmima kao osnovni objekt nad kojim će oni operirati.

S obzirom na generičnost opisanog koncepta rješenja, koji ne ovisi o konkretnom optimizacijskom problemu niti o njegovom skupu ulaznih varijabli, već samo o klasi problema (SO, SOWC, MO, MOWC), izgrađena su 4 konkretna razreda: SolutionSO, SolutionSOWC, SolutionMO, SolutionMOWC. S obzirom na sličnost izgradnje za sva četiri navedena razreda, o ovdje će se prikazati detalji izgradnje samo za jedan izgrađeni razred.

```
public ref class SolutionSO : public ISolutionSO
{
public:
        SolutionSO();

        // funkcije iz sučelja
        virtual ISolution^ Clone()  override;
        virtual void                UpdateProblemOutput()    override;
        virtual String^             GetStringRepr()   override;

        virtual void                setProblem(IProblem ^inProblem)  override;
        virtual IProblem^           getProblem() override;
        virtual IProblemOutput^     getProblemOutput() override;

        // forwarding funkcija za jednostavnije korištenje
        virtual    double           getCurrObjectiveValue();
```

```
private:
        ProblemSO               ^_refProblem;
        ProblemInput            ^_refInputVal;
        ProblemOutputSO         ^_OutputVal;
};
void    SolutionSO::UpdateProblemOutput()
{
        if( _refProblem != nullptr )  {
                if( _refInputVal != nullptr )
                        _refProblem->CalcProblemOutput(_refInputVal, _OutputVal);
                else
                        throw gcnew Solution_ProblemInput_NotSet();
        }
        else
                throw gcnew Solution_Problem_NotSet();
}
```

*Programski odsječak 15: Deklaracija SolutionSO razreda*

Unutar razreda SolutionSO su deklarirane tri reference na odgovarajuće objekte problema (ProblemSO), ulaznih varijabli (ProblemInput) i izračunatih izlaza (ProblemOutputSO) koji se mogu dohvatiti preko definiranih članskih funkcija. Ostali izgrađeni razredi će, jasno, koristiti odgovarajuće objekte (SOWC, MO, MOWC). Najvažniji dio izgrađenih razreda čini funkcija UpdateProblemOutput() koja koristeći sadržane objekte ažurira vrijednosti izlaza u skladu s vrijednostima ulaza.

Zanimljivo je još i pogledati realizaciju SolutionMOWC razreda:

```
public ref class SolutionMOWC : public ISolutionMO, ISolutionWC
{
public:
        SolutionMOWC();

        virtual ISolution^          Clone()  override;
        virtual void                UpdateProblemOutput()   override;
        virtual String^             GetStringRepr()   override;

        virtual void                setProblem(IProblem ^inProblem) override;
        virtual IProblem^           getProblem() override;
        virtual IProblemOutput^     getProblemOutput() override;

        virtual int         getNumObjectives();
        virtual double      getCurrObjectiveValue(int ObjInd);

        virtual int         getNumConstraints() ;
        virtual double      getCurrConstraintValue(int ConstrInd);

        virtual int         getNumViolatedConstraints();
        virtual double      getAmountViolatedConstraints();
        virtual bool        isFeasible();
private:
        ProblemMOWC                     ^_refProblem;
        ProblemInput                    ^_refInputVal;
        ProblemOutputMOWC               ^_OutputVal;
}
```

*Programski odsječak 16: Deklaracija SolutionMOWC razreda*

Pažnju je potrebno obratiti na činjenicu da izgradnja MOWC specijalizacije razreda rješenja zadovoljava dva definirana sučelja. Od sučelja ISolutionMO se preuzimaju aspekti vezani uz činjenicu da dani razred predstavlja rješenje višekriterijskog problema, dok zadovoljavanjem sučelja ISolutionWC modeliramo činjenicu da postoje definirana ograničenja nad zadanim problemom. Isti princip je primjenjen i kod izgradnje razreda SolutionSOWC koji je izveden iz sučelja ISolutionSO i ISolutionWC.

### 5.3.1.4 BaseAlgorithmsLib

Razred sučelja IIterativeAlgorithm, implementiran u biblioteci BaseInterfaces-Lib, definira sučelje koje moraju zadovoljavati svi razredi / objekti koji predstavljaju optimizacijski algoritam unutar ESOP okruženja. Međutim, određeni dio funkcionalnosti modelirane članskim funkcijama razreda sučelja je generički, odnosno isti za sve izgrađene optimizacijske algoritme. To se poglavito odnosi na funkcionalnost vezanu uz rukovanje rezultatima i međurezultatima te upravljanje trajanjem optimizacije.

Stoga je unutar ove biblioteke ugrađen apstraktni razred IterativeAlgorithm, koji zadovoljava zadano sučelje IIterativeAlgortihm. Izgrađeni razred je apstraktan zato što ne definira sve tražene dijelove sučelja, već njihovu implementaciju delegira na konkretne razrede koji će biti izvedeni iz ovog apstraktnog razreda. Odnosno, razred IterativeAlgorithm ugrađuje dio funkcionalnosti vezan uz rukovanje rezultatima i međurezultatima i u stvari predstavlja osnovni razred iz kojega će se izvoditi konkretni razredi koji će predstavljati pojedine optimizacijske postupke. S obzirom na tranzitivni karakter relacije nasljeđivanja, svi razredi izveden iz apstraktnog razreda IterativeAlgorithm automatski nasljeđuju i sučelje IIterativeAlgorithm.

Deklaracija razreda je sljedeća:

```
public ref class IterativeAlgorithm abstract : public IIterativeAlgorithm,
                                               public IOptTermIterNumProvider,
                                               public IOptTermDurationProvider
{
public:
        IterativeAlgorithm();

        virtual    int          Initialize() = 0;
        virtual    int          PerformIteration() = 0;
        virtual void            Run();

        virtual void            SetProblem(IProblem ^inProblem) = 0;
        virtual IProblem^       GetProblem() = 0;
        virtual void            SetProblemInput(IProblemInput ^inPV) = 0;

        virtual    void         SetOptTerminator(IOptimizationTerminator^ inOptTerm);

        virtual IOptimizationResult^          GetResultRef()  { return _Result; }

        virtual int             getCurrIterNum();
        virtual double          getCurrTimeSpanInMs();

        virtual void            AddObserver(IIntermediateTimer ^inTimer,
                                            IObservedValueSaver ^inSaver);
protected:
        IntermediateValueMasterObserver^          getMasterObserver();
```

```
        int                 _CurrIterNum;
        DateTime            _OptStartTime;

        IOptimizationTerminator             ^_OptTerm;
        IntermediateValueMasterObserver     ^_IntermedResObserver;

        OptimizationResult          ^_Result;
};
```

*Programski odsječak 17: Deklaracija apstraktnog razreda IterativeAlgorithm*

Iz gornje deklaracije je vidljiv skup članskih funkcija koje nisu implementirane u ovom razredu. To su funkcije koje predstavljaju srž optimizacijskog algoritma (Initialize() i PerformIteration()) i skup funkcija vezan uz definiranje optimizacijskog problema koji se rješava i definiciju skupa ulaznih varijabli.

#### 5.3.1.4.1 Opis ugrađene funkcionalnosti

Upravljanje algoritmom je poglavito vezano uz određivanje slijeda iteracija i određivanje trajanja optimizacije. Postavljena arhitektura jasno definira slijed operacija kod provođenja optimizacije. Najprije se poziva funkcija Initialize() koja je namijenjena obavljanju inicijalizacije objekta algoritma, nakon čega se obavlja određeni broj iteracija algoritma pozivom funkcije PerformIteration(). Ponovno je potrebno naglasiti da je definicija ovih funkcija dio izvedenog razreda koji predstavlja realizaciju konkretnog optimizacijskog postupka.

Međutim, sam opisani slijed operacija je isti za sve algoritme (ili preciznije rečeno, takav slijed operacija je preduvjet da bi se optimizacijski postupak uopće mogao ugraditi u ESOP okruženje). Stoga je unutar razreda IterativeAlgorithm ugrađena implementacija funkcije Run() koja predstavlja osnovnu funkciju koju se poziva nad danim objektom algoritma prilikom provođenja optimizacije.

Definiranjem funkcije Run() se unutar razreda IterativeAlgorithm omogućava ugradnja funkcionalnosti za određivanje trajanja optimizacije te za prikupljanje međurezultata. Radi jasnoće opisa ugradnje ove funkcionalnosti, u sljedećem programskom odsječku je dana implementacija funkcije Run:

```
void    IterativeAlgorithm::Run()
{
        Initialize();
        _CurrIterNum = 0;
        _OptStartTime = DateTime::Now;
        while( _OptTerm->IsTermConditionSatisfied(this) == false ) {
                PerformIteration();

                IObservedValueTag    ^tag = gcnew ObservedValueTag(getCurrIterNum(),
                                                                   getCurrTimeSpanInMs());
                _IntermedResObserver->Check(tag);
                _CurrIterNum++;
        }

        _Result->TotalIterNum = getCurrIterNum();
        _Result->DurationMs    = getCurrTimeSpanInMs();
}
```

### Trajanje optimizacije

Za određivanje trajanja optimizacije je definiran IOptimizationTerminator razred sučelja koji od izvedenih konkretnih razreda zahtijeva implementaciju funkcije IsTermConditionSatisfied() koja definira kriterij zaustavljanja optimizacije. Kreiranje konkretnog IOptimizationTerminator objekta se obavlja izvan objekta algoritma i kreirani objekt se predaje algoritmu preko funkcije SetOptimization-Terminator() i pohranjuje kao referenca u _OptTerm člansku varijablu.

Definirana su dva konkretna razreda koji zadovoljavaju sučelje IOptimizationTerminator: TerminateOnIterationNum i TerminateOnDuration. Prvi definira trajanje optimizacije kroz zadani broj iteracija, dok je kod drugoga kriterij vremensko trajanje optimizacije (u milisekundama). Ovo su dva osnovna načina definiranja trajanja optimizacije, ali je unutar ESOP okruženja moguće ugraditi i druge, npr. definiranje trajanja optimizacije preko broja evaluacija funkcija cilja.

### Prikupljanje međurezultata

Prikupljanje međurezultata je kod izgradnje razreda IterativeAlgorithm delegirano na drugi razred – IntermediateValueMasterObserver (skraćeno IVMO). Razred IterativeAlgortihm sadrži referencu na objekt tipa IVMO a jedini zadatak objekta algoritma je da kreira vremensku oznaku (engl. *time-tag*) trenutka u kojem se prikuplja međurezultat i pozove funkcije Check nad referenciranim IVMO objektom.

Iz programskog odsječka 19 se vidi kako je realizirana opisana funkcionalnost.

```
public ref class IntermediateValueMasterObserver
{
public:
        // konstruktori i destruktori

        void            AddObserver(IntermediateValueObserver ^inObs);

        void            Check(IObservedValueTag ^inTag);
        void            UpdateTimers();
public:
        List<IntermediateValueObserver ^>                   ^_listObs;
};
void    IntermediateValueMasterObserver::Check(IObservedValueTag ^inTag)
{
        int Count = _listObs->Count;

        for(int i=0; i<Count; i++ )
        {
                IIntermediateTimer^ timer = _listObs[i]->getTimer();
                if( timer->IsTimeToSave(inTag) )
                        _listObs[i]->getSaver()->Save(inTag);
        }
}
```

*Programski odsječak 19: Deklaracija razreda IntermediateValueMasterObserver*

Pozivom funkcije Check(), IVMO objekt će proći kroz listu definiranih IntermediateValueObserver objekata i za svakoga provjeriti da li je vrijeme za prikupljanje međurezultata preko poziva IsTimeToSave() i ako jest, obaviti snimanje danog međurezultata pozivom funkcije Save().

Funkcionalnost IntermediateValueObserver objekta je dvojaka: on mora imati ugrađen mehanizam za određivanje trenutka u kojem će se prikupljati međurezultati (npr. želimo ih prikupljati svakih 10 iteracija) i mora znati vrstu međurezultata koja se prikuplja. Iz deklaracije razreda IntermediateValueObserver:

```
public ref class IntermediateValueObserver
{
public:
        void            setTimer(IIntermediateTimer ^inTimer);
        void            setSaver(IObservedValueSaver ^inSaver);

        IIntermediateTimer^             getTimer();
        IObservedValueSaver^            getSaver();

private:
        IIntermediateTimer              ^_Timer;
        IObservedValueSaver             ^_Saver;
};
```

*Programski odsječak 20: Deklaracija razreda IntermediateValueObserver.*

vidimo način ugradnje te funkcionalnosti. Referencirani objekt tip IItermediateTimer ima ugrađenu funkcionalnost za određivanje trenutka prikupljanja međurezultata. IIntermediateTimer je razred sučelja, kao što se vidi i iz njegove deklaracije:

```
public interface class IIntermediateTimer
{
public:
        virtual void        SetIntervalParameter(int par) = 0;
        virtual  bool       IsTimeToSave(IObservedValueTag ^inTag) = 0;
};
```

*Programski odsječak 21: Deklaracija razreda sučelja IIntermediateTimer*

Konkretno su definirana dva objekta koji zadovoljavaju dano sučelje: IntermedTimerOnEveryIterNum koji omogućava prikupljanje međurezultata svakih *n* iteracija i IntermedTimerOnTimeInterval razred koji omogućava prikupljanje međurezultata svakih *m* milisekundi (gdje su *n* i *m* parametri koji se postavljaju nad instanciranim objektima ovih razreda). IntermediateValueObserver se stoga oslanja na funkciju IsTimeToSave() za određivanje trenutka kada će se obaviti prikupljanje međurezultata.

Samo prikupljanje međurezultata se obavlja preko razreda sučelja IObservedValueSaver. Njegova deklaracija je sljedeća:

```
public interface class IObservedValueSaver
{
public:
        virtual String^     getObserverDesc() = 0;

        virtual void        setCollection(IObservedValuesCollection ^inCollToSave) = 0;
        virtual void        setProvider(IIntermediateValueProvider ^provider) = 0;

        virtual void        Save(IObservedValueTag ^inTag) = 0;
};
```

*Programski odsječak 22: Deklaracija razreda sučelja IObservedValueSaver.*

Ponovno je važno naglasiti ulogu IObservedValueSaver kao razreda sučelja. Ovaj razred samo definira sučelje prema konkretnim izgrađenim razredima koji imaju ugrađenu funkcionalnost za prikupljanje pojedine vrste međurezultata. Tako je ugrađen konkretni razred CurrObjectiveValueSaver koji omogućava prikupljanje podatka o trenutnoj vrijednosti funkcije cilja tijekom provođenja optimizacije. Drugi primjer je razred AverageFitnessValueInPopulation koji omogućava prikupljanje podatka o prosječnoj vrijednosti fitnessa u populaciji rješenja kod genetičkih algoritama.

Osnovna je funkcija Save() koju IntermediateValueObserver poziva nad referenciranim IObservedValueSaver objektom u trenutku kada je potrebno prikupiti međurezultate. Pitanje je odakle dolaze ti međurezultati u konkretan objekt izveden iz IObservedValeuSaver razreda? Odgovor leži u IIntermediateValueProvider razredu sučelja. Svaki promatrač (engl. *observer*) ima svog davatelja rezultata (engl. *provider*) koji mu se postavlja pozivom funkcije setProvider() prilikom čega se obavlja *down-cast* na konkretnu vrstu objekta koju observer očekuje.

Interakcija među definiranim objektima se najbolje može ilustrirati na primjeru CurrObjectiveValueSaver razreda koji očekuje da mu se preda referenca na IIntermedValueProvider_CurrentObjectiveValue provider-a međurezultata. Njegova deklaracija, sa relevantnim implementacijama je sljedeća:

```
public ref class CurrObjectiveValueSaver : public IObservedValueSaver
{
public:
        CurrObjectiveValueSaver(  IObservedValuesCollection ^inCollToSave,
                                  IIntermedValueProvider_CurrentObjectiveValue ^provider)  {
                _ValCollection = inCollToSave;
                _Provider = provider;
        }

        // ...

        virtual    void      Save(IObservedValueTag ^inTag);
        {
           double a = _Provider->getCurrObjectiveValue();

           RealObsValue          ^newVal = gcnew RealObsValue();
           newVal->_Value = a;

           _ValCollection->Add(newVal, inTag);
        }

        virtual void         setCollection(IObservedValuesCollection ^inCollToSave)
        {
             _ValCollection = inCollToSave;
        }
        virtual void         setProvider(IIntermediateValueProvider ^provider)
        {
            _Provider = dynamic_cast<IIntermedValueProvider_CurrentObjectiveValue ^> (provider);
        }

private:
        IIntermedValueProvider_CurrentObjectiveValue        ^_Provider;
        IObservedValuesCollection                           ^_ValCollection;
};
```

```
public interface class IIntermedValueProvider_CurrentObjectiveValue :
                                                    IIntermediateValueProvider
{
public:
        virtual   double     getCurrObjectiveValue() = 0;
};
```

*Programski odsječak 23: Prikaz ugradnje CurrObjectiveValueSaver razreda*

U ovom programskom odsječku se trebaju dodatno pojasniti dva detalja. Prvi je vezan uz sam koncept davatelja rezultata međurezultata. Naime, IIntermedValueProvider_CurrentObjectiveValue predstavlja također razred sučelja koji nad izvedene razrede postavlja zahtjev da implementiraju definirane funkcije (odnosno, u ovom konkretnom slučaju, funkciju getCurrObjectiveValue()). Pitanje je koji je konkretni objekt koji zadovoljava to sučelje ? Odgovor se nameće sam po sebi – onaj koji ima tražene podatke – a to je objekt algoritma. Stoga je za potpuno razumijevanje interakcije poteebno dati i prikaz deklaracije jednog konkretnog razreda algoritma.

```
public ref class SO_GenerationalGA :
                        public IterativeAlgorithm,
                        public IIntermedValueProvider_CurrentObjectiveValue,
                        public IIntermedValueProvider_CurrentAverageObjective
                        public IIntermedValueProvider_BestFitnessInCurrPopulation,
                        public IIntermedValueProvider_AverageFitnessInCurrPopulation,
                        public IIntermedValueProvider_WorstFitnessInCurrPopulation,

{
public:
        virtual   double          getCurrAverageObjectiveValue()  { ... implementacija }
        virtual   double          getCurrObjectiveValue()  { ... implementacija }
        virtual   double          getCurrBestFitnessValue()  { ... implementacija }
        virtual   double          getCurrAverageFitnessValue()  { ... implementacija }
        virtual   double          getCurrWorstFitnessValue()  { ... implementacija }

        // ... ostatak razreda vezan uz ugradnju funkcionalnosti genetičkih algoritama
```

*Programski odsječak 24: Prikaz interakcije razreda algoritma s razredima za praćenje međurezultata*

Odnosno, prilikom same izgradnje optimizacijskog algoritma (odnosno pripadnog razreda izvedenog iz IterativeAlgorithm-a) se odabirom podržanih IIntermedValueProvider sučelja definira kakvi će se međurezultati moći prikupljati tijekom optimizacije. Svako od definiranih *provider* sučelja ima svog odgovarajućeg *ObservedValueSaver*-a.

Završno pitanje na koje je potrebno odgovoriti je – gdje se pohranjuju prikupljeni međurezultati? Iz danih programskih odsječaka vidimo da konkretni ObserverSaver objekti imaju referencu na IObservedValueCollection objekt. Pitanje je gdje se i od strane koga kreira ta kolekcija? Kako je jedna od arhitekturnih postavki ESOP okruženja centralna uloga IOptimizationResult koncepta/razreda koji predstavlja cjelokupni rezultat optimizacije, logično je da taj razred preuzme odgovornost za upravljanje kolekcijom.

Sljedeći programski odsječak je ilustrativan:

```
SO_GenerationalGA              ^Alg = gcnew SO_GenerationalGA(inRnd);

// ... definiranje cjelokupnog konteksta algoritma

IObservedValuesCollection  ^newColl = gcnew ObservedValuesCollection("Curr.obje.value");
Alg->GetResultRef()->AddNewObsCollection(newColl);

CurrObjectiveValueSaver   ^newSaver = gcnew CurrObjectiveValueSaver(newColl, Alg);
Alg->AddObserver(gcnew IntermedTimerOnEveryIterNum(10), newSaver);

Alg->Run();     // pokretanje optimizacije
```

*Programski odsječak 25: Uspostavljanje veza među objektima*

Vidimo da se kreira nova kolekcija međurezultata (newColl) koja se preko objekta algoritma sprema u listu unutar objekta rezultata (IOptimizationResult). Zatim se kreira objekt *ObserverSaver*-a koji se dodaje u listu observera za dani objekt algoritma. Ovdje je prikladno podsjetiti se na deklaraciju konstruktora koji se koristi za kreiranje objekta CurrObjectiveValueSaver:

```
CurrObjectiveValueSaver(IObservedValuesCollection ^inCollToSave,
                        IIntermedValueProvider_CurrentObjectiveValue ^provider)
  {
  }
```

Bitno je primjetiti da je kao *provider* parametar predana upravo referenca na SO_GenerationalGA objekt.

Na gore opisani način je postignuta značajna fleksibilnost kod ugradnje funkcionalnosti za praćenje međurezultata generiranih tijekom optimizacije i zadovoljen osnovni cilj da se kod izgradnje optimizacijskog algoritma što više pojednostavni ugradnja funkcionalnosti za praćenje rezultata. Odabirom prikladnog skupa definiranih sučelja i implementacijom funkcija definiranim u tim sučeljima, izgrađeni objekt optimizacijskog algoritma na vrlo jednostavan način pruža široke mogućnosti za prikupljanje međurezultata.

#### 5.3.1.5 BaseResultsLib

U ovoj biblioteci su implementirani konkretni objekti povezani s konceptom rješenja algoritma. U konceptualnom modelu su identificirana tri osnovna elementa:

- optimizacijsko rješenje
- završno rješenje
- međurezultat optimizacije

Navedeni koncepti su programski realizirani preko odgovarajućih razreda sučelja, koji su opisani u sklopu opisa biblioteke BaseInterfacesLib dok su u ovoj biblioteci ugrađene realizacije konkretnih razreda koji zadovoljavaju dana sučelja.

Osnovni izgrađeni razred je razred OptimizationResult koji sadrži podatke o rezultatima optimizacije. Bitni detalji ugradnje se vide iz deklaracije razreda koja je prikazana u programskom odsječku 26.

```
public ref class OptimizationResult : public IOptimizationResult
{
public:
        OptimizationResult();

        virtual    void              SetFinalResult(IFinalResult ^inRes);
        virtual IFinalResult^        GetFinalResult();

        virtual String^     GetResultAsString()
        {
           // funkcija koja stvara string reprezentaciju kompletnog rezultata optimizacije
        }

        virtual void         AddNewObsCollection(IObservedValuesCollection ^inColl);

public:
        property int                 TotalIterNum;
        property double              DurationMs;

private:
        IFinalResult                 ^_FinalResult;
        List<IObservedValuesCollection ^> ^_listObsColl;
};
```

*Programski odsječak 26: Deklaracija razreda OptimizationResult*

Završni rezultat optimizacije (odnosno najbolje nađeno rješenje) je sadržan preko reference na IFinalResult objekt, dok su u listi _listObsColl sadržane sve generirane kolekcije međurezultata. Struktura opisa završnog rezultata ovisi o vrsti problema koji se rješavao, i to je razlog definiranja sučelja IFinalResult. To sučelje realiziraju četiri konkretna razreda, od kojih svaki reprezentira rezultat optimizacije za pojedinu vrstu problema (SO, SOWC, MO, MOWC). Bitna karakteristika svih izgrađenih konkretnih razreda je da u sebi sadrže reference na IProblemInput i IProblemOutput objekte, koji sadržavaju vrijednost ulaznih varijabli, odnosno vrijednosti izlaza za završno rješenje optimizacije. Pored ugradnje tih članskih varijabi, i realizacije članskih funkcija definiranih u sučelju IFinalResult, svaki od četiri implementirana razreda sadrži i određeni broj dodatnih članskih funkcija čija je namjena jednostavnije dohvaćanje relevantnih podataka.

Deklaracije razreda su sljedeće:

```
public ref class ResultSO : public IFinalResult
{
public:
        ResultSO(IProblemInput ^inInput, ProblemOutputSO ^inOutput);
        virtual String       ^GetResultAsString();

        virtual IProblemOutput        ^GetProblemOutputValues();
        virtual IProblemInput         ^GetFoundInputValues();

        double   getObjectiveValue() { return _OutputValues->getObjectiveValue(); }
private:
        IProblemInput                 ^_InputValues;
        ProblemOutputSO               ^_OutputValues;
};
```

```
public ref class ResultSOWC : public IFinalResult
{
public:
	ResultSOWC(IProblemInput ^inInput, ProblemOutputSOWC ^inOutput);

	virtual String                                          ^GetResultAsString();

	virtual IProblemOutput     ^GetProblemOutputValues();
	virtual IProblemInput                ^GetFoundInputValues();

	double   getObjectiveValue()        { return _OutputValues->getObjectiveValue(); }
	int      getNumConstraints()        { return _OutputValues->getNumConstraints(); }
	int      getNumViolatedConstraints();
	double   getCurrConstraintValue(int ConstrInd);
private:
	IProblemInput              ^_InputValues;
	ProblemOutputSOWC          ^_OutputValues;
};
public ref class ResultMO : public IFinalResult
{
public:
	ResultMO(IProblemInput ^inInput, ProblemOutputMO ^inOutput);

	virtual String      ^GetResultAsString();

	virtual IProblemOutput        ^GetProblemOutputValues();
	virtual IProblemInput         ^GetFoundInputValues();

	double   getObjectiveValue(int i)  { return _OutputValues->getObjectiveValue(i); }
	int      getNumObjectives()         { return _OutputValues->getNumObjectives(); }
private:
	IProblemInput              ^_InputValues;
	ProblemOutputMO            ^_OutputValues;
};
public ref class ResultMOWC : public IFinalResult
{
public:
	ResultMOWC(IProblemInput ^inInput, ProblemOutputMOWC ^inOutput);

	virtual String      ^GetResultAsString();

	virtual IProblemOutput        ^GetProblemOutputValues();
	virtual IProblemInput         ^GetFoundInputValues();

	double   getObjectiveValue(int i)    { return _OutputValues->getObjectiveValue(i); }
	int      getNumObjectives()          { return _OutputValues->getNumObjectives(); }

	int      getNumConstraints()         { return _OutputValues->getNumConstraints(); }
	int      getNumViolatedConstraints();
	double   getCurrConstraintValue(int ConstrInd);
private:
	IProblemInput              ^_InputValues;
	ProblemOutputMOWC          ^_OutputValues;
};
```

*Programski odsječak 27: Deklaracije razreda za modeliranje rezultata optimizacije*

Međutim, ukoliko se ima u vidu činjenica da je kod višekriterijske optimizacije cilj optimizacije generiranje Pareto skupa nedominiranih rješenja, jasno je da gore definirani razredi ResultMO i ResultMOWC neće biti prikladni za modeliranje takvog rezultata optimizacije. Stoga su definirana još dva razreda, ParetoSetMO i ParetoSetMOWC koji su također izvedeni iz sučelja IFinalResult i koji sadrže listu ResultMO, odnosno ResultMOWC objekata koji predstavljaju pojedine elemente Pareto skupa rješenja. Deklaracije razreda su sljedeće:

```
public ref class ParetoSetMO : public IFinalResult
{
public:
	virtual String^	GetResultAsString();

	int		GetNumParetoSol();
	ResultMO^	GetMember(int i);

	bool	AddWithCheck(IProblemMO ^prob, ResultMO ^resToAdd);
	void	AddWithoutCheck(ResultMO ^res);
private:
	List<ResultMO ^> ^_listSol;
};
public ref class ParetoSetMOWC : public IFinalResult
{
public:
	virtual String^	GetResultAsString();

	int		GetNumParetoSol();
	ResultMOWC^	GetMember(int i);

	bool	AddWithCheck(ProblemMOWC ^prob, ResultMOWC ^resToAdd);
	void	AddWithoutCheck(ResultMOWC ^res);
private:
	List<ResultMOWC ^> ^_listSol;
};
```

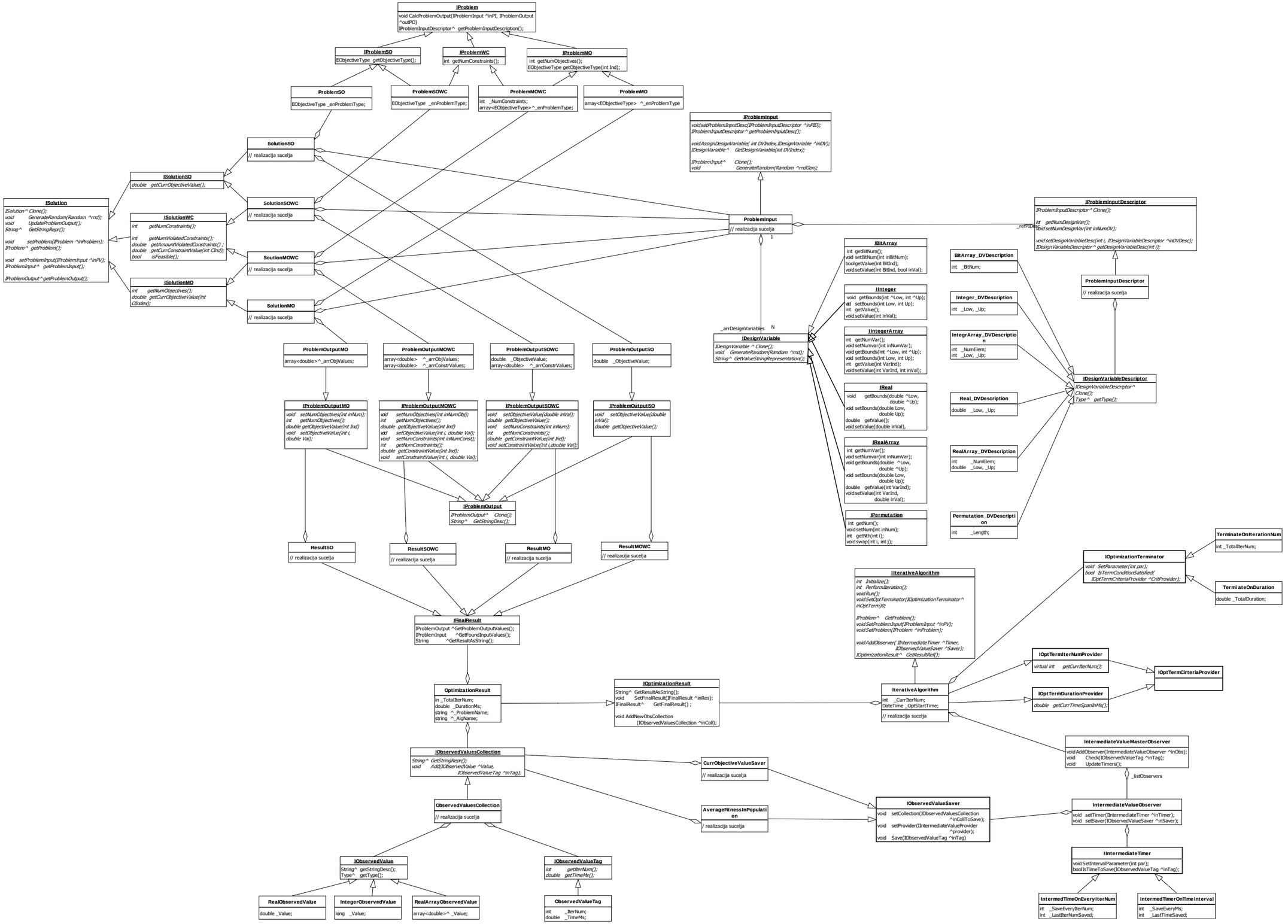
*Programski odsječak 28: Deklaracije razreda koji predstavljaju Pareto skup rješenja*

Funkcije AddWithCheck(), odnosno AddWithoutCheck() dodaju novo rješenje u listu Pareto rješenja, s time da funkcija AddWithCheck() i dodatno provjerava da li je rješenje koje se dodaje doista nedominirano u odnosu na rješenja koja su već sadržana u skupu.

Biblioteka BaseResultsLib sadrži i implementaciju određenog skupa razreda za praćenje međurezultata. Način njihovog rada je opisan kod opisa razreda sučelja IIterativeAlgorithm, a u ESOP okruženje su standardno ugrađeni:

- CurrObjectiveValueSaver
  - pohranjivanje trenutne vrijednosti funkcije cilje
- AverageObjectiveValueInPopulationSaver
  - pohranjivanje prosječne vrijednosti funkcije cilja u populaciji
- CurrMultiobjectiveValueSaver
  - pohranjivanje trenutne vrijednosti funkcija cilja kod višekriterijskih optimizacijskih problema

- CurrConstraintValuesSaver
  - pohranjivanje trenutne vrijednosti ograničenja
- BestFitnessInPopulationValueSaver
  - pohranjivanje vrijednosti dobrote najbolje jedinke u populaciji
- WorstFitnessInPopulationValueSaver
  - pohranjivanje vrijednosti dobrote najgore jedinke u populaciji
- AverageFitnessInPopulationValueSaver
  - pohranjivanje prosječne vrijednosti dobrote u populaciji
- BestObjectiveValueInPopulation
  - pohranjivanje najbolje vrijednosti funkcije cilja u populaciji
- AverageObjectiveValueInPopulation
  - pohranjivanje prosječne vrijednosti funkcije cilja za sve jedinke u populaciji
- WorstObjectiveValueInPopulation
  - pohranjivanje najlošije vrijednosti funkcije cilja u populaciji




**Slika 5.3 Dijagram razreda ESOP okvira za razvoj**

## 5.3.2 Skup ugrađenih optimizacijskih komponenti

Razredi sadržani u prethodno opisanom skupu biblioteka čine infrastrukturu ESOP optimizacijskog okruženja. U tehničkom smislu, opisani skup razreda predstavlja okvir za razvoj (odnosno, razvojnu okosnicu) unutar kojega se mogu ugraditi konkretne optimizacijske komponente.

Kako je jedan od zahtjeva na općenito optimizacijsko okruženje i postojanje što većeg skupa ugrađenih optimizacijskih komponenti koje omogućavaju *out-of-the-box* rješavanje određenog skupa optimizacijskih problema, u ovom potpoglavlju će se opisati skup komponenti koje su već ugrađene u ESOP okruženje.

Taj skup komponenti je ugrađen unutar biblioteke StandardOptimization-ObjectsLib. Ovdje je bitno naglasiti da se proširenje ESOP okruženja s novim optimizacijskim komponentama obavlja na potpuno istovjetan način. Odnosno, skup programskih tehnika koje su primjenjene kod izgradnje standardnih optimizacijskih komponenti je isti i u slučaju definiranja novih komponenti prilikom proširivanja ESOP okruženja, što će biti opisano u potpoglavlju 5.4. S obzirom na veliki broj ugrađenih optimizacijskih komponenti, sama biblioteka je organizirana u četiri virtualna direktorija unutar kojih su optimizacijske komponente grupirane po vrsti (problemi, vrste ulaznih varijabli, algoritmi, operatori) a tako je strukturiran i opis koji slijedi.

### 5.3.2.1 Optimizacijski problemi

Ugradnja novih optimizacijskih komponenti koje će predstavljati optimizacijske probleme je vrlo jednostavna. Potrebno je izgraditi novi razred koji će biti izveden iz odgovarajućeg sučelja (IProblem i njegove specijalizacije) i implementirati članske funkcije definirane u tom sučelju. S obzirom da je dio funkcionalnosti koji je zajednički svim problemima ugrađen u apstraktne razrede (opisane u odjeljku 5.3.1.3.1), novi razredi se ne izvode direktno iz sučelja IProblem, već iz odgovarajućeg apstraktnog razreda (ProblemSO, ProblemSOWC, ProblemMO, ProblemMOWC) koji odgovara vrsti optimizacijskog problema.

#### 5.3.2.1.1 Opis načina ugradnje optimizacijskog problema

Način ugradnje se najbolje može ilustrirati na konkretnom primjeru. Kao primjer će poslužiti jednostavan problem optimizacije realne funkcije, definirane u DeJong-ovom skupu testnih problema [DeJong1975]. Definicija problema je sljedeća:

$$\min\ f(x) = \sum_{i=1}^{N} \left(x_i - 1\right)^2,\ \ uz\quad -5.12 \le x_i \le 5.12 \tag{5.3.1}$$

Kompletan programski kôd koji je potrebno implementirati kako bi se navedeni problem ugradio u ESOP okruženje je sljedeći (problem je definiran za $N = 3$):

```
public ref class DeJong_F1_TestProblem : public ProblemSO
{
public:
        DeJong_F1_TestProblem() : ProblemSO(EObjectiveType::MIN_OBJECTIVE) { }

        virtual    void      CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO);
```

```
        virtual    IProblemInputDescriptor^ getProblemInputDescription();
}
void      DeJong_F1_TestProblem::CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO)
{
        IRealArray ^arrX = dynamic_cast<IRealArray ^>(inPI->GetDesignVariable(0));

        double sum = 0;
        for( int i=0; i<3; i++ ) {
             double x = arrX->getValue(i);
             sum += (x - 1) * (x - 1);
        }
        ProblemOutputSO          ^outVal = dynamic_cast<ProblemOutputSO ^>(outPO);
        outVal->setObjectiveValue(sum);
}
IProblemInputDescriptor^ DeJong_F1_TestProblem::getProblemInputDescription()
{
        ProblemInputDescriptor ^newPID = gcnew ProblemInputDescriptor(1);

        RealArray_DVDescription ^RealArr = gcnew RealArray_DVDescription(3,-5.12,5.12);
        newPID->setDesignVariableDesc(0, RealArr);

        return newPID;
}
```

*Programski odsječak 29: Deklaracija razreda problema DeJong_F1*

U funkciji getProblemInputDescription() se kreira novi ProblemInputDescriptor objekt u kojem se opisuje skup varijabli nad kojima je problem definiran. Kako je DeJong_F1 problem definiran nad skupom od 3 realne varijable istih karakteristika, skup ulaznih varijabli je modeliran kao polje realnih brojeva (3 varijable → polje s 3 elementa) te se u ProblemInputDescriptor objekt dodanje jedan RealArray_DVDescription objekt koji definira karakteristike tog polja realnih varijabli.

Druga funkcija koju je potrebno implementirati je funkcija CalcProblemOutput() u koju se ugrađuje programski kôd pomoću kojega će se izračunati vrijednosti izlaza optimizacijskog problema. S obzirom da se radi o problemu jednokriterijske optimizacije bez ograničenja, u ovoj funkciji je potrebno ugraditi kôd za izračunavanje vrijednosti funkcije cilja na osnovu vrijednosti ulaznih varijabli.

Ulazne varijable problema su sadržane u objektu ProblemInput te je najprije potrebno dohvatiti odgovarajuću varijablu (poziv funkcije GetDesignVariable()) i zatim obaviti konverziju (engl. c*ast*) na odgovarajuću vrstu varijable. Ovdje je bitno uočiti ulogu sučelja IRealArray. Naime, kako je polje realnih varijabli moguće konkretno programski realizirati na više načina (npr. kao standardno polje *float* ili *double* varijabli, ali i kao polje bitova) implementacija funkcije CalcProblemOutput() koristi samo apstraktno definirano sučelje IRealArray koje zadovoljavaju konkretni razredi koji predstavljaju pole realnih brojeva. Odnosno, kod rješavanja DeJong_F1 problema se mogu koristiti različite reprezentacije za varijable problema, sve dok one zadovoljavaju sučelje IRealArray.

Nakon dohvata skupa ulaznih varijabli problema (u ovom slučaju je samo jedna, ali ih može biti proizvoljan broj), provodi se izračunavanje vrijednosti funkcije cilja, te se rezultat sprema u objekt ProblemOutputSO. Ovdje je također potrebno obaviti konverziju s obzirom da je funkcija CalcProblemOutput() definirana

apstraktno, odnosno, u svojoj deklaraciji koristi općeniti razred sučelja IProblemOutput).

#### 5.3.2.1.2 Ugrađeni problemi optimizacije realnih funkcija

S obzirom na heuristički karakter optimizacijskih postupaka kojima je prilagođeno ESOP optimizacijsko okruženje, vrlo važan element tog okruženja čini i skup ugrađenih testnih problema nad kojima se mogu ispitivati kvaliteta i efikasnost izgrađenih optimizacijskih postupaka. Ugradnja problema je obavljena u skladu s procedurom opisanom u prethodnom odjeljku, a skup testnih problema (engl. *test suite*) koji su standardno ugrađeni u ESOP je sljedeći:

- DeJong-ov skup problema [DeJong1975] u kojem su definirane sljedeće funkcije:
  - SphereModel funkcija (nazvana DeJong_F1 u ESOP okruženju)
  - Generalizirana Rosenbrockova funkcija
  - Step funkcija
  - Kvartična funkcija sa šumom
  - *Shekel's foxhole* funkcija
- skup problema definiran za prvo i drugo međunarodno natjecanje u evolucijskom računarstvu (ICEO - International Contest in Evolutionary Computation [Bersini1996]) koji sadržava sljedeće funkcije:
  - Michalewicz-eva funkcija s 5 i 10 varijabli
  - Modificirana Langerman-ova funkcija s 5 i 10 varijabli
  - Odd square funkcija
  - Bump funkcija
  - Epistatička Michalewicz-eva funkcija

Pored funkcija koje su sadržane u gore navedenim skupovima testnih problema, ugrađeno je još nekoliko popularnih testnih problema realne optimizacije: Ackleyeva funkcija [Ackley1987], Rastriginova funkcija [Mühlenbein1991] i Griewangkov problem [Törn1989].

Optimizacijski problemi definirani gornjim funkcijama predstavljaju jednokriterijske probleme bez ograničenja. Pored tog skupa problema, u ESOP je ugrađen i skup jednokriterijskih problema s ograničenjima koji je definiran na posebnoj sesiji u sklopu IEEE kongresa o evolucijskom izračunavanju koja je bila posvećena optimizaciji realnih problema s ograničenjima [Suganthan2005]. Skup problema čine 24 problema kojima su obuhvaćene različite vrste ograničenja (linearna, kvadratična, nelinearna) s različitim varijantama funkcija cilja.

S obzirom da je razvijeno ESOP okruženje primjenjeno na rješavanje složenog praktičnog problema višekriterijske optimizacije parametara mehaničke strukture brodske konstrukcije (što je opisano u završnom poglavlju ove disertacije), definiran je i odgovarajući skup testnih problema koji predstavljaju višekriterijske probleme s ograničenjima. Unutar tog skupa je definirano šest optimizacijskih problema koji su često korišteni u praksi. U literaturi se referenciraju kao BNH, SRN, TNK, OSY, Two-bar Truss Design i Welded Beam Design problemi, a njihova definicija se može naći u [Kurpati2002]:

Svi navedeni problemi su izgrađeni kao posebni razredi unutar StandardOptimizationObjectsLib biblioteke na način kako je to opisano u prethodnom odjeljku, s time da se prilikom njihove izgradnje koristilo odgovarajuće apstraktne bazne razrede, ovisno o vrsti optimizacijskog problema.

#### 5.3.2.1.3 Ugradnja problema trgovačkog putnika (TSP)

Kako je već ranije spomenuto, mnogobrojni problemi iz prakse se mogu svesti na problem optimizacije realne funkcije (eventualno s ograničenjima). Međutim, u praksi se često javljaju i problemi koji spadaju u područje diskretne, odnosno kombinatorijalne optimizacije. Kako bi se prikazao način ugradnje takvih problema u ESOP okruženje, u ovom odjeljku će se dati opis ugradnje problema trgovačkog putnika.

Problem trgovačkog putnika predstavlja jednokriterijski problem u kojem nema definiranih ograničenja. Preciznije rečeno, korištenjem prikladne reprezentacije se vrlo jednostavno može eliminirati jedino ograničenje problema a to je da je se svaki grad posjeti samo jedanput. U ESOP okruženju su karakteristike problema trgovačkog putnika modelirane preko apstraktnog razreda TSP_Base sa sljedećom deklaracijom:

```
public ref class TSP_Base abstract : public ProblemSO
{
public:
    TSP_Base() : ProblemSO(EObjectiveType::MIN_OBJECTIVE) { }

    //  funkcije definirane u sučelju ProblemSO
    virtual void CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO) override;
    virtual IProblemInputDescriptor^ getProblemInputDescription() override;

    // funkcije specifične za TSP problem
    virtual  int            getNumCities() = 0;
    virtual  double         getDistBetweenCities(int CityInd1, int CityInd2) = 0;
};
```

*Programski odsječak 30: Deklaracija razreda TSP_Base*

Uloga ovog razreda je modeliranje aspekata optimizacijskog problema koji su specifični za problem trgovačkog putnika i on će predstavljati bazni razred iz kojega će biti izvedeni konkretni razredi koji će predstavljati pojedine instance TSP problema. Zajednička funkcionalnost za sve TSP probleme je ugrađena u funkcijama CalcProblemOutput() i getProblemInputDescription(). Funkcija CalcProblemOutput() za predanu instancu rješenja problema koje definira konkretan redoslijed obilaska gradova izračunava ukupnu dužinu puta, dok funkcije getProblemInputDescription() definira karakteristike reprezentacije varijabli problema. Izgradnja tih funkcija je vrlo jednostavna:

```
IProblemInputDescriptor^ TSP_Base::getProblemInputDescription()
{
   ProblemInputDescriptor^         newPID = gcnew ProblemInputDescriptor(1);

   Permutation_DVDescription^      Perm = gcnew Permutation_DVDescription(getNumCities());
   newPID->setDesignVariableDesc(0, Perm);

   return newPID;
}
```

```
void TSP_Base::CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO)
{
    IPermutation^    x = dynamic_cast<IPermutation ^>(inPI->GetDesignVariable(0));
    double           TotalTripLen = 0;

    int    City1 = x->getNth(0);
    int    City2;
    for( int i=1; i<getNumCities(); i++ )
    {
        City2 = x->getNth(i);
        TotalTripLen += getDistBetweenCities(City1, City2);
        City1 = City2;
    }

    ProblemOutputSO^    outVal = dynamic_cast<ProblemOutputSO ^>(outPO);
    outVal->setObjectiveValue(TotalTripLen);
}
```

*Programski odsječak 31: Ugradnja funkcija getProblemInputDescription() i CalcProblemOutput()*

Funkcija getProblemInputDescription() definira karakteristike ulazne varijable problema, odnosno, u njenoj implementaciji je definirano da odabrana konkretna reprezentacija mora zadovoljavati sučelje IPermutation što je prirodna reprezentacija za TSP s obzirom da rješenje mora definirati redoslijed obilaska *n* gradova.

Funkcija CalcProblemOutput() se u svom radu oslanja na funkcije getNumCities() i getDistBetweenCities() koje opisuju karakteristike konkretne instance TSP problema koji se riješava. Navedene dvije funkcije su čiste virtualne funkcije što znači da će se morati definirati u izvedenim razredima.

Pomoću ove dvije funkcije je modelirana činjenica da je za potpuno definiranje TSP problema potrebno znati ukupan broj gradova koji se mora obići, i udaljenost između svaka dva grada. Kod kreiranja konkretnih instanci TSP problema, potrebni podaci se mogu programski opisati na dva načina. Može se izgraditi razred koji direktno u programskom kôdu sadrži potrebne podatke, a druga mogućnost je da se potrebni podaci učitavaju iz datoteke (što je i najčešće slučaj, s obzirom da realne instance problema imaju i do desetak tisuća gradova).

Stoga su izgrađena dva razreda, TSP_Complete i TSP_FromFile. Razred TSP_Complete je također apstraktan razred i njegova definicija je sljedeća:

```
public ref class TSP_Complete abstract : public TSP_Base
{
public:
        TSP_Complete();
        TSP_Complete(int inNumCities);

        virtual int          getNumCities() override;
        virtual double       getDistBetweenCities(int CityInd1, int CityInd2) override;

        void                 setNumCities(int inCityNum);
protected:
        int        _NumCities;
        array<double, 2> ^_arrDist;
};
```

*Programski odsječak 32: Deklaracija razreda TSP_Complete*

Razred u sebi sadrži članske varijable koje predstavljaju podatke o broju gradova (_NumCities) odnosno podatke o udaljenosti među gradovima. Kod definiranja nove instance TSP problema je stoga potrebno kreirati novi razred koji će u svom konstruktoru definirati vrijednosti tih članskih varijabli. Primjer definiranja konkretnog razred koji predstavlja jednostavnu instancu problema sa 17 gradova je sljedeći:

```
public ref class TSP_gr17 : public TSP_Complete
{
public:
    TSP_gr17() : TSP_Complete(17)
    {
        // polje se inicijalizira sa udaljenostima gradova, s time da je dana samo donja
        // dijagonalna matrica udaljenosti (problem je simetričan)
        array<int> ^tempDist = {
            0,
            633, 0,
            257, 390, 0,
            91,  661, 228, 0,
            412, 227, 169, 383, 0,
            150, 488, 112, 120, 267, 0,
            80,  572, 196, 77,  351, 63,  0,
            134, 530, 154, 105, 309, 34,  29, 0,
            259, 555, 372, 175, 338, 264, 232, 249, 0,
            505, 289, 262, 476, 196, 360, 444, 402, 495, 0,
            353, 282, 110, 324, 61,  208, 292, 250, 352, 154, 0,
            324, 638, 437, 240, 421, 329, 297, 314, 95,  578, 435, 0,
            70,  567, 191, 27,  346, 83,  47,  68,  189, 439, 287, 254, 0,
            211, 466, 74,  182, 243, 105, 150, 108, 326, 336, 184, 391, 145, 0,
            268, 420, 53,  239, 199, 123, 207, 165, 383, 240, 140, 448, 202, 57,  0,
            246, 745, 472, 237, 528, 364, 332, 349, 202, 685, 542, 157, 289, 426, 483, 0,
            121, 518, 142, 84,  297, 35,  29,  36,  236, 390, 238, 301, 55,  96,  153, 336, 0
        };

        // sada s tim vrijednostima treba popuniti matricu udaljenosti
        int        Ind = 0;
        for( int i=0; i<17; i++ )
                for(int j=0; j<=i; j++ ) {
                    _arrDist[i,j] = tempDist[Ind++];
                    _arrDist[j,i] = _arrDist[i,j];
                }
        }
};
```

*Programski odsječak 33: Primjer deklaracije razreda s definiranim kompletnim TSP problemom*

Razred TSP_ FromFile ima sljedeću (vrlo jednostavnu) deklaraciju:

```
public ref class TSP_FromFile : public TSP_Complete
{
public:
        virtual void        InitFromFile(String ^FileName) = 0;
}
```

Razred TSP_FromFile također predstavlja apstraktni razred koji je izveden iz razred TSP_Complete, ali koji definira i dodatni zahtjev na izvedene razrede a to je implementacija funkcije InitFromFile(). S obzirom da formati podataka u datotekama koje opisuju pojedine instance TSP problema mogu biti različiti, na ovaj način je

omogućeno definiranje posebnog (sada konkretnog) razreda za različite vrste formata ulaznih datoteka. Razlika u odnosu na TSP_Complete razred je u tome što se potrebni podaci ne definiraju programski u konstruktoru već će biti popunjeni unutar funkcije InitFromFile(), a na osnovu sadržaja datoteke s predanim imenom.

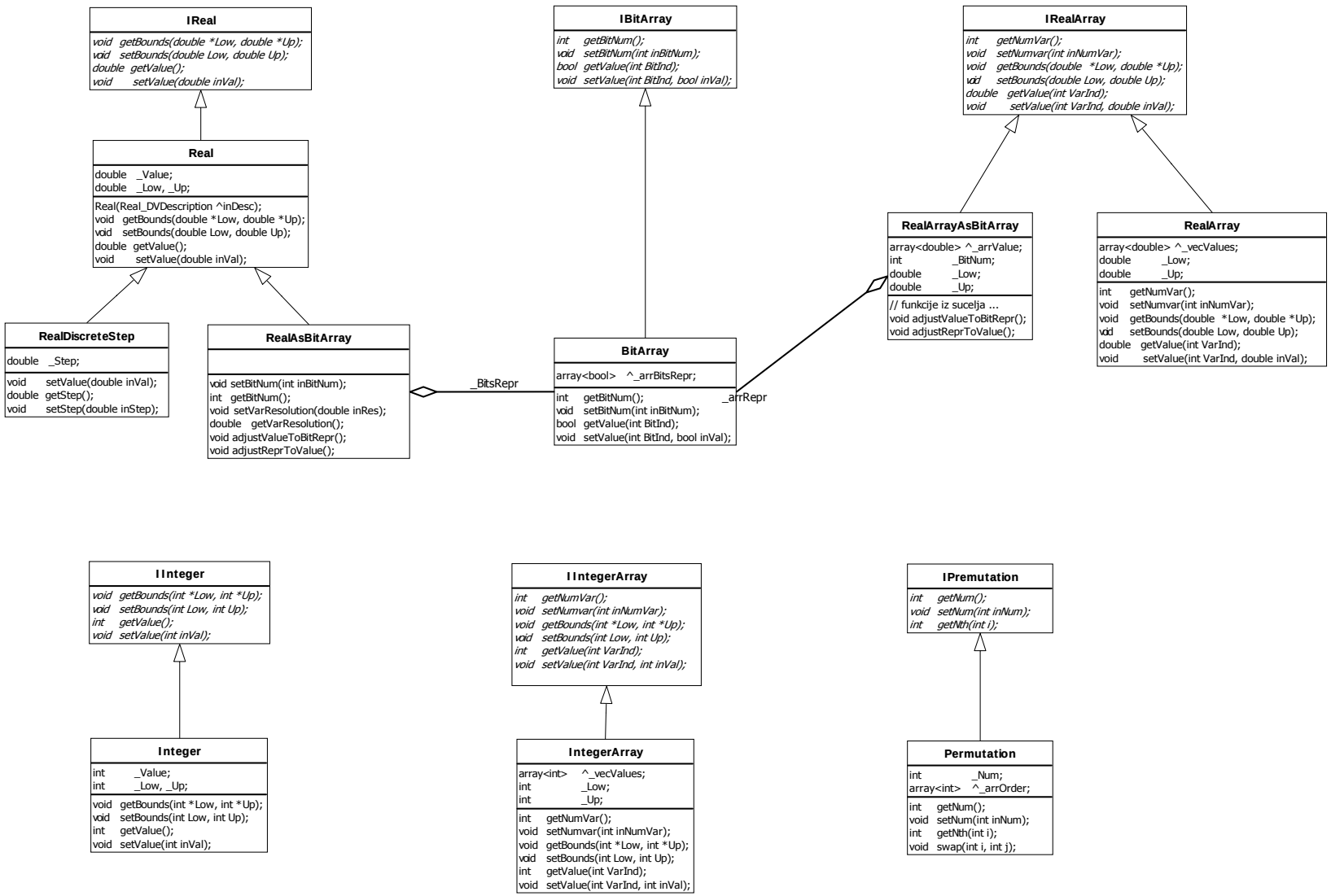
S obzirom da je za problem trgovačkog putnika najšire korišten skup problema definiran u biblioteci TSPLIB [Reinelt1991], izgrađen je razred TSP_LoadFromTSPLIB koji ima ugradnju funkcije InitFromFile() na osnovu poznatog formata datoteka kojima su definirani problemi sadržani u toj biblioteci problema. Korištenjem tog razreda je omogućeno vrlo jednostavno programsko definiranje objekata koji predstavljaju pojedine TSP probleme što istraživaču značajno olakšava rad prilikom istraživanja optimizacijskih postupaka primjenjivih na rješavanje problema trgovačkog putnika.

#### 5.3.2.2 Reprezentacije varijabli

Unutar ESOP okruženja je predefiniran skup razreda pomoću kojih su realizirane standardne vrste ulaznih varijabli problema. Ugrađene su sljedeće vrste ulaznih varijabli:

- BitArray – predstavlja polje bitova. Rijetko se koristi kao ulazna varijabla u optimizacijskim problemima, ali se koristi kao reprezentacija za *AsBitArray* varijable. Zadovoljava sučelje IBitArray.
- Integer – predstavlja klasičnu cjelobrojnu varijablu sa definiranim rasponom dopuštenih vrijednosti. Zadovoljava sučelje IInteger.
- IntegerArray – predstavlja polje istovjetnih cjelobrojnih varijabli. Zadovoljava sučelje IIntegerArray.
- Real – predstavlja realnu varijablu s definiranim rasponom dopuštenih vrijednosti. Zadovoljava sučelje IReal.
- RealAsBitArray – predstavlja realnu varijablu reprezentiranu preko polja bitova s određenom točnošću (koja je određena brojem bitova za rpeezentaciju). Zadovoljava sučelje IReal.
- RealDiscreteStep – predstavlja realnu varijablu s diskretnim skupom dopuštenih vrijednosti, definiranih određenim korakom. Zadovoljava sučelje IReal.
- RealArray – predstavlja polje istovjetnih realnih varijabli. Zadovoljava sučelje IRealArray.
- RealArrayAsBitArray – predstavlja polje realnih varijabli koje su interno reprezentirane kao polja bitova. Zadovoljava sučelje IRealArray, a u implementaciji koristi BitArray reprezentaciju
- Permutation – razred koji predstavlja matematičku strukturu permutacije.

S obzirom da su implementacije navedenih razreda relativno jednostavne, neće se dati detaljni opis već samo UML dijagram razreda iz kojeg je vidljiv skup članskih funkcija i članskih varijabli svakog pojedinog razreda te njihovi međusobni odnosi.




**Slika 5.4 Dijagram razreda za skup razreda reprezentacija ugrađenih u ESOP**

#### 5.3.2.3 Ugrađeni optimizacijski algoritmi

Izgrađeni apstraktni razred IterativeAlgorithm predstavlja osnovu za ugrađivanje bilo kakvog optimizacijskog algoritma u ESOP okruženje. Ugradnja novog algoritma zahtijeva implementaciju novog razreda koji će biti izveden iz razreda IterativeAlgorithm čime će novoizgrađenom razredu postati dostupna cjelokupna funkcionalnost ugrađena u razred IterativeAlgorithm.

Osnovni cilj definiranja razreda IterativeAlgorithm je oslobađanje graditelja novog algoritma potrebe da se brine o ugradnji generičke funkcionalnosti vezane uz određivanje trajanje optimizacije te rukovanje rezultatima i međurezultatima. Za istraživača koji želi ugraditi novi optimizacijski postupak u ESOP okruženje to znači da je u novoizgrađenom razredu kroz koji će se realizirati dani optimizacijski postupak potrebno samo ugraditi implementacije funkcija Initialize() i PerformIteration() koje obavljaju inicijalizaciju objekta algoritma, odnosno obavljaju jednu njegovu iteraciju, respektivno. U sljedećih nekoliko odjeljaka će se opisati implementacija nekoliko optimizacijskih postupaka koji su standardno ugrađeni u ESOP okruženje.

##### 5.3.2.3.1 Ugradnja optimizacijskog postupka slučajnog pretraživanja

Iako optimizacijski postupak slučajnog pretraživanja (engl. *random search*) predstavlja gotovo trivijalnu optimizacijsku metodu koja se u praksi nikada ne koristi za rješavanje iole složenijih optimizacijskih problema, u određenim situacijama je zgodno iskoristiti takav postupak radi istraživanja prostora rješenja danog optimizacijskog problema. Slučajnim generiranjem određenog broja rješenja se može steći dobar uvid u složenost optimizacijskog problema, odnosno u strukturu pripadnog prostora rješenja. Takav postupak je pogotovo prikladan kod optimizacijskih problema s ograničenjima.

Ugradnja samog algoritma u ESOP okruženje je vrlo jednostavna, s obzirom na postojanje GenerateRandom() funkcije, definirane u ISolution razredu sučelja, a koja se oslanja na GenerateRandom() članske funkcije ugrađene u svaku vrstu ulazne varijable problema.

Izgrađene je nekoliko verzija algoritma slučajnog pretraživanja, po jedan za svaku vrstu optimizacijskog problema (SO, SOWC, MO i MOWC). Sama ugradnja se svodi na slučajno generiranje rješenja u svakoj iteraciji optimizacijskog postupka, s time da se tijekom postupka optimizacije pamti najbolje nađeno rješenje. Kod izgradnje višekriterijske varijante algoritma slučajnog pretraživanja, u svakoj iteraciji se generira populacija slučajnih rješenja, iz koje se uzimaju nedominirana rješenja i stavljaju u Pareto skup rješenja. Izgrađeni su sljedeći razredi koji predstavljaju optimizacijske postupke slučajnog pretraživanja za odgovarajuću vrstu problema:

- RandomSearch_AlgorithmSO
- RandomSearch_AlgorithmSOWC
- RandomSearch_AlgorithmMO
- RandomSearch_AlgorithmMOWC

#### 5.3.2.3.2 Ugradnja genetičkih algoritama

Kod izgradnje genetičkih algoritama (odnosno razreda koji će predstavljati optimizacijski postupak genetičkog algoritma), potrebno je obratiti pažnju na sljedeće generičke karakteristike svake realizacije genetičkog algoritma koje proizlaze iz teorijskog opisa danog u odjeljku 2.2.3:

- genetički algoritam operira nad populacijom rješenja koja se prije početka optimizacije mora inicijalizirati
- genetički algoritam svakom rješenju (jedinki) pridružuje određenu mjeru kvalitete (dobrota rješenja) korištenjem operatora pridruživanja dobrote (hrvatski termin dobrota će se u daljenjem tekstu koristiti zajedno s terminom fitness, a kako bi se olakšalo razumijevanje realiziranih programskih komponenti kod čijeg imenovanja se koristio engleski jezik)
- genetički algoritam koristi operator selekcije za odabir jedinki za reprodukciju na osnovu kojih će se kreirati nova populacija
- genetički algoritam ima definirane operatore križanja i mutacije pomoću kojih se iz skupa jedinki odabranih za reprodukciju generiraju nove jedinke koje će sačinjavati novu populaciju

Skup zajedničkih karakteristika genetičkih algoritama je realiziran definiranjem apstraktnog razreda BaseGA sa sljedećom definicijom:

```
public ref class BaseGA abstract :
                        public IterativeAlgorithm,
                        public IIntermedValueProvider_BestFitnessInCurrPopulation,
                        public IIntermedValueProvider_AverageFitnessInCurrPopulation,
                        public IIntermedValueProvider_WorstFitnessInCurrPopulation,
{
public:
        BaseGA();

        // realizacija funkcija definiranih u IIterativeAlgorithm razredima sučelja
        virtual   int       Initialize() override;
        virtual   int       PerformIteration() override;

        // zadovoljavanje sučelja definiranih u IIntermediateValueProvider razredima sučelja
        virtual   double    getCurrBestFitnessValue();
        virtual   double    getCurrAverageFitnessValue();
        virtual   double    getCurrWorstFitnessValue();

        void      setPopulationSize(int inSize);
        int       getPopulationSize();

        void      setFitnessAssignmentOp(IFitnessAssignmentOperator  ^inFitAssign);
        void      setFitnessSelectionOp(IFitnessBasedSelectionOperator  ^inSelection);

        void      setCrossoverOperator(int Index, ICrossoverOperator2To2 ^inCrossover);
        void      setMutationOperator(int Index, IMutationOp ^inMutation);

protected:
        int       _PopSize;
        SolutionContainer  ^_Population;
        array<double>      ^_arrFitness;
```

```
        ISolution           ^_SolutionInstance;

        IFitnessAssignmentOperator          ^_opFitnesAssignment;
        IFitnessBasedSelectionOperator      ^_opSelector;

        double      _CrossoverProbability;
        double      _MutationProbability;

        OperatorContainer       ^_opCrossover;
        OperatorContainer       ^_opMutation;

        Random ^_RandomGenerator;
};
```

*Programski odsječak 34: Deklaracija razreda BaseGA*

Nasljeđivanjem razreda IterativeAlgorithm, BaseGA razred preuzima cjelokupnu funkcionalnost vezanu uz definiranje trajanja optimizacije i rukovanje rezultatima i međurezultatima.

Članska varijabla _Population predstavlja populaciju rješenja i za njenu realizaciju se koristi prethodno izgrađeni razred SolutionContainer, dok je članska varijabla _arrFitness polje realnih brojeva koje sadrži izračunati fitnes za svako rješenje iz populacije. Varijable _opFitnessAssignment i _opSelector predstavljaju reference na odabrane konkretne objekte koji će predstavljati operatore pridruživanja fitnesa i selekcije.

Kako su operatori križanja i mutacije ovisni o odabranoj reprezentaciji jedinki (rješenja) nad kojima operira algoritam, a uzimajući u obzir da reprezentaciju rješenja problema čini skup ulaznih varijabli nad kojima je definiran problem i koje mogu biti različitih tipova, nije moguće genetskom algoritmu pridružiti samo jedan operator križanja i mutacije već se za svaku od ulaznih varijabli mora definirati operator prikladan za tu vrstu varijable.

Stoga je definiran novi razred OperatorContainer koji predstavlja jednostavnu kolekciju operatora (odnosno, kao člansku varijablu ima polje referenci na IOperator objekte). S obzirom da je objektu koji realizira algoritam poznata konkretno odabrana reprezentacija rješenja (preko funkcije SetProblemInput() koja se mora pozvati prije pokretanja optimizacije), poznat mu je i broj ulaznih varijabli te se preko definiranih funkcija setCrossoverOperator() i setMutationOperator() može (mora) pridružiti konkretan operator za svaku pojedinačnu ulaznu varijablu problema.

Realizacija funkcija Initialize() i PerformIteration() predstavlja dušu optimizacijskog postupka. Zahvaljujući korištenju definiranih sučelja, unutar razreda BaseGA su ugrađene generičke implementacije danih funkcija čiji je način rada najjasnije može vidjeti iz njihovog programskog kôda:

```
int     BaseGA::Initialize()
{
        _Population->Initialize(_PopSize, _SolutionInstance);

        // i sada treba svakoj kreiranoj jedinki treba ažurirati izlaz
        for( int i=0; i<_Population->getSize(); i++ )
        {
```

```
                ISolution    ^SolPop = _Population->getSolution(i);
                SolPop->GenerateRandom(_RandomGenerator);
                SolPop->UpdateProblemOutput();
        }
        return 0;
}
int     BaseGA::PerformIteration()
{
    SolutionContainer       ^newPopulation = gcnew SolutionContainer();

    _opFitnesAssignment->Assign(_Population, GetProblem(), _arrFitness);

    while(newPopulation->getSize() < _Population->getSize() )
    {
        array<int>          ^vecOutSel = gcnew array<int>(2);
        _opSelector->Select(_arrFitness, vecOutSel, 2);

        // jedinke odabrane za reprodukciju
        ISolution ^Parent1 = _Population->getSolution(vecOutSel[0]);
        ISolution ^Parent2 = _Population->getSolution(vecOutSel[1]);

        // kreiramo djecu (koji će inicijalno biti klonovi roditelja)
        ISolution ^Child1 = _Population->getSolution(vecOutSel[0])->Clone();
        ISolution ^Child2 = _Population->getSolution(vecOutSel[1])->Clone();

        // primjenjujemo operatore križanja (s određenom vjerojatnošću)
        if( _RandomGenerator->Next() % 10000 / 10000. < _CrossoverProbability )
        {
            int NumVar = GetProblem()->getProblemInputDescription()->getNumDesignVar();
            for( int i=0; i<NumVar; i++ )  {
               ICrossoverOperator2To2 ^Cross = dynamic_cast<ICrossoverOperator2To2 ^>
                                                    (_opCrossover->getOperator(i));

              IDesignVariable ^p1 = Parent1->getProblemInput()->GetDesignVariable(i));
              IDesignVariable ^p2 = Parent2->getProblemInput()->GetDesignVariable(i));
              IDesignVariable ^c1 = Child1->getProblemInput()->GetDesignVariable(i);
              IDesignVariable ^c2 = Child2->getProblemInput()->GetDesignVariable(i);

               // obavljamo križanje
              Cross->Recombine(p1, p2, c1, c2);
            }
          }
         if( _RandomGenerator->Next() % 10000 / 10000. < _MutationProbability )
          {
             for( int i=0; i<NumVar; i++ )
             {
                  // na svako kreirano dijete primjenjujemo (možda) mutaciju
                  IMutationOp         ^Mutator = dynamic_cast<IMutationOp ^>
                                                    (_opMutation->getOperator(i));

                  Mutator->Mutate(Child1->getProblemInput()->GetDesignVariable(i));
                  Mutator->Mutate(Child2->getProblemInput()->GetDesignVariable(i));
              }
          }
          // za novokreirane jedinke treba izracunati ažurirati vrijednost izlaza
          Child1->UpdateProblemOutput();
          Child2->UpdateProblemOutput();
```

```
            // ubacujemo kreirane jedinke u novu populaciju
            newPopulation->pushbackNewSolution(Child1);
            newPopulation->pushbackNewSolution(Child2);
        }  // end while

        // i sada nam nova populacija postaje trenutna populacija
        _Population = newPopulation;

        return 0;
    }
```

*Programski odsječak 35: Ugradnja funkcija Initialize() i PerformIteration() u BaseGA razredu*

Iako bi se iz navedenog opisa dalo zaključiti da izgrađeni BaseGA razred predstavlja generičku implementaciju genetičkog algoritma koja se može primjeniti na rješavanje bilo kakvog optimizacijskog problema, jasno uz postojanje prikladnih objekata koji realiziraju operatore križanja i mutacije, to nije posve istina. Razlog tome leži u ovisnosti realizacije algoritma o vrsti optimizacijskog problema, odnosno o tome da li se radi o jednokriterijskom ili višekriterijskom optimizacijskom problemu. Kako je ranije navedeno, unutar ESOP okruženja je definirano nekoliko konkretnih razreda koji modeliraju koncept rješenja problema (SolutionSO, SolutionSOWC, SolutionMO, SolutionMOWC) i koji predstavljaju objekte nad kojima operiraju optimizacijski postupci ugrađeni u ESOP. Uloga objekta rješenja se može vidjeti iz članske varijable _SolutionInstance koja predstavlja referencu na konkretan objekt rješenja.

Stoga su izgrađene četiri specijalizacije razreda BaseGA (SO_GenerationalGA, SOWC_GenerationalGA, MO_GenerationalGA i MOWC_GenerationalGA). Svaka od danih specijalizacija je izgrađena tako da kreira objekt rješenja odgovarajućeg tipa, a većinu svoje funkcionalnosti nasljeđuju iz bazne klase (BaseGA).

Bitna karakteristika izgrađenih razreda je da omogućavaju daljnje nasljeđivanje, odnosno izgradnju novih optimizacijskih komponenti koje će iskoristiti dio ugrađene funkcionalnost. To je omogućeno činjenicom da su funkcije Initialize() i PerformIteration() virtualne, što omogućava njihovo redefiniranje u izvedenim razredima. Primjerice, u gornjoj realizaciji funkcije PerformIteration() je korišten operator ICrossoverOperator2To2 koji na osnovu dvaju jedinki roditelja kreira dvoje djece. Izgradnja genetičkog algoritma koji bi koristio operator ICrossoverOperator2To1 (znači da se iz dvaju roditelja kreira jedno dijete) zahtijeva izgradnju novog razreda (izvedenog iz odgovarajućeg GenerationalGA razreda) i implementaciju samo funkcije PerformIteration().

**Skup ugrađenih operatora**

Esencijalnu ulogu kod optimizacije pomoću genetičkih algoritama imaju genetički operatori pridruživanja dobrote, selekcije, križanja i mutacije. Njihova ugradnja u ESOP okruženje je vrlo jednostavna, i bazira se na zadovoljavanju razreda sučelja definiranih u pododjeljku 5.3.1.2.6. Odnosno, ugradnja operatora podrazumijeva izgradnju novog razreda koji će zadovoljavati odgovarajuće definirano sučelje.

***Ugrađeni operatori pridruživanja fitnesa***

Osnovni zadatak operatora fitnesa je da na osnovu sadržaja jedinke i njenog „genetičkog materijala“ izračuna njen fitnes. ESOP okruženje standardno dolazi sa sljedećim skupom ugrađenih operatora pridruživanja dobrote:

- FitAssign_ScaleObjValue – operator primjenjiv na jednokriterijske probleme kod kojega je fitnes proprocionalan skaliranoj vrijednosti funkcije cilja
- FitAssign_LinearRanking – operator primjenjiv na jednokriterijske probleme kod kojega se jedinke rangiraju te se pridružuje fitnes proporcionalan rangu jedinke
- SPEA_FitAssign – operator primjenjiv na višekriterijske probleme[Zitzler2001]
- CHNA_FitAssign, CH_I1_FitAssign i CH_I2_FitAssign – operatori primjenjivi na višekriterijske probleme s ograničenjima [Kurpati2002]

***Ugrađeni operatori selekcije***

Osnovni zadatak operatora selekcije je da na osnovu pridruženih vrijednosti fitnesa selektiraju skup jedinki za daljnju reprodukciju. U ESOP okruženje su ugrađeni sljedeći operatori selekcije:

- RouletteWheelSelectionFitness – operator selekcije koji obavlja selekciju jedinki s vjerojatnošću selekcije koja je analogna vjerojatnosti odabira određenog broja na kotaču ruleta (engl. *roulette wheel selection*)
- StohasticUniversalSamplingSelection – operator selekcije koji radi na principu stohastičkog uzorkovanja

***Ugrađeni operatori križanja i mutacije***

Pomoću operatora križanja i mutacije genetički algoritmi kreiraju nova rješenja s ciljem što boljeg pretraživanja prostora rješenja i nalaženja što boljih rješenja optimizacijskog problema. S obzirom da izgradnja tih operatora ovisi o korištenoj reprezentaciji ulazne varijable, za svaku vrstu varijabli je definiran prikladan skup operatora koji su standardno ugrađeni u ESOP okruženje.

Skup ugrađenih operatora za binarnu reprezentaciju (BitArray) je sljedeći:

- BitArray1PointCrossover
- BitArray2PointCrossover
- BitArrayUniformCrossover
- BitArrayMutationFlopBit

Kod ugradnje operatora za cjelobrojnu reprezentaciju je iskorištena činjenica da cjelobrojne varijable imaju dobro definiranu i pripadnu binarnu reprezentaciju, pa je za njih definiran i skup operatora križanja i mutacije koji operira nad njihovom pripadnom binarnom reprezentacijom.

- Integer_1PointCrossover
- Integer_2PointCrossover
- Integer_UniformCrossover
- Integer_ArithmeticRecombinationCrossover
- Integer_FlipBitMutation

- Integer_RandomRessetingMutation
- Integer_CreepMutation

Operatori koji djeluju nad poljem cjelobrojnih varijabli (IntegerArray) se oslanjaju na implementacije odgovarajućih operatora za pojedinačnu cjelobrojnu varijablu:

- IntegerArray_1PointCrossover
- IntegerArray_2PointCrossover
- IntegerArray_UniformCrossover
- IntegerArray_ArithmeticRecombinationCrossover
- IntegerArray_ExchangeCrossover
- IntegerArray_RandomRessetingMutation
- IntegerArray_CreepMutation

Operatori za realnu reprezentaciju (Real):

- Real_IntermediateRecombinationCrossover
- Real_LineRecombinationCrossover
- Real_RandomUniformMutation
- Real_GaussianMutation

Operatori za realne varijable reprezentirane poljem bitova (RealAsBitArray):

- RealAsBitArray_1PointCrossover
- RealAsBitArray_2PointCrossover
- RealAsBitArray_UniformCrossover
- RealAsBitArray_FlopBitMutation

Operatori za polje realnih varijabli (RealArray):

- RealArray_IntermediateRecombinationCrossover
- RealArray_LineRecombinationCrossover
- RealArray_ExchangeCrossover
- RealArray_GaussianMutation

Operatori za polje realnih varijabli reprezentiranih poljem bitova (RealArray_AsBitArray):

- RealArray_AsBitArray_1PointCrossover
- RealArray_AsBitArray_2PointCrossover
- RealArray_AsBitArray_UniformCrossover
- RealArray_AsBitArray_FlopBitMutation

Operatori za reprezentaciju permutacijom:

- PermutationPMXCrossover
- PermutationCycleCrossover
- PermutationOrderCrossover
- PermutationEdgeCrossover
- PermutationSwapMutation
- PermutationInsertMutation
- PermutationScrambleMutation

#### 5.3.2.3.3 Ugradnja algoritma simuliranog kaljenja

Iz teorijskog opisa algoritma simuliranog kaljenja danog u odjeljku 2.2.3. proizlaze sljedeće karakteristike:

- algoritam kreće s nekom početnom vrijednošću parametra temeprature
- za svaku vrijednost parametra temperature se provodi određeni broj rekonfiguracija sistema, odnosno, generiraju se rješenja iz okoline trenutnog rješenja
- ukoliko je novo rješenje bolje od trenutnog (ima manju „energiju“), ono se prihvaća kao novo rješenje
- ukoliko je novo rješenje lošije od trenutnog, prihvaća se kao novo rješenje s određenom vjerojatnošću definiranom Boltzmannovom raspodjelom
- nakon provođenje određenog broja rekonfiguracija na danoj temepraturi, temperatura se smanjuje po nekom rasporedu kaljenja
- cilj optimizacije je minimizirati „energiju“ sistema

Iz ovog skupa karakteristika proizlaze zahtjevi na programsku ugradnju razreda / objekta koji će predstavljati optimizacijski postupak simuliranog kaljenja, i oni su sljedeći:

- mora postojati način za određivanje početne temeprature sistema
- mora postojati način za generiranje rješenja iz okoline trenutnog rješenja
- na osnovu karakteristika trenutnog rješenja se mora moći proračunati „energija“ danog rješenja, koja je prorporcionalna s kvalitetom rješenja
- mora se moći definirati trajanje optimizacije na određenoj temperaturi, odnosno, potrebno je moći definirati koliko će se rekonfiguracija sistema obaviti na zadanoj temepraturi
- mora postojati mogućnost za zadavanje rasporeda kaljenja kojim je definiran način smanjivanja temperature tijekom provođenja optimizacije

Skup zajedničkih karakteristika algoritma simuliranog kaljenja je realiziran definiranjem apstraktnog razreda BaseSA sa sljedećom definicijom:

```
public ref class BaseSA abstract :    public IterativeAlgorithm
{
public:
        BaseSA();

        virtual   float     InitialTemperature() = 0;
        virtual   float     NextTemperature() = 0;
        virtual   bool      IsEquilibriumReached() = 0;
        virtual   double    GetCurrEnergy(ISolution ^Sol) = 0;

        virtual   int       Initialize() override;
        virtual   int       PerformIteration() override;

        void      setNeighbourhodGen(int Index, INeighbourhoodGeneratorOperator ^inNeighGen);

protected:
        ISolution^        _CurrSolObj;
        ISolution^        _BestSolObj;
```

```
        OperatorContainer^      _NeighbourSolGenerators;

        Random ^        _rndGen;
        double          _CurrTemp;
};
```

*Programski odsječak 36: Deklaracija razreda BaseSA*

Razredi koji će implementirati konkretne postupke simuliranog kaljenja će biti izvedeni iz ovog razreda. Ugradnjom navedene četiri čiste virtualne funkcije će se odrediti karakteristike konkretnog postupka simuliranog kaljenja.

Unutar razreda BaseSA su ugrađene funkcije Initialize() i PerformIteration() koje se u svom radu oslanjaju na implementaciju funkcija iz sučelja u izvedenim razredima. Funkcija Initialize() obavlja jednostavnu inicijalizaciju početnog rješenja i početne temperature, dok funkcija PerformIteration() predstavlja generičku implementaciju jedne iteracija postupka simuliranog kaljenja. Njihova ugradnja je prikazana u sljedećem programskom odsječku:

```
int  BaseSA::Initialize() {
    _CurrSolObj->GenerateRandom(_rndGen);
    _CurrSolObj->UpdateProblemOutput();
    _BestSolObj = dynamic_cast<ISolution ^>(_CurrSolObj->Clone());
    _CurrTemp = InitialTemperature();
    return 1;
}
int  BaseSA::PerformIteration()
{
    while( IsEquilibriumReached() == false )
    {
        SolutionSO      ^WorkSol = dynamic_cast<SolutionSO ^>(_CurrSolObj->Clone());

        for( int j=0; j<GetProblem()->getProblemInputDescription()->getNumDesignVar(); j++ ) {
            INeighbourhoodGeneratorOperator      ^NGenerator =
                        dynamic_cast<INeighbourhoodGeneratorOperator ^>
                                    (_NeighbourSolGenerators->getOperator(j));

            NGenerator->GenerateNeighbour(_CurrSolObj->getProblemInput()->GetDesignVariable(j),
                                          WorkSol->getProblemInput()->GetDesignVariable(j));
        }
        WorkSol->UpdateProblemOutput();

        double  dE = GetCurrEnergy(WorkSol) - GetCurrEnergy(_CurrSolObj);
        bool    bAccept = false;

        if( dE < 0 )
            bAccept = true;
        else if( (_rndGen->Next() % 10000) / 10000.0 < Math::Exp(-dE / _CurrTemp) )
            bAccept = true;

        if( bAccept == true )
            _CurrSolObj = WorkSol;
    }
    _CurrTemp = NextTemperature();
};
```

*Programski odsječak 37: Ugradnja funkcija Initialize() i PerformIteration() u BaseSA razredu*

Kako je tijekom izrade ove disertacije i ESOP optimizacijskog okruženja naglasak stavljen na genetičke algoritme, unutar ESOP-a je ugrađen samo jedan konkretan razred za optimizaciju SO problema pomoću simuliranog kaljenja. Korištena je najjednostavnija ugradnja postupka simuliranog kaljenja gdje se unutar svake iteracije (znači na svakoj temperaturi) odrađuje preddefiniran broj rekonfiguracija, a sama temperatura se nakon svake iteracije smanjuje množenjem sa zadanom konstantom manjom od 1. Energija sistema je jednaka vrijednosti funkcije cilja.

## 5.3.3 Skup pomoćnih ugrađenih komponenti

Pored već opisanih razreda pomoću kojih su u ESOP optimizacijskom okruženju predstavljeni optimizacijski problemi, reprezentacije ulaznih varijabli, optimizacijski postupci i njihovi operatori, u ESOP okruženje je ugrađen i skup razreda koji predstavlja pomoćne komponente koje nisu direktno vezane uz provođenje optimizacije i čija namjena je vizualizacija dobivenih rezultata optimizacije.

U poglavlju 5.3.1.5 je opisan skup razreda koji predstavljaju rezultate i međurezultate optimizacije. Osnovni razred je OptimizationResult koji predstavlja konačni rezultat optimizacije pomoću optimizacijskog algoritma i koji sadrži podatke o završnom rezultatu optimizacije (razred sučelja IFinalResult i pripadne SO, SOWC, MO i MOWC specijalizacije) i podatke o generiranim međurezultatima (predstavljenim preko ObservedResultCollection razreda). Unutar tih razreda je ugrađena funkcionalnost za njihov prikaz u obliku stringa, što omogućava vrlo jednostavan ispis rezultata na konzolu nakon provedene optimizacije. Međutim, kod izgradnje aplikacije s vizualnim sučeljem ugrađena funkcionalnost će biti od male koristi s obzirom da takve aplikacije koriste bitno drugačiju U/I paradigmu (ugrađena funkcionalnost je prikladna i dovoljna samo u slučaju izgradnje programskog rješenja za optimizaciju u vidu konzolne aplikacije).

Pored neprikladnosti vizualizacije rezultata u jednom stringu, dodatni problem je vizualizacija generiranih Pareto skupova kod višekriterijske optimizacije. U slučaju ispisa rezultata na konzoli vizualizaciju Pareto skupa je jedino moguće postići ispisom dobivenih *n*-torki vrijednosti funkcija cilja za pojedine članove generiranog Pareto skupa. Neprikladnost takvog pristupa u slučaju problema s dvije, tri ili više funkcija cilja kada generirani Pareto skupovi u *n*-dimenzionalnom prostoru tvore složene hiperpovršine je očita. Naime, i u slučaju višekriterijske optimizacije se mora odabrati jedno konačno rješenje problema (iz dobivenog skupa Pareto rješenje), koje istraživač mora odabrati na osnovu svog iskustva i u koordinaciji sa stručnjacima koji dubinski poznaju problem koji se rješavao. Mogućnost vizualizacije dobivenih Pareto ploha u 2D ili 3D prostoru stoga predstavlja značajno olakšanje za istraživača prilikom odabira konačnog rješenja optimizacije.

### 5.3.3.1 Ugrađene komponente za vizualizaciju rezultata

Kod ugradnje komponenti za vizualizaciju rezultata iskorištena je mogućnost .NET razvojnog okruženja da se i vizualne programske komponente (poznate *forme*, odnosno *dijalozi* u Windows operacijskom sustavu) također grade kao razredi unutar aplikacije koja ih definira. Prije pojave .NET okruženja, vizualne komponente su se u Windows operacijskom sustavu, pod uvjetom da govorimo o korištenju Visual

Studio razvojnog okruženja, uglavnom u aplikacije ugrađivale definiranjem sadržaja *resource* datoteka koje su opisivale izgled vizualnog sučelja (korištene kontrole, njihove pozicije unutar forme) a koje su s ostatkom aplikacije komunicirale na (za programera) netransparentan način. Odnosno, izgrađene vizualizacijske komponente nisu bile zatvorene i dobro definirane unutar izgrađene aplikacije jer su se za svoje ispravno funkcioniranje oslanjale na funkcionalnost koju je, bez potrebe (i znanja) programera, u izvršni kôd aplikacije ugrađivalo samo razvojno okruženje.

U .NET okruženju je koncept forme postavljen na posve drugačiji način (bitno je napomenuti da su razvojna okruženja bazirana na Java programskom jeziku takav pristup usvojila od samih početaka). Forma, odnosno dijalog, se u .NET aplikaciju ugrađuje kao novi razred, koji je izveden iz apstraktnog razreda Form (definiranog u System.Windows.Forms dll-u i istovjetnom prostoru imena) koji ima ugrađenu potrebnu infrastrukturu koju graditelj konkretne forme koristi prilikom definiranja izgleda i funkcionalnost svog vizualnog sučelja.

Time je omogućeno da se izgrađene forme stave unutar biblioteke razreda i iskorištavaju kao neovisne programske komponente, ovisne jedino o postojanju System.Windows.Forms.dll biblioteke razreda, koja je podrazumijevani dio .NET izvršnog okruženja i koje mora biti instalirano na računalo prilikom pokretanja .NET aplikacija. Izgrađeni razredi formi se koriste kao i svi drugi razredi. Potrebno je najprije instancirati objekt iz odgovarajućeg razreda koji definira formu i zatim nad tako kreiranim objektom pozvati člansku funkciju Show(), i forma se prikazuje na ekranu.

Kod izgradnje formi za vizualizaciju rezultata, iskorištena je i ugrađena podrška za nasljeđivanje razreda formi. S obzirom da se forme u .NET okruženju grade kao zasebni razredi, postojanje te mogućnosti izgleda sasvim logično, iako u odnosu na mogućnost pred-.NET razvojnih okruženja izgleda kao znanstvena fantastika. Uspostavljanjem odnosa nasljeđivanja između dva razreda, izvedeni razred nasljeđuje cjelokupnu funkcionalnost ugrađenu u bazni razred. Kod običnih razreda to znači da sve članske varijable i funkcije definirane u baznom razredu postaju dio izvedenog razreda. Kod razreda formi to znači da sve vizualne kontrole koje su definirane na baznoj formi postaju dio izvedene forme, zajedno s njihovim ponašanjem koje je definirano preko događaja (engl. *events*) na koje one reagiraju. Prirodno se podrazumijeva da izvedena forma može modificirati tako naslijeđeno vizualno sučelje dodavanjem svog specifičnog skupa kontrola i njihovog pripadnog ponašanja.

Oslanjajući se na gore opisane mogućnosti .NET okruženja, kao dio ESOP optimizacijskog okruženja je izgrađena biblioteka razreda ResultsVisualization-Controls unutar koje je ugrađen skup razreda za vizualno prikazivanje rezultata optimizacije.

#### 5.3.3.1.1 Forme za vizualizaciju rezultata optimizacije

Osnovni razred u biblioteci ResultsVisualizationControls je razred frmOptimizationResult i namjena forme koju on ugrađuje je vizualizacija OptimizationResults objekta koji predstavlja završni rezultat optimizacije. Odnosno, preciznije bi bilo reći da izgrađena forma vizualizira podatke o rezultatu optimizacije koji su zajednički za sve vrste rezultata. Taj zajednički skup podataka obuhvaća

podatak o imenu problema koji je optimiran, korištenom algoritmu, trajanju optimizacije i broju obavljenih iteracija.

Međutim, iz definicije razreda OptimizationResult je vidljivo da je konačni rezultat optimizacije modeliran referencom na objekt koji zadovoljava IFinalResult sučelje, a konkretni objekti na koje ta referenca u stvarnosti pokazuje su ResultSO, ResultSOWC, ResultMO ili ResultMOWC tipa. Kako bi se omogućio vizualni prikaz podataka o rezultatu koji su specifični za pojedinačnu vrstu rezultata (npr. vizualizacija rezultata optimizacije jednokriterijskog problema bez ograničenja podrazumijeva prikaz postignute vrijednosti cilja, dok vizualizacija rezultata optimizacije višekriterijskog problema podrazumijeva grafički prikaz nađene Pareto plohe), definiran je skup razreda formi (frmResultSO, frmResultSOWC, frmResultMO i frmResultMOWC) koje su izvedene iz frmOptimizationResult razreda.

Nasljeđivanjem ovih razreda iz frmOptimizationResult razreda, oni preuzimaju cjelokupni izgled definiranog vizualnog sučelja, a na površinu forme dodaju odgovarajući skup kontrola preko kojih se vizualiziraju podaci specifični za svaku pojedinačnu vrstu rezultata.

Pored korištenja standardnih Windows vizualnih kontrola (TextBox, ListControl kontrole), za vizualizaciju Pareto skupa su izgrađena dvije korisničke kontrole (engl. *user control*), namijenjene vizualizaciji 2D i 3D Pareto skupova. Za realizaciju tih kontrola je iskorištena NevronChart kontrola [Nevron2006], koja predstavlja profesionalnu kontrolu za vizualizaciju različitih grafova (koja je kod izgradnje ESOP-a iskorištena u evaluacijskoj verziji kod koje se prilikom vizualizacije na ekran transparentnim fontom preko grafa iscrtava upozorenje da se radi o evaluacijskoj verziji, što se i vidi na prikazanim slikama ekranskih zaslona).

S stajališta programske implementacije, frmOptimizationResult kontrola i njene specijalizacije su vrlo jednostavne. Osnovna je funkcija SetResult(OptimizationResult inRes) pomoću koje se formi predaje referenca na OptimizationResult objekt čiji sadržaj treba vizualizirati i u kojoj se obavlja ažuriranje prikaza vizualnih kontrola u skladu sa sadržajem prenesenog OptimizationResult objekta.

Forme za prikaz rezultata jednokriterijske optimizacije prikazuju nađenu vrijednost funkcije cilja i vrijednosti skupa ulaznih varijabli a ukoliko se radi o SOWC problemu se prikazuju i postignute vrijednosti ograničenja. Kod vizualizacije MO i MOWC rezultata se umjesto toga koriste kontrole za vizualizaciju Pareto skupa rješenja. Izgleda ResultSO forme se može vidjeti na slici 5.5 (u sljedećem odjeljku koji opisuje primjenu ESOP-a na rješavanje problema optimizacije realnih funkcija), a izgled ResultMOWC kontrole s pripadnom vizualizacijom Pareto skupa je prikazan na slici 5.11.

#### 5.3.3.2 Prikaz međurezultata

Dio razreda OptimizationResult čini i kolekcija međurezultata koji su prikupljeni tijekom optimizacije koja je programski realizirana kao polje referenci na ObservedValuesCollection objekte. Kako je rukovanje s međurezultatima neovisno o vrsti optimizacije (SO, SOWC, MO ili MOWC), realizacija funkcionalnosti povezane s vizualizacijom tih međurezultata je ugrađena u sam frmOptimizationResult razred.

Kolekcija prikupljenih međurezultata je u frmOptimizationResult formi vizualizirana preko ListControl kontrole a iz slike ekranskog zaslona (slika 5.5) se vidi način te vizualizacije. Svaka generirana kolekcija međurezultata predstavlja jedan redak u List kontroli i za svaku kolekciju se ispisuje njen naziv (koji je definiran prilikom kreiranja objekta kolekcije), broj prikupljenih međurezultata i vrsta (tip) međurezultata koji se nalaze u kolekciji.

S obzirom da se u kolekcijama međurezultata mogu nalaziti različiti tipovi podataka, kako je opisano u poglavlju 5.3.1.4.1, potrebno je postaviti arhitekturu koja će omogućiti njihov prikaz. Najprije je bitno uočiti da postoje dvije mogućnosti vizualizacije međurezultata. Može se vizualizirati cijela kolekcija međurezultata istovremeno, a druga mogućnost je vizualizacija jednog po jednog međurezultata iz kolekcije. Prva situacija se npr. javlja kod vizualizacije prikupljenih međurezultata o prosječnoj vrijednosti dobrote u populaciji kod genetičkog algoritma kada želimo vidjeti kretanje te prosječne vrijednosti po iteracijama. Vizualizacija takvog skupa međurezultata ustvari predstavlja prikaz realne funkcije gdje se na *x* osi nalazi broj iteracija, a na *y* osi se nalazi vrijednost koju pratimo.

Druga mogućnost je, npr. potrebna ukoliko se želi promatrati poboljšavanje pronađene Pareto plohe kod višekriterijske optimizacije. S obzirom da je istovremena vizualizacija više Pareto skupova na ekran nepraktična, najbolje rješenje je omogućiti korisniku vizualizaciju pojedinačnih Pareto skupova s mogućnošću odabira međurezultata (Pareto skupa) koji se želi prikazati.

U skladu s ove dvije mogućnosti su izgrađene i dvije generičke forme za vizualizaciju međurezultata. Forma frmShowObservedCollComplete služi za prikaz cijele kolekcije međurezultata, dok forma frmObservedCollIndividual služi za prikaz pojedinačnog međurezultata, a arhitektura programskog rješenja je sljedeća.

Forma frmOptimizationResult preko *pop-up* menija omogućava selekciju pojedine kolekcije međurezultata u List kontroli i odabir da li se želi vizualizirati cijela kolekcija ili njene pojedinačne elemente. U skladu s time odabirom se kreira odgovarajuća frmShowObservedColl... forma kojoj se prenosi referenca na IObservedValuesCollection objekt gdje se nalaze međurezultati za vizualizaciju, nakon čega navedene forme preuzimaju kontrolu tijeka obavljanja vizualizacije.

S obzirom da je skup međurezultata koji se mogu pratiti tijekom rada optimizacijskog postupka otvoren, pod čime se misli na činjenicu da će se s definiranjem novih postupaka pojaviti potreba za vizualizacijom novih vrsta međurezultata, potrebno je omogućiti ugradnju novih programskih komponenti za vizualizaciju tih novih vrsta međurezultata. Kako bi se to postiglo, dvije navedene forme su generičke i za obavljanje same vizualizacije se oslanjaju na skup drugih programskih komponenti koje su realizirane kao korisničke kontrole (engl. *user control*).

Po već isprobanom principu proširivanja, unutar ResultVisualizationControls biblioteke su definirana dva razreda sučelja:

```
public interface ICompleteCollVisualizationControl
{
    Type    GetRequestedMemberType();
    void    ShowCollection(IObservedValuesCollection inColl);
}
```

```
public interface IIndividualCollMemberVisualizationControl
{
    Type    GetRequestedMemberType();
    void    Show(Object obj);
}
```

Sučelje ICompleteCollVisualizatioControl moraju zadovoljiti korisničke kontrole predviđene za vizualizaciju cijele kolekcije međurezultata, dok sučelje IIndividualCollMemberVisualizationControl zadovoljavaju kontrole koje vizualiziraju pojedinačni međurezultat iz kolekcije. Preko ugradnje funkcije GetRequestedMemberType() korisnička kontrola kazuje koji tip međurezultata zna vizualizirati, što je bitno s obzirom da se podaci prenose po referenci na bazni razred. Sama vizualizacija se obavlja pozivom funkcije ShowCollection() odnosno Show(). Ovdje je bitno primjetiti da je kod funkcije Show() primjenjena tehnika slabog tipiziranja (engl. *weak typing*) gdje se objekt koji sadrži međurezultat za vizualizaciju prenosi preko reference na Object razred.

Kao primjer izgradnje korisničke kontrole za vizualizaciju kolekcije realnih brojeva, u sljedećem programskom odsječku se mogu vidjeti detalji izgradnje kontrole ctrlRealValueCollection_Func2D_Nevron koja služi za vizualizaciju kolekcije realnih brojeva:

```
public class ctrlRealValueCollection_Func2D_Nevron : UserControl,  ICompleteCollVisualizationControl
{
   private IObservedValuesCollection _Coll;
   private NChart m_Chart;
   private NLineSeries m_Line;

   public Type GetRequestedMemberType() {
     RealObsValue a = new RealObsValue();
     return a.GetType();
   }

   public void ShowCollection(IObservedValuesCollection inColl) {
     _Coll = inColl;

     // postavljanje parametara Nevron chart kontrole
     m_ChartControl.Settings.RenderDevice = RenderDevice.GDI;
     ...

     // dodavanje vrijednosti u graf
     for (int i = 0; i < _Coll.NumValues; i++)  {
       IObservedValueTag tag = (IObservedValueTag)_Coll.GetValueTag(i);
       RealObsValue val = (RealObsValue)_Coll.GetValue(i);

       m_Line.AddXY(val._Value, tag.getIterNum());
     }
     m_ChartControl.Refresh();
   }
 }
```

Gore opisana korisnička kontrola predstavlja dobro zatvorenu programsku komponentu koja se može kreirati na proizvoljnoj formi i koja je u stanju vizualizirati kolekciju međurezultata ukoliko se u toj kolekciji nalaze objekti tipa RealObsValue. Izgradnja kontrola za vizualizaciju drugačijih tipova podataka je

ekvivalentna, a i izgradnja kontrola za vizualizaciju pojedinačnih međurezultata slijedi isti obrazac.

Još se postavlja pitanje načina na koji forme frmShowObservedCollComplete i frmShowObservedCollIndividual kreiraju korisničku kontrolu koja može prikazati kolekciju međurezultata, odnosno pojedinačni međurezultat optimizacije. Ta funkcionalnost je ugrađena u poseban razred, VisualizationControlsManager koji sadrži popis svih definiranih korisničkih kontrola za vizualizaciju međurezultata. Deklaracija razreda je sljedeća:

```
public class LoadedVisControl
{
    public string Namespace;
    public string DLLName;
    public string ControlName;

    public Type _Type;
    public Type _RequestedCollMemeberType;
}
public class VisualizationControlsManager
{
   public ArrayList          _arlCompleteVisCtrl = new ArrayList();
   public ArrayList          _arlIndividualMemberVisCtrl = new ArrayList();

   public VisualizationControlsManager()
   {
       RealObsValue        a = new RealObsValue();
       RealArrayObsValue  b = new RealArrayObsValue();

       ICompleteCollVisualizationControl  ctrl1 = new ctrlRealValueCollection_Func2D_Nevron();
       ICompleteCollVisualizationControl  ctrl2 = new ctrlRealArrayValueCollection_Func2D_Nevron();

       LoadedVisControl LoadCtrl1 = new LoadedVisControl("Namespace", "DLLName",
                           "ctrlRealValueCollection_Func2D_Nevron", ctrl1.GetType(), a.GetType());
       LoadedVisControl LoadCtrl2 = new LoadedVisControl("Namespace", "DLLName",
                          "ctrlRealValueCollection_Func2D_Nevron", ctrl2.GetType(), b.GetType());

      _arlCompleteVisCtrl.Add(LoadCtrl1);
      _arlCompleteVisCtrl.Add(LoadCtrl2);
   }
   // osnovna funkcija kojom se učitavaju novodefinirane korisničke kontrole iz drugih dll-ova
   public bool LoadVisualizationControls(String filePath)
   {
      Assembly a = Assembly.LoadFrom(filePath);
      if (a == null)          return false;

      Type[] mytypes = a.GetExportedTypes();
      foreach (Type t in mytypes)
      {
         TypeFilter myFilter = new TypeFilter(MyInterfaceFilter);

         string s = "ESOP.ResultsVisualizationControls.ICompleteCollVisualizationControl";
         Type[] myInterfaces = t.FindInterfaces(myFilter, s);
         if (myInterfaces.Length > 0)
         {
              object obj = Activator.CreateInstance(t);
```

```
                Type reqType = (Type)t.InvokeMember("GetRequestedMemberType",
                BindingFlags.DeclaredOnly |
                BindingFlags.Public | BindingFlags.NonPublic |
                BindingFlags.Instance | BindingFlags.InvokeMethod, null, obj, null);

                LoadedVisControl ctrl = new LoadedVisControl(t.Namespace, filePath, t.Name, t, reqType);

               _arlCompleteVisCtrl.Add(ctrl);
            }
            // isto tako za IIndividualCollMemberVisualizationControl
          }
          return true;
        }

      // funkcije koje vraćaju skup kontrola koje znaju vizualizirati međurezultate zadanog tipa
      public ArrayList getAvailableCompleteCtrlForType(Type inType);
      public ArrayList getAvailableIndividualCtrlForType(Type inType);
}
```

Razred VisualizationControlsManager u poljima _arlCompleteVisCtrl i _arlIndividualMemberVisCtrl sadrži popis svih učitanih kontrola za vizualizaciju međurezultata. Kontrole se u taj popis mogu dodati ručno, na način kao što je to u konstruktoru obavljeno za ctrlRealValueCollection_Func2D_Nevron i ctrlRealArrayValueCollection_Func2D_Nevron kontrole, a novoizgrađene korisničke kontrole se mogu učitati direktno iz dll-a u kojem su ugrađene, korištenjem funkcije LoadVisualizationControls. Za to učitavanje se razred VisualizationControlsManager oslanja na .NET *Reflection* mehanizam koji će detaljno biti opisan u odjeljku 5.6.1.1.

## 5.4 Korištenje ESOP-a kao biblioteke razreda

U prethodnom potpoglavlju je opisana izgradnja osnova ESOP optimizacijskog okruženja. Izgrađeni okvir za razvoj je zajedno s ugrađenim skupom optimizacijskih komponenti namijenjen iskorištavanju na nivou izvornog koda i u ovom potpoglavlju će se dati prikaz primjene na rješavanje nekoliko različitih vrsta optimizacijskih problema.

Kako je već ranije opisano, korištenje optimizacijskog okruženja na nivou izvornog kôda podrazumijeva izgradnju posebne aplikacije koja se povezuje s izgrađenim bibliotekama razreda i u kojoj se dostupne optimizacijske komponente (odnosno razredi) programski kombiniraju i koriste za rješavanje zadanog optimizacijskog problema. Izgradnja takve aplikacije zahtjeva znanje iz tehnika programiranja što donekle sužava širinu primjene.

Međutim, zahvaljujući naprednim mogućnostima .NET razvojnog okruženja u kojem je izgrađen ESOP, iskorištavanje izgrađenih optimizacijskih komponenti na nivou izvornog kôda je ipak značajno jednostavnije u odnosu na korištenje optimizacijskih okruženja razvijenih u C++ programskom jeziku (ovdje se misli na standardni C++ koji se prevodi direktno u izvršni kod, za razliku od C++/CLI programskog jezika koji predstavlja varijantu C++a u .NET okruženju).

### 5.4.1 Optimizacija realnih funkcija pomoću genetičkih algoritama

U ovom odjeljku će se prikazati razvoj kompletne aplikacije za rješavanje problema optimizacije realnih funkcija pomoću genetičkih algoritama, s osnovnim ciljem prikaza jednostavnosti korištenja ESOP optimizacijskog okruženja.

S obzirom da je primarni fokus ovog odjeljka prikaz osnovnih programskih tehnika koje se primjenjuju kod iskorištavanja optimizacijskih komponenti ugrađenih u ESOP, izgrađena je jednostavna konzolna aplikacija za rješavanje problema optimizacije realnih funkcija. Iako bi se način iskorištavanja ESOP-a mogao prikazati i na primjeru Windows aplikacije s GUI sučeljem (znači, aplikacije s menijima, prozorima, i ostalim standardnim elementima GUI sučelja) kod izgradnje takve vrste aplikacija se mora posvetiti dodatna pažnja programskoj realizaciji tog GUI sučelja, što bi donekle zakrilo navedeni cilj.

Korištenjem *čarobnjaka* (engl. *wizards*) ugrađenih u Visual Studio 2005 razvojno okruženje, izgradnja kostura konzolne aplikacije se svodi na nekoliko klikova mišem, nakon čega je sve spremno za ugradnju programskog koda posvećenog optimizaciji. Za primjer je odabrano rješavanje već opisanog DeJong_F1 test problema, a kompletan programski kôd koji je potrebno napisati kako bi se pomoću genetičkog algoritma riješio navedeni optimizacijski problem je sljedeći:

```
#using "D:\\ESOP\\debug\\ESOP.BaseInterfacesLib.dll"
#using "D:\\ESOP\\debug\\ESOP.BaseObjectsLib.dll"
#using "D:\\ESOP\\debug\\ESOP.BaseResultsLib.dll"
#using "D:\\ESOP\\debug\\ESOP.BaseAlgorithmsLib.dll"
#using "D:\\ESOP\\debug\\ESOP.StandardOptimizationObjectsLib.dll"
```

```
using namespace System;

using namespace ESOP::Framework::Base;
using namespace ESOP::Framework::Representations;
using namespace ESOP::Framework::Results;
using namespace ESOP::Framework::Problems::RealFunctionOptimization;

using namespace ESOP::Framework::Algorithms::GeneticAlgorithms;

using namespace ESOP::Framework::Operators::GeneticAlgorithms;
using namespace ESOP::Framework::Operators::GeneticAlgorithms::Crossover;
using namespace ESOP::Framework::Operators::GeneticAlgorithms::Mutation;
using namespace ESOP::Framework::Operators::GeneticAlgorithms::Selection;

int main(array<System::String ^> ^args)
{
   Random        ^inRnd = gcnew Random();
   int           inIterNum = 100;
   int           inPopSize = 50;
   double        inCrossProb = 1;
   double        inMutProb = 0.05;

   // definiramo problem
   DeJong_F1_TestProblem^   ProbObj = gcnew DeJong_F1_TestProblem();
   ProblemInput^             PI = gcnew ProblemInput(ProbObj->getProblemInputDescription());

   // definiramo koristenu reprezentaciju
   IRealArray     ^DV1    = gcnew RealArray(dynamic_cast<RealArray_DVDescription ^>
                                (PI->getProblemInputDesc()->getDesignVariableDesc(0)));
   PI->AssignDesignVariable(0, DV1);

   // kreiramo objekt algoritma i postavljamo njegove parametre
   SO_GenerationalGA     ^Alg = gcnew SO_GenerationalGA(inRnd);

   Alg->SetProblem(ProbObj);              // povezivanje s optimizacijskim problemom
   Alg->SetProblemInput(PI);              // povezivanje s kreiranim objektom ProblemInput

   Alg->setPopulationSize(inPopSize);
   Alg->setMutationProbability(inMutProb);
   Alg->setCrossoverProbability(inCrossProb);

   // definiramo operatore
   IFitnessAssignment        ^fitAssign = gcnew FitnesAssignment_EqualToObjValue();
   Alg->setFitnessAssignmentOp(fitAssign);

   IFitnessSelectionOperator  ^select = gcnew RouleteWheelSelectionFitness(inRnd);
   Alg->setFitnessSelectionOp(select);

   ICrossoverOperator2To2    ^cross = gcnew RealArray_ArithmeticRecombinationCrossover();
   IMutationOperator          ^mut  = gcnew RealArray_RandomMutation(inRnd);

   Alg->setCrossover(0, cross);
   Alg->setMutation(0, mut);

   // definiramo trajanje optimizacije
   TerminateOnIterationNum  ^optTerm = gcnew TerminateOnIterationNum();
   optTerm->SetParameter(inIterNum);
   Alg->SetOptTerminator(optTerm);
```

```
    // definiramo međurezultate koji će se pratiti
    IObservedValuesCollection        ^newColl = gcnew ObservedValuesCollection
                                                        ("Current obtained objective value");
    Alg->GetResultRef()->AddNewObsCollection(newColl);
    CurrObjectiveValueSaver ^newSaver = gcnew CurrObjectiveValueSaver(newColl, Alg);
    Alg->AddObserver(gcnew IntermedTimerOnEveryIterNum(10), newSaver);

    IObservedValuesCollection        ^newColl1 = gcnew ObservedValuesCollection
                                                    ("Average objective value in population");
    Alg->GetResultRef()->AddNewObsCollection(newColl1);
    AverageObjectiveValueInPopulationSaver ^newSaver1 = gcnew
                                    AverageObjectiveValueInPopulationSaver(newColl1, Alg);
    Alg->AddObserver(gcnew IntermedTimerOnEveryIterNum(10), newSaver1);

    // pokrecemo optimizaciju
    Alg->Run();

    IOptimizationResult ^Res = Alg->GetResultRef();
    Console::WriteLine(Res->GetResultAsString());

    return 0;
}
```

*Programski odsječak 38: Primjena genetičkog algoritma za rješavanje DeJong_F1 problema*

Izlaz koji se na konzoli dobije nakon izvršavanja ovog programa je sljedeći:

```
FINAL SOLUTION:
Objective value : 0,0116011487260151
Variable values :
   Real array : 3 elements, [-5,12 - 5,12] :
        x[0] = 1,09563606380685
        x[1] = 0,961118077138311
        x[2] = 1,03070973949945

INTERMEDIATE VALUES COLLECTIONS:
Collection : Current obtained objective value
       [10,15,625]     - 0,437382567951432
       [20,31,25]      - 0,19794889436337
       [30,31,25]      - 0,716229165007202
       [40,46,875]     - 0,359378549519345
       [50,46,875]     - 0,312092046534883
       [60,93,75]      - 0,29702952721315
       [70,109,375]    - 0,01798508841206404
       [80,109,375]    - 0,0633305103061926
       [90,125]        - 0,0225409719354385

Collection : Average objective value in population
       [10,31,25]      - 1,217522370980054
       [20,46,875]     - 1,157826887119798
       [30,46,875]     - 0,9555892012105271
       [40,46,875]     - 0,8155865796560792
       [50,93,75]      - 0,656057221900361
```

```
[60,93,75]      - 0,45155800273695037
[70,93,75]      - 0,17244146604852
[80,109,375]    - 0,155797844785635
[90,109,375]    - 0,105797843982301
```

Iako je dani programski kôd uz pripadne komentare dovoljno jasan, ipak će se opisati struktura izgrađenog rješenja, s posebnim osvrtom na promjene koje je potrebno provesti prilikom rješavanja drugih instanci problema.

Izgrađeno programsko rješenje se strukturno može podijeliti na dva dijela. Prvi dio je posvećen definiranju korištenih programskih komponenti i njihovom uvozu (engl. *import*) u programski kontekst aplikacije. Do izražaja ovdje dolazi jednostavnost .NET okruženja u kojem je za iskorištavanje funkcionalnosti ugrađene u biblioteku razreda potrebno jedino obaviti referenciranje dane biblioteke. U većini slučajeva se to radi korištenjem „*Add reference*“ funkcije ugrađene u Visual Studio razvojno okruženje pomoću koje se kroz vizualno sučelje odabire biblioteka razreda (dll). Međutim, potpuno istovjetan rezultat se može postići i korištenjem *#using* direktive direktno u programskom kôdu. Taj način je odabran i u ovom primjeru kako bi cjelokupna ugradnja bila vidljiva iz programskog kôda. Odnosno, kod izgradnje ovog primjera se interakcija s razvojnim okruženjem svodi na kreiranje kostura konzolne aplikacije što je doista jednostavna operacija koja ne zahtijeva gotovo nikakvo poznavanje naprednih načina korištenja Visual Studio okruženja a sama izgradnja se dalje svodi na editiranje sadržaja tako kreiranih datoteka izvornog koda. Pored #using direktiva kojima se referenciraju potrebne biblioteke razreda, potrebno je (iako ne i nužno) referencirati i korištene prostore imena (engl. *namespaces*) kako se prilikom korištenja pojedinih razreda ne bi trebalo navoditi njihovo puno ime. Konkretno, korištenjem direktive #using namespace ESOP::Framework::Algorithms::GeneticAlgorithms se omogućava da se kasnije u programskom kodu direktno koristi razred SO_GenerationalGA, umjesto njegovog punog imena ESOP::Framework::Algorithms::GeneticAlgorithms:: SO_GenerationalGA.

Drugi dio izgrađene aplikacije čini realizacija funkcije main() u koju je ugrađen sam programski kôd za provođenje optimizacije. Kôd se sastoji od nekoliko dijelova, posvećenih kreiranju cjelokupnog optimizacijskog konteksta. Najprije se kreira objekt optimizacijskog problema i odgovarajući skup ulaznih varijabli (koji se ovdje sastoji samo od jedne varijable – RealArray s tri člana). Zahvaljujući postojanju konstruktora za RealArray razred koji na osnovu RealArray_DVDescription objekta (koji je dohvaćen iz ProblemInputDescriptor objekta definiranog za DeJong_F1 problem) postavlja parametre za kompletno definiranje korištene reprezentacije (broj elemenata polja i njihov raspon) kod kreiranja skupa ulaznih varijabli je jedino potrebno odabrati korištenu reprezentaciju za varijablu i pridružiti je u kreirani ProblemInput objekt. Korištenje *dynamic_cast* operatora je potrebno zbog toga što getDesignVariableDesc() funkcija vraća referencu na razred sučelja IDesignVariableDescriptor, iz kojega je izveden konkretan razred RealArray_DVDescription).

Nakon kreiranja objekata koji predstavljaju optimizacijski problem i definiraju korištenu reprezentaciju, potrebno je kreirati objekt optimizacijskog algoritma i definirati sve njegove potrebne parametre. U izgrađenom programskom rješenju je kao optimizacijski postupak odabran generacijski genetički algoritam

kojem je potrebno postaviti odgovarajući skup operatora koji će se koristiti kod optimizacije (operatori pridruživanja dobrote, selekcije, križanja i mutacije) i odgovarajući skup parametara samog algoritma (veličina populacije i vjerojatnost obavljanja križanja i mutacije). Pored toga se kreira i odgovarajući objekt za određivanje trajanja optimizacije, u ovom slučaju TerminateOnIterationNum koji definira trajanje optimizacije preko broja iteracija.

Završini dio programskog kôda je posvećen definiranju skupa međurezultata koji će se pratiti tijekom provođenja optimizacije. Način realizacije praćenja međurezultata optimizacije je opisan u ranijim potpoglavljima, a u gornjem programu je ugrađeno praćenje dvije vrste međurezultata: trenutno najbolja postignuta vrijednost funkcije cilja i prosječna vrijednost funkcije cilja za cijelu populaciju. Na sličan način je moguće vrlo jednostavno ugraditi i praćenje ostalih vrsta međurezultata.

Još jedna bitna karakteristika izgrađenog rješenja se vidi iz danog programskog koda a vezana je uz ispis rezultata optimizacije. Naime, sve što je potrebno obaviti kako bi se konačni rezultat optimizacije vizualizirao (odnosno ispisao na ekran zajedno s pripadnim generiranim međurezultatima) je poziv funkcije GetResultAsString() nad dobivenim OptimizationResult objektom, čime se dodatno olakšava posao istraživaču prilikom izgradnje programskog rješenja za provođenja optimizacije.

Zahvaljujući općenitosti izgrađenog ESOP okvira za razvoj i ugrađenom skupu optimizacijskih komponenata, prikazani programski kôd se može vrlo jednostavno modificirati za korištenje drugih optimizacijskih komponenti. Ukoliko se, npr. želi iskoristiti standardna reprezentacija za genetičke algoritme u kojoj se ulazne varijable reprezentiraju poljem bitova, potrebno je samo promijeniti tip objekta kod kreiranja odabrane reprezentacije:

```
// definiramo koristenu reprezentaciju
 RealArray_DVDescription ^d = PI->getProblemInputDesc()->getDesignVariableDesc(0)));
 IRealArray ^Real1 = gcnew RealArray_AsBitArray(d);
 PI->AssignDesignVariable(0, Real1);
```

i prilagoditi odabrane vrste operatora (koje su ovisne o reprezentaciji nad kojom djeluju):

```
ICrossoverOperator2To2 ^cross = gcnew RealArrayAsBitArray_1PointCrossover(inRnd);
IMutationoperator    ^mut = gcnew RealArrayAsBitArray_FlopBitMutation(inRnd);

Alg->setCrossover(0, cross);
Alg->setMutation(0, mut);
```

Ukoliko se želi iskoristiti druga vrsta genetičkog algoritma (npr. *steady-state* genetički algoritam), jedina promjena je kod kreiranja objekta algoritma (jasno, ovo je omogućeno činjenicom da su te dvije vrste algoritma vrlo slične, i dijele isti skup parametara i operatora):

```
SO_SteadyStateGA       ^Alg = gcnew SO_SteadyStateGA(inRnd);
```

Na isti način se mogu promijeniti i odabrani operatori pridruživanja dobrote, selekcije, križanja i mutacije (s time da se kod odabira operatora križanja i mutacije mora voditi računa i o odabranoj reprezentaciji ulaznih varijabli problema).

#### 5.4.1.1 Iskorištavanje ugrađenih komponenti za vizualizaciju rezultata

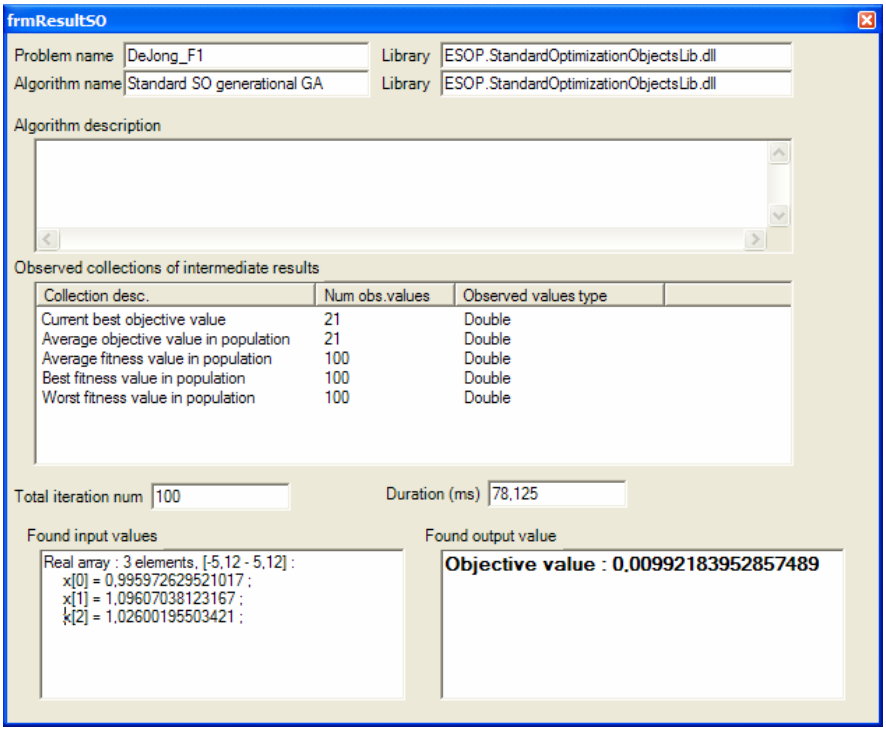
Vrlo jednostavnom modifikacijom gore navedenog programa se dobija prikaz rezultata optimizacije preko ugrađenih komponenti za vizualizaciju. Potrebno je jedino dodatno referencirati biblioteku s ugrađenim vizualizacijskim kontrolama:

```
#using "D:\\ESOP\\debug\\ESOP.ResultVisualizationControls.dll"
```

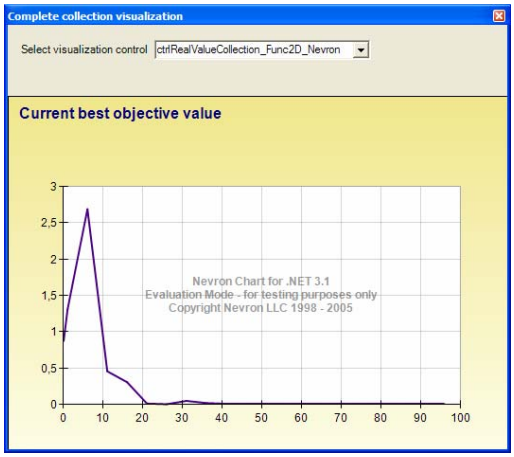
i umjesto ispisa na ekran, nakon provedene optimizacije iskoristiti uvezene vizualizacijske kontrole:

```
frmResultSO     frmRes = new frmResultSO();
frmRes.SetResult(Res);
frmRes.Show();
```

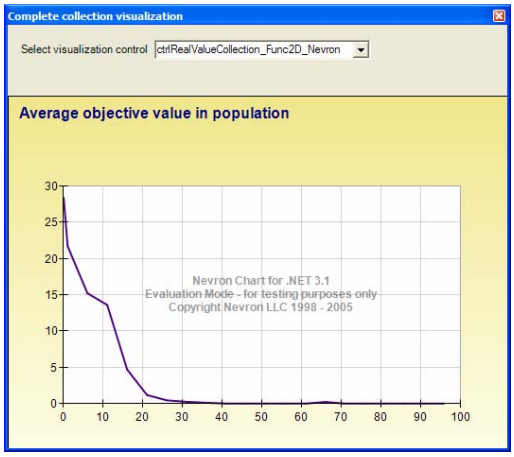
Pozivanjem funkcije Show() za formu frmResultSO se na ekranu prikazuje ugrađena forma za vizualizaciju rezultata jednokriterijske optimizacije (slika 5.5), a klikom na pojedinačne kolekcije međurezultata se otvaraju forme koje vizualiziraju prikupljene međurezultate (slike 5.6 – 5.10). Ovdje je bitno primjetiti da je u ovom slučaju skup međurezultata koji se prati proširen s objektima za praćenje međurezultata vezanih uz dobrotu jedinki u populaciji.



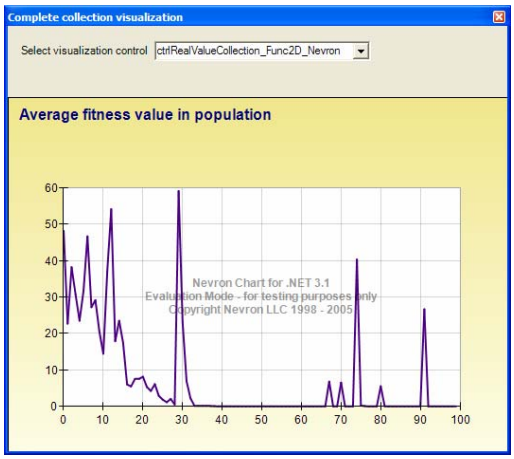

**Slika 5.5 Rezultati optimizacije prikazani pomoću ugrađene forme za vizualizaciju**



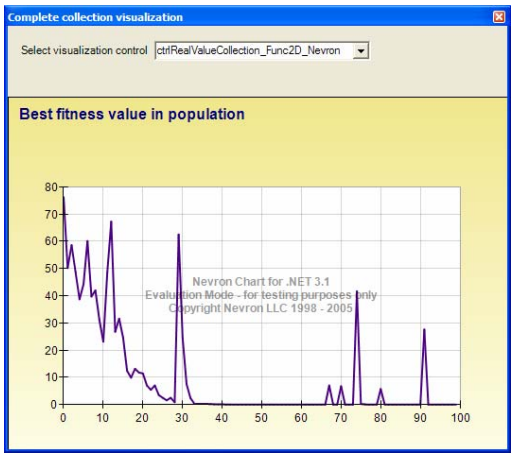

**Slika 5.6 Prikaz nađene najbolje vrijednosti funkcije cilja po iteracijama**



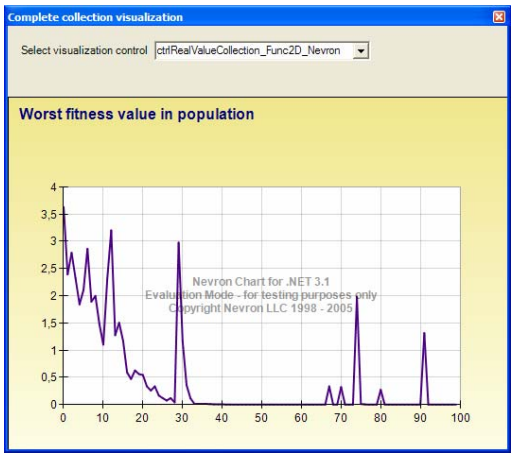

**Slika 5.7 Prikaz prosječne vrijednosti funkcije cilja u populaciji**




**Slika 5.8 Prikaz prosječne vrijednosti fitnesa jedinki u populaciji**




**Slika 5.9 Prikaz fitnesa najbolje jedinke u populaciji**




**Slika 5.10 Prikaz fitnesa najgore jedinke u populaciji**

## 5.4.2 Primjena genetičkih algoritama na rješavanje višekriterijskih problema

Rješavanje višekriterijskih optimizacijskih problema s ograničenjima se u ESOP optimizacijskom okruženju provodi na sličan način kao što je to opisano u prethodnom odjeljku za jednokriterijske probleme. Potrebno je kreirati objekte koji će predstavljati optimizacijski problem, odabranu reprezentaciju, optimizacijski postupak i skup njegovih operatora.

Za prikaz optimizacije je odabran BNH problem sa sljedećom definicijom [Kurpati2002]:

| Problem | Variable bounds | Objectives functions $f(x)$ and Constraints $C(x)$ |
|---|---|---|
| BNH | $x_1 \in [0,5]$<br>$x_2 \in [0,3]$ | $f_1(x) = 4x_1^2 + 4x_2^2$<br>$f_2(x) = (x_1 - 5)^2 + (x_2 - 5)^2$<br>$C_1(x) \equiv (x_1 - 5)^2 + x_2^2 \le 25$<br>$C_2(x) \equiv (x_1 - 8)^2 + (x_2 + 3)^2 \ge 7.7$ |

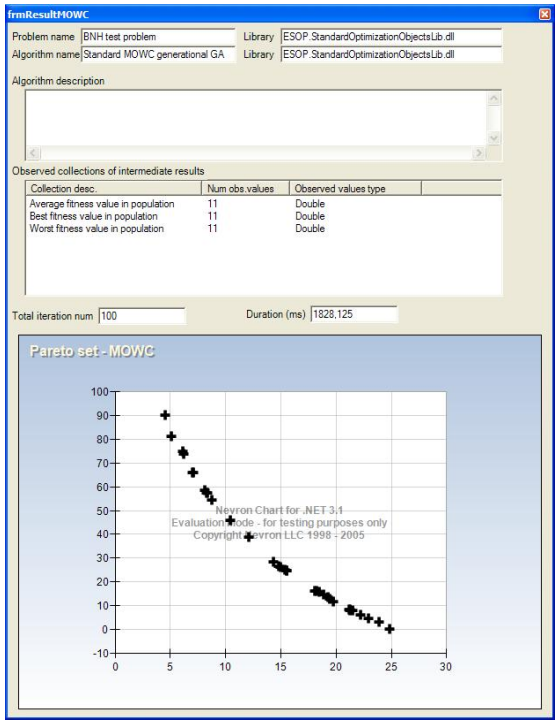
Programsko rješenje je izgrađeno kao konzolna aplikacija, ali korištenjem C# programskog jezika kako bi se ilustrirala mogućnost korištenja ESOP optimizacijskih komponenti i u drugim programskim jezicima u .NET okruženju. S obzirom da je dio programskog kôda namijenjen definiranju referenci na potrebne biblioteke razreda isti kao u primjeru jednokriterijske optimizacije opisane u prethodnom odjeljku, ovdje će se navesti samo dio programskog koda u kojem se postavlja cjelokupni kontekst za provođenje optimizacije:

```
Random      rnd = new Random();
int         IterNum = 100;
int         PopSize = 50;
double      CrossProb = 1;
double      MutProb = 0.05;

// definiramo problem
BNH_TestProblem ProbObj = new BNH_TestProblem();
ProblemInput PI = new ProblemInput(ProbObj.getProblemInputDescription());

// definiramo koristenu reprezentaciju
RealAsBitArray DV1 = new RealAsBitArray (PI.getProblemInputDesc().getDesignVariableDesc(0)
                                                                    as RealArray_DVDescription);
RealAsBitArray DV2 = new RealAsBitArray (PI.getProblemInputDesc().getDesignVariableDesc(1)
                                                                    as RealArray_DVDescription);
PI.AssignDesignVariable(0, DV1);
PI.AssignDesignVariable(1, DV2);

// definiramo algoritam
MOWC_GenerationalGA Alg = new MOWC_GenerationalGA(inRnd, 100);

Alg.SetProblem(ProbObj);
Alg.SetProblemInput(PI);

Alg.setPopulationSize(PopSize);
Alg.setMutationProbability(MutProb);
Alg.setCrossoverProbability(CrossProb);
```

```
// definiramo operatore
ZV_MOWC_FitnessAssignment fitAssign = new ZV_MOWC_FitnessAssignment();
Alg.setFitnessAssignmentOp(fitAssign);

RouleteWheelSelectionFitness select = new RouleteWheelSelectionFitness(Rnd);
Alg.setFitnessSelectionOp(select);

ICrossoverOperator2To2 cross = new RealAsBitArray_1PointCrossover();
IMutationOperator      mut   = new RealAsBitArray_FlopBitMutation(Rnd);

Alg.setCrossover(0, cross);
Alg.setMutation(0, mut);
Alg.setCrossover(1, cross);
Alg.setMutation(1, mut);

TerminateOnIterationNum optTerm = new TerminateOnIterationNum();
optTerm.SetParameter(IterNum);
Alg.SetOptTerminator(optTerm);

// pokrecemo optimizaciju
Alg.Run();

OptimizationResult Res = Alg.GetResultRef() as OptimizationResult;

frmResultMOWC      frmRes = new frmResultMOWC();
frmRes.SetResult(Res);
frmRes.Show();
```

*Programski odsječak 39: Primjena genetičkih algoritama na rješavanje višekriterijskog problema*

Za razliku od programskog rješenja prikazanog u prethodnom odjeljku, ovdje su iskorištene razvijene komponente za vizualizaciju rezultata optimizacije. Izvršavanjem gornjeg programa se na ekranu pojavljuje forma sa vizualizacijom dobivenog Pareto skupa:




**Slika 5.11 Prikaz rezultata višekriterijske optimizacije pomoću forme frmResultMOWC**

## 5.5 Prikaz mogućnosti proširivanja ESOP-a

S obzirom da je mogućnost proširivanja optimizacijskog okruženja s novim optimizacijskim komponentama navedena kao jedan od osnovnih zahtjeva na općenito optimizacijsko okruženje, u ovom poglavlju će se prikazati način na koji ESOP okruženje pruža mogućnost ugradnje novih vrsta optimizacijskih komponenti.

Sam mehanizam proširivanja je s implementacijske strane donekle sličan u prethodnom poglavlju opisanom načinu iskorištavanja postojećih optimizacijskih komponenti. I u jednom i u drugom slučaju se mora izgraditi novo programsko rješenje. Kod iskorištavanja postojećih objekata za optimizacije gradi se aplikacija koja se povezuje s bibliotekama razreda u kojima se nalaze potrebne optimizacijske komponente. U slučaju izgradnje novih optimizacijskih komponenti, potrebno je izgraditi novu biblioteku razreda koja će se povezati s osnovnim bibliotekama razreda ugrađenim u ESOP okvir za razvoj (primarno BaseInterfacesLib, ali i BaseObjectsLib, i eventualno, BaseAlgorithmsLib ukoliko se definira novi optimizacijski algoritam). To uključivanje je potrebno jer novougrađene optimizacijske komponente moraju zadovoljavati definirana sučelja kako bi se mogle iskoristiti unutar ESOP optimizacijskog okruženja.

U sljedećim odjeljcima će se prikazati način definiranja nove vrste optimizacijskog problema i nove vrste operatora. Pritom je bitno naglasiti da su te nove optimizacijske komponente izgrađene kao posve nova biblioteka razreda, na isti način kao što bi to radio bilo koji optimizacijski istraživač prilikom proširivanja ESOP okruženja.

### 5.5.1 Proširivanje ESOP-a s novim problemom

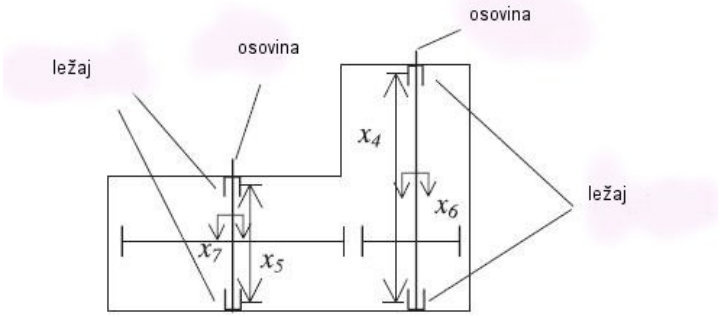
Proširivanje ESOP okruženja s novim optimizacijskim problemom može biti dvojako. Prva mogućnost je ugrađivanje nove instance postojeće vrste optimizacijskog problema, dok druga mogućnost podrazumijeva ugrađivanje posve nove vrste optimizacijskog problema u ESOP okruženje. Prva mogućnost će se prikazati definiranjem nove instance problema optimizacije višekriterijske realne funkcije s ograničenjima, dok će se druga mogućnost prikazati ugrađivanjem kvadratičnog problema pridruživanja (QAP).

#### 5.5.1.1 Definiranje nove instance optimizacijskog problema

Kako je već ranije opisano kod prikaza ugradnje problema optimizacije realnih funkcija koje su standardno ugrađene u ESOP, ugradnja nove instance optimizacijskog problema zahtjeva izgradnju novog razreda izvedenog iz odgovarajućeg apstraktnog IProblem razreda (SO, SOWC, MO ili MOWC) te implementaciju funkcija CalcProblemOutput(), pomoću koje se na osnovu vrijednosti ulaznih varijabli izračunavaju izlazi za dani problem, i funkcije getProblemInputDescriptor() koja definira karakteristike ulaznih varijabli problema.

Kako bi se pokazala jednostavnost izgradnje novog optimizacijskog problema korištenjem Visual Studio razvojne okoline, grafički će se ilustrirati svi koraci potrebni za tu izgradnju. Kao primjer optimizacijskog problema iz prakse je odabran je problem optimizacije parametara mjenjačke kutije koja se može koristiti u laganim

zrakoplovima između motora i propelera [Kurpati2002]. Na slici 5.12 je prikazan izgled mehaničke konstrukcije mjenjačke kutije.




**Slika 5.12 Mehanička konstrukcija mjenjačke kutije**

Problem je definiran kao višekriterijski optimizacijski problem s ograničenjima. Definirano je sedam ulaznih varijabli koje modeliraju odgovarajuće aspekte mehaničke konstrukcije i dvije funkcije cilja. Vrijednost prve funkcije cilja predstavlja volumen, odnosno težinu cjelokupne konstrukcije, dok je vrijednost druge funkcije cilja definirana naprezanjem u jednoj od osovina. Konačni mehanički dizajn mora zadovoljavati ograničenja nametnuta standardima dizajna mjenjačkih kutija i osovina, što je u problemu modelirano definiranjem 11 nelinearnih ograničenja. Pored toga, za svaku od sedam varijabli je definirana donja i gornja granica dopuštenih vrijednosti.

Matematička definicija problema je sljedeća:

$$\text{minimize } f_{\text{weight}} = f_1 =$$

$$0.7854x_1x_2^2(10x_3^2/3 + 14.933x_3 - 43.0934) -$$

$$1.508x_1(x_6^2 + x_7^2) + 7.477(x_6^3 + x_7^3) + 0.7854(x_4x_6^2 + x_5x_7^2)$$

$$\text{minimize } f_{\text{stress}} = f_2 = \frac{\sqrt{(745x_4/x_2x_3)^2 + 1.69 \times 10^7}}{0.1x_6^3}$$

uz definirani skup ograničenja:

$$g_1 : \frac{1}{(x_1x_2^2x_3)} - \frac{1}{27} \le 0\,, \quad g_2 : \frac{1}{(x_1x_2^2x_3^2)} - \frac{1}{397.5} \le 0$$

$$g_3 : \frac{x_4^3}{(x_2x_3x_6^4)} - \frac{1}{1.93} \le 0\,, \quad g_4 : \frac{x_5^3}{(x_2x_3x_7^4)} - \frac{1}{1.93} \le 0$$

$$g_5 : x_2x_3 - 40 \le 0\,, \quad g_6 : \frac{x1}{x_2} - 12 \le 0\,,$$

$$g_7 : 5 - \frac{x_1}{x_2} \le 0\,, \quad g_8 : 1.9 - x_4 + 1.5x_6 \le 0\,,$$

$$g_9 : 1.9 - x_5 + 1.1x_7 \le 0\,, \quad g_{10} : f_1(x) \le 1300\,,$$

$$g_{11} : \frac{\sqrt{(745x_5/x_2x_3)^2 + 1.575 \times 10^8}}{0.1x_7^3} \le 1100\,. \qquad ($$

Granice ulaznih varijabli problema su:

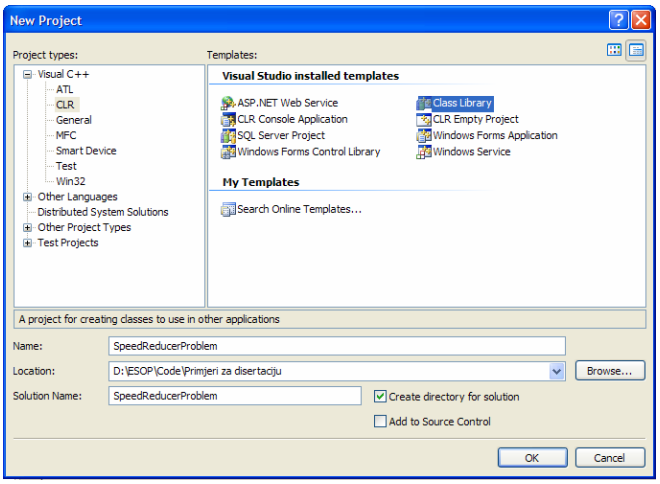
$$g_{12,13};\ 2.6 \le x_1 \le 3.6\,, \quad g_{14,15}:\ 0.7 \le x_2 \le 0.8\,,$$

$$g_{16,17};\ 17 \le x_3 \le 28\,, \quad g_{18,19}:\ 7.3 \le x_4 \le 8.3\,,$$

$$g_{20,21};\ 7.3 \le x_5 \le 8.3\,, \quad g_{22,23}:\ 2.9 \le x_6 \le 3.9\,,$$

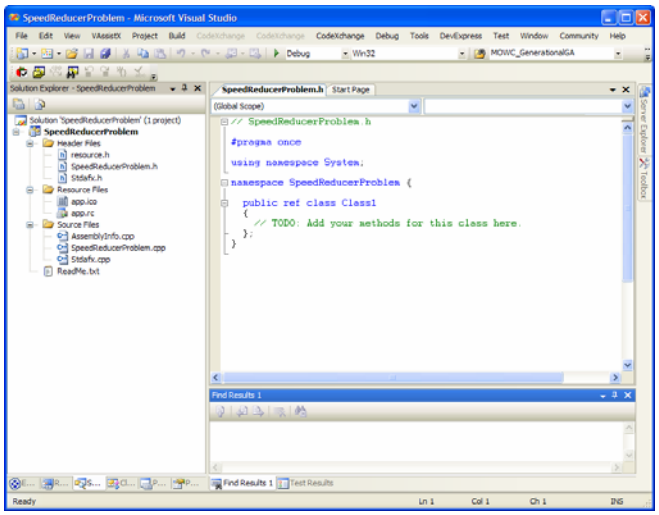
$$g_{24,25};\ 5.0 \le x_7 \le 5.5\,.$$

Ugradnja opisanog problema u ESOP okruženje zahtijeva izgradnju biblioteke razreda unutar koje će biti izgrađen razred koji će predstavljati gore opisani optimizacijski problem. U Visual Studio razvojnom okruženju se novi projekt biblioteke razreda dodaje vrlo jednostavno. Klikom na opciju *File → New → Project* se otvara prozor u kojem je potrebno odabrati vrstu projekta i definirati ime projekta te stazu do direktorija gdje će se smjestiti programski kôd.



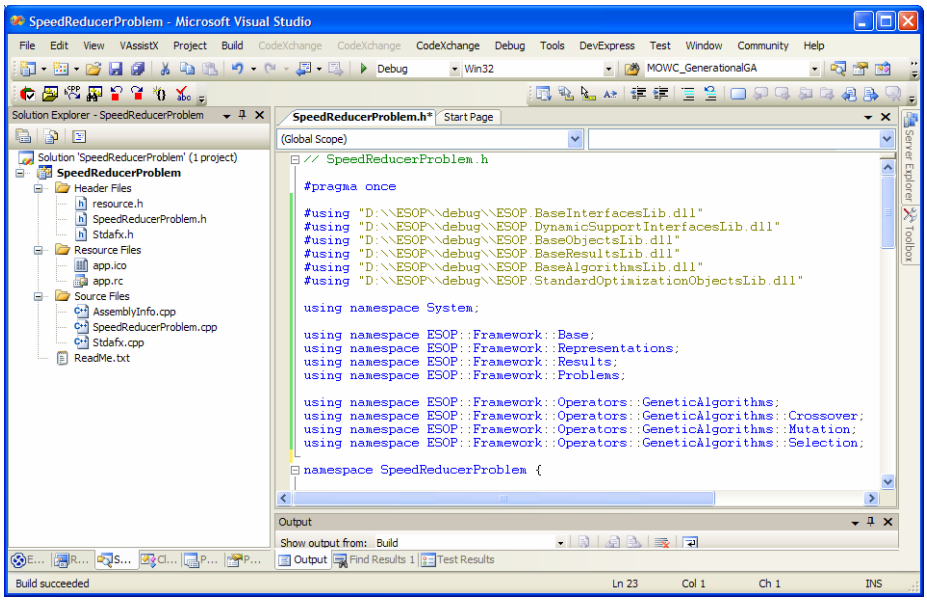

**Slika 5.13 Kreiranje biblioteke razreda u Visual Studio razvojnom okruženju**

Klikom na OK se stvara kompletna infrastruktura za biblioteku razreda, kako je prikazano na slici 5.14.



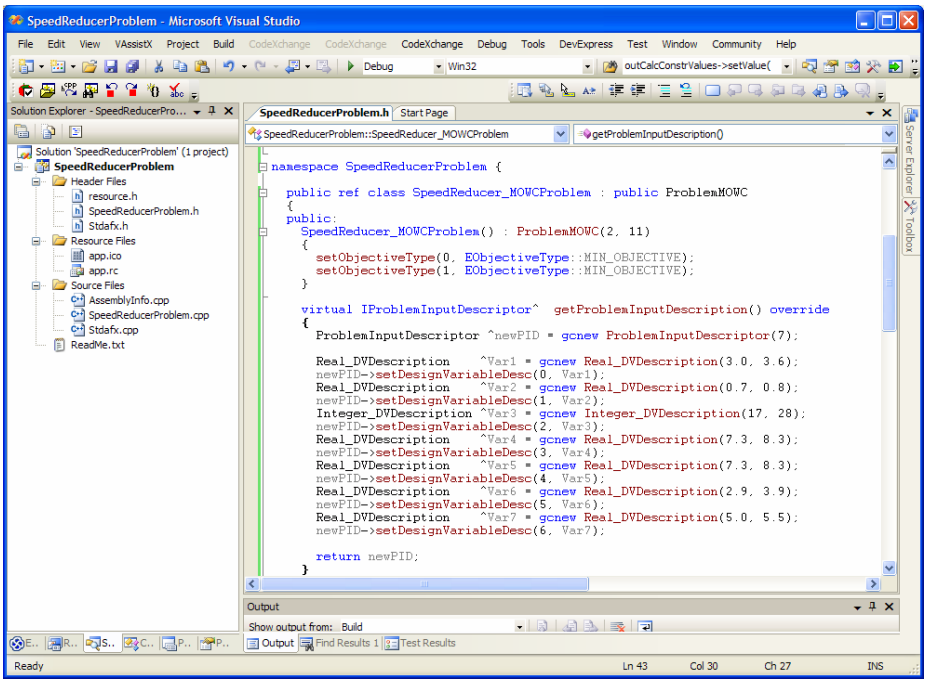

**Slika 5.14 Prikaz kreirane biblioteke razreda u Visual Studio razvojnom okruženju**

Nakon obavljanja ove operacije, potrebno je ugraditi programski kôd kojim će se implementirati razred optimizacijskog problema. Prvi korak je definiranje korištenih ESOP biblioteka razreda i postavljanje #using deklaracija za uvoženje potrebnih prostora imena:



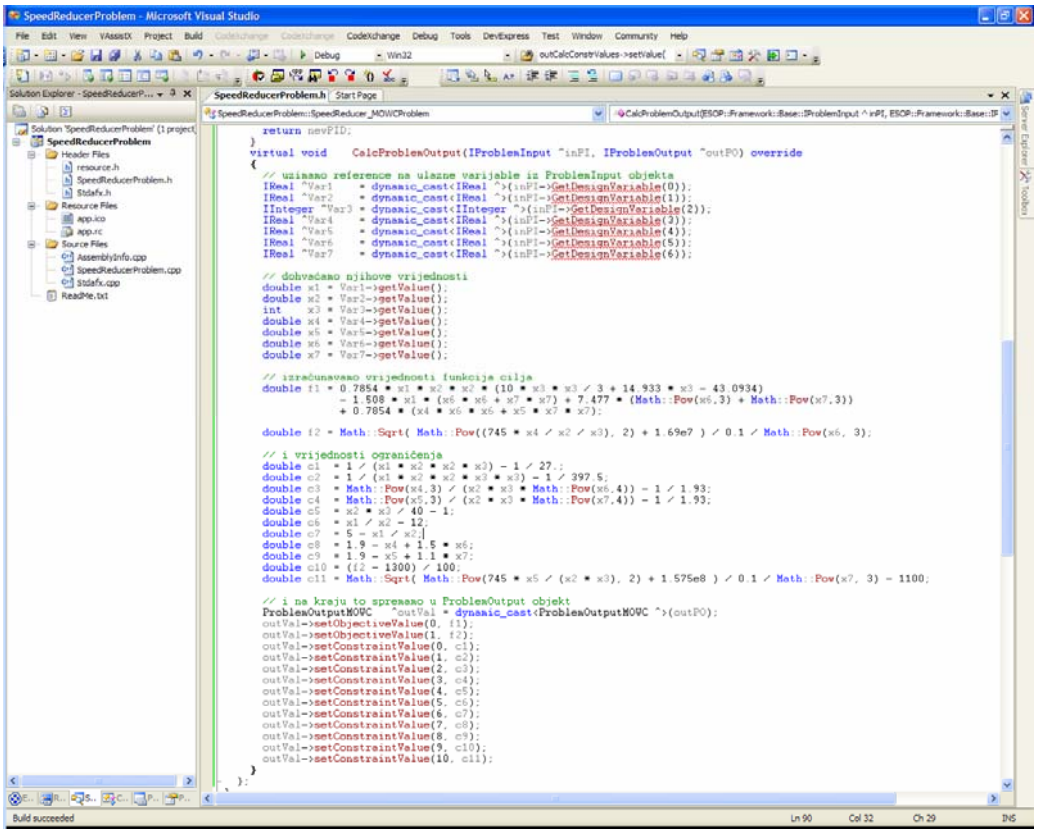

**Slika 5.15 Ugradnja *#using* deklaracija**

Nakon toga je potrebno implementirati zahtjevanu funkcionalnost unutar SpeedReducerProblem razreda. Najprije implementiramo konstruktor (u kojem se definirani broj funkcija cilja i broj ograničenja prenosi konstruktoru baznog razreda ProblemMOWC) i funkciju getProblemInputDescription():



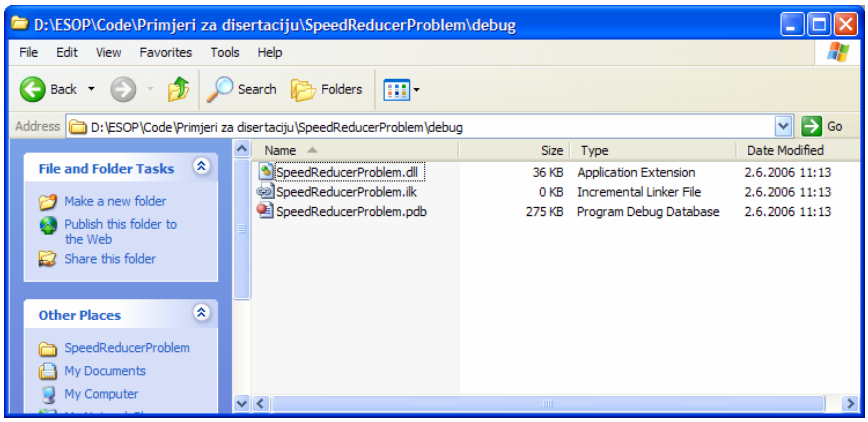

**Slika 5.16 Ugradnja funkcije getProblemInputDescription()**

A zatim implementiramo i funkciju CalcProblemOutput() koja će izračunati izlaze na osnovu predanih vrijednosti ulaznih varijabli (sadržanih u ProblemInput objektu):




**Slika 5.17 Ugradnja funkcije CalcProblemOutput()**

Prevođenjem ugrađenog programskog kôda se kreira biblioteka razreda (*dll*) kako je prikazano na slici 5.18.




**Slika 5.18 Prikaz kreirane *dll* biblioteke razreda**

Izgrađena biblioteka SpeedReducerProblem.dll sadrži potpuni opis danog optimizacijskog problema i predstavlja optimizacijsku komponentu koja se može iskoristiti unutar ESOP okruženja. Rješavanje problema sada podrazumijeva izgradnju nove aplikacije (bilo konzolne, bilo one s GUI sučeljem) na potpuno

istovjetan način kao što je opisano u odjeljku 5.4.2. Potrebno je jedino dodati #using deklaraciju (#using „D:\ESOP\Code\Primjeri za disertaciju\SpeedReducerProblem.dll") kojom će se referencirati biblioteka razreda te prilagoditi skup konkretnih ulaznih varijabli i skup korištenih operatora.

#### 5.5.1.2 Ugradnja kvadratičnog problema pridruživanja

Ugradnja novih instanci već definiranih vrsti optimizacijskih problema se oslanja na izvođenje novih razreda iz predefiniranih sučelja u ESOP okruženju. Međutim, ESOP okruženje se može proširiti i posve novim vrstama problema i u ovom odjeljku će se prikazati ugradnja kvadratičnog problema pridruživanja.

Kvadratični problem pridruživanja (QAP) je na konceptualnom nivou vrlo sličan problemu trgovačkog putnika i ugradnja optimizacijskih komponenti preko kojih će on biti realiziran je izvedena na sličan način.

I ovdje je kreiran razred QAP_Base koji modelira karakteristike svakog kvadratičnog problema pridruživanja. Iz teorijskog opisa QAP-a danog u odjeljku 1.3.4 proizlazi da svaki QAP problem mora definirati broj jedinki pridruživanja (tvornica, strojeva, ...) i da mora definirati sadržaj matrica A i B pomoću kojih se izračunava trošak svakog konkretnog pridruživanja. Deklaracija apstraktnog razreda QAP_Base je stoga sljedeća:

```
public ref class QAP_Base abstract: public ProblemSO
{
public:
   QAP_Base() : ProblemSO(EObjectiveType::MIN_OBJECTIVE) { }

   virtual   void    CalcProblemOutput(IProblemInput ^inPI, IProblemOutput ^outPO) override
   {
      IPermutation  ^x = dynamic_cast<IPermutation ^>(inPI->GetDesignVariable(0));

      int            N = getFacilityNum();
      double       TourLength=0.0;
      for( int  i=0; i<N; i++ ) {
          for( int j=0; j<N; j++ ) {
              int    pi = x->getNth(i);
              int    pj = x->getNth(j);

              TourLength += getA(i,j) * getB(pi, pj);
          }
      }
      ProblemOutputSO     ^outVal = dynamic_cast<ProblemOutputSO ^>(outPO);
      outVal->setObjectiveValue(TourLength);
   }
   virtual IProblemInputDescriptor^           getProblemInputDescription() override
   {
       ProblemInputDescriptor       ^newPID = gcnew ProblemInputDescriptor(1);

       int FacNum = getFacilityNum();
       Permutation_DVDescription ^Perm = gcnew Permutation_DVDescription(FacNum);
       newPID->setDesignVariableDesc(0, Perm);

       return newPID;
    }
```

```
    // čiste virtualne funkcije koje definiraju karakteristike QAP-a
    virtual         int     getFacilityNum() = 0;
    virtual         double  getA(int i, int j) = 0;
    virtual         double  getB(int i, int j) = 0;
};
```

*Programski odsječak 40: Ugradnja baznog asptraktnog razreda za QAP*

S obzirom da za definiranje konkretnih instanci QAP problema vrijedi sve što je već navedeno i za TSP problem (odnosno, mogu se direktno u programskom kodu izgraditi razredi koji definiraju sadržaj matrica A i B a moguće je te podatke učitati iz datoteke), izgrađena su dva razreda, QAP_Complete i QAP_FromFile.

```
public ref class QAP_Complete : public QAP_Base
{
public:
        QAP_Complete();
        QAP_Complete(int inNumFacilities);

        void    setFacilityNum(int inFacNum) {
                _NumFacilities = inFacNum;
               _arrA = gcnew array<double, 2>(_NumFacilities, _NumFacilities);
               _arrB = gcnew array<double, 2>(_NumFacilities, _NumFacilities);
        }
        virtual int     getFacilityNum() override { return _NumFacilities; }

        virtual double  getA(int i, int j) override { return _arrA[i,j];  }
        virtual double  getB(int i, int j) override { return _arrB[i,j];  }
protected:
        int     _NumFacilities;
        array<double, 2>        ^_arrA;
        array<double, 2>        ^_arrB;
};
public ref class QAP_FromFile : public QAP_Complete
{
public:
        virtual void        InitFromFile(String ^FileName) = 0;
};
```

*Programski odsječak 41: Deklaracija razreda QAP_Complete*

U razredu QAP_Complete su definirane članske varijable koje sadrže podatke o broju tvornica, i matrice _arrA i _arrB koje definiraju strukturu troškova pridruživanja. Kod definiranja instance QAP-a programski, potrebno je te vrijednosti popuniti u konstruktoru razreda, odnosno razredi izvedeni iz razreda QAP_Complete moraju definirati vrijednosti tih članskih varijabli. Ukoliko se želi inicijalizirati te podatke na osnovu sadržaja neke datoteke, potrebno je u razredu izvedenom iz razreda QAP_FromFile implementirati člansku funkciju InitFromFile() koja će na osnovu poznatog formata datoteke s učitanim podacima inicijalizirati matrice A i B. Najčešće korištena biblioteka problema kvadratičnog pridruživanja je QAPLIB [Burkard1997] u kojoj je definiran široki skup kako teorijski postavljenih problema tako i konkretnih problema koji su se pojavljivali u praksi. Stoga je izgrađen razred QAP_FromQAPLIB koji implementira funkcionalnost za učitavanje podataka iz datoteka u QAPLIB biblioteci problema.

Ovdje je bitno naglasiti da opisana implementacija nije izgrađena kao standardni dio ESOP okruženja, odnosno nije izgrađena kao dio razvojnog projekta ESOP. Izgrađene komponente su dio zasebne biblioteke razreda koja s ESOP-om interagira samo preko iskorištavanja (referenciranja) dll-ova koji su dio ESOP okruženja (konkretno, BaseInterfacesLib i BaseObjectsLib).

Opisana implementacija samo definira QAP optimizacijski problem. Za njegovo rješavanje je potrebno ugraditi optimizacijske komponente specifične za taj problem (prikladne operatore lokalnog pretraživanja, križanja i mutacije). Detaljan opis ugradnje različitih postupaka optimizacije se može naći u [Vanjak2001].

### 5.5.2 Mogućnost proširenja s ostalim vrstama optimizacijskih komponenti

U prethodna dva odjeljka je opisano proširivanje ESOP okruženja s novim instancama i vrstama optimizacijskih problema. Međutim, time nije opisan cjelokupan skup mogućnosti proširenja ESOP okruženja. Izrađeni objektni model podržava mogućnost ugradnje i ostalih vrsta optimizacijskih komponenti. Izvođenjem konkretnih razreda iz definiranih sučelja se mogu ugraditi nove vrste optimizacijskih postupaka, operatora i reprezentacija. Osnovna sučelja koje nove optimizacijske komponente moraju zadovoljavati su IterativeAlgorithm za optimizacijske postupke, IOperator za ugrađene operatore i IDesignVariable za reprezentacije varijabli. Proširivanje ESOP-a s takvim komponentama se zasniva na korištenju istih programskih tehnika kao i u slučaju proširivanja s novim optimizacijskim problemima i u detalje tih implementacije se neće dalje ulaziti.

Pored proširivanja novim optimizacijskim komponentama, u ESOP okruženje se mogu ugraditi i nove vizualizacijske komponente, odnosno korisničke kontrole za vizualizaciju rezultata i međurezultata optimizacije.

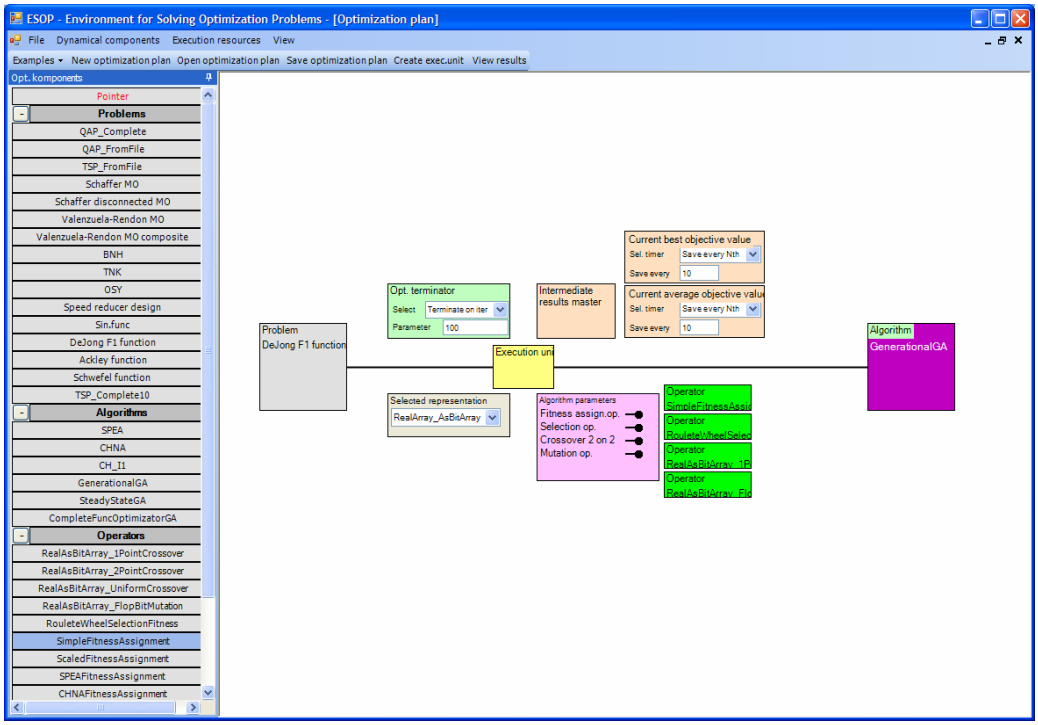
## 5.6 Izgradnja integriranog optimizacijskog okruženja

U prethodnim potpoglavljima je opisana izgradnja osnovne infrastrukture ESOP okruženja, opisana je realizacija ugrađenih programskih komponenti i dan opis mogućnosti proširenja ESOP te je prikazana primjena na rješavanje skupa optimizacijskih problema pri čemu se ugrađena funkcionalnost koristi na nivou izvornog kôda. Zahvaljujući dobro definiranom objektnom modelu, izgradnja programskih rješenja za rješavanje ugrađenih optimizacijskih problema nije komplicirana. Međutim, kako je još u uvodu ove disertacije navedeno, izgradnja takvih programskih rješenja ipak zahtjeva određeni stupanj poznavanja tehnika programiranja i poznavanje korištenja razvojnog okruženja u kojem se gradi dano programsko rješenje.

Kako je jedan od osnovnih ciljeva ove disertacije izgradnja optimizacijskog okruženja koje neće zahtijevati programiranje od strane istraživača u slučaju da već postoje izgrađene optimizacijske komponente, u ovom poglavlju će se dati opis ugradnje te funkcionalnosti u ESOP optimizacijsko okruženje. Ta izgradnja će biti uvelike bazirana na iskorištavanju mogućnosti .NET razvojnog okruženja vezanih uz dinamičko (*run-time*) manipuliranje razredima i objektima, preko skupa razreda iz .NET `Reflection` prostora imena, a čiji detaljan opis će se dati kasnije u ovom poglavlju.

Ukoliko se pažljivo promotre izgrađena programskih rješenja na kojima je prikazana primjena ESOP-a za rješavanje različitih optimizacijskih problema, vidljivo je da programski kôd za provođenje optimizacije ima određenu strukturu. Kôd se u cijelosti sastoji od kreiranja i povezivanja odgovarajućih optimizacijskih komponenti (objekata) koji predstavljaju optimizacijski problem, odabranu reprezentaciju ulaznih varijabli te optimizacijski postupak i njegove potrebne operatore, te postavljanja odgovarajućih parametara. Postojanje širokog skupa ugrađenih programskih komponenti svodi posao istraživača na definiranje *optimizacijskog konteksta*, odnosno na odabir korištenih optimizacijskih komponenti i njihovih parametara, a ugrađeni objekti za definiranje trajanja optimizacije i rukovanje rezultatima i međurezultatima dodatno pojednostavljuju posao kod izgradnje programskog rješenja za provođenje optimizacije.

Upravo postojanje predefiniranih sučelja koja optimizacijske komponente moraju zadovoljavati omogućava izgradnju programskog rješenja koje će omogućavati vizualno definiranje optimizacijskog konteksta preko GUI sučelja. Kako bi izlaganje koje slijedi bilo što jasnije i kako bi se stekla bolja slika o tome što se zapravo misli pod pojmom *vizualno slaganje optimizacijskog konteksta*, na slici 5.19 je prikazan izgled optimizacijskog konteksta dinamički definiranog u ESOP okruženju, za rješavanje DeJong_F1 testnog problema pomoću generacijskog genetičkog algoritma.




**Slika 5.19 Prikaz ESOP okruženja u akciji**

Vizualizirani optimizacijski kontekst je u funkcionalnom pogledu posve identičan programskom rješenju koje je izgrađeno kao konzolna aplikacija (opisanom u odjeljku 5.4.1) za rješavanje DeJong_F1 testnog problema. To znači da će se prilikom provođenja optimizacije iskoristiti potpuno isti skup optimizacijskih komponenti. Prednost ovakvog načina definiranja optimizacijskog konteksta u odnosu na izgradnju posvećenog programskog rješenja u nekom .NET programskom rješenju je očigledna. U nastavku ovog poglavlja će biti opisana programska realizacija integriranog optimizacijskog okruženja ESOP, koje će se daljnjem tekstu nazivati ESOP *ljuskom*. S obzirom na određene prednosti koje programski jezik C# kod izgradnje aplikacija s vizualnim sučeljem pokazuje u odnosu na programski jezik C++/CLI, ESOP ljuska je izgrađena korištenjem programskog jezika C#, u Visual Studio 2005 razvojnom okruženju.

## 5.6.1 Analiza utjecaja zahtjeva na arhitekturu

Da bi se izgradila aplikacija koja će omogućavati vizualno definiranje optimizacijskog konteksta, programsko rješenje mora zadovoljavati nekoliko zahtjeva.

Kao prvi zahtjev se postavlja mogućnost dinamičkog pronalaženja optimizacijskih komponenti u izgrađenim bibliotekama i njihovo učitavanje u ESOP ljusku. Drugi zahtjev je povezan s ugradnjom mogućnosti vizualnog definiranja optimizacijskog konteksta, primjer kojega je prikazan na slici 5.19. Ugradnja te mogućnosti podrazumijeva definiranje vizualne reprezentacije za pojedine vrste optimizacijskih komponenti, definiranje načina njihovog povezivanja i postavljanja određenih parametara. I na kraju, mora se ugraditi mogućnost upravljanja s tako definiranim vizualnim optimizacijskim kontekstom što znači mogućnost pokretanja

optimizacije, upravljanja njenim tijekom i u konačnici mogućnost vizualizacije dobivenih rezultata.

#### 5.6.1.1 Dinamička identifikacija i kreiranje komponenti

Kod korištenja standardnih Windows dll-ova, programske komponente koje ih grade nemaju ugrađene metapodatke koji bi opisivali skup ugrađenih tipova i njihove funkcionalnosti. Stoga je kod korištenja takvih biblioteka uvijek potrebno koristiti dodatne podatke koji su najčešće sadržani u bibliotekama tipova (engl. *type library*). Takvo razdvajanje, uz činjenicu da standardni Windows dll-ovi ne sadrže ugrađene razrede već skup samostalnih funkcija, značajno komplicira ugradnju mogućnosti dinamičkog istraživanja sadržaja pojedinog dll-a.

Srećom, .NET okruženje je iz osnova projektirano s podrškom za mogućnost dinamičkog istraživanja sadržaja dll-ova preko tzv. *reflection* mehanizma. Detaljan opis realizacije te funkcionalnosti u .NET okruženju je izvan dosega ove disertacije i može se naći u [Richter2002]. Pojednostavljeno rečeno, svaka .NET biblioteka razreda u sebi ima ugrađene metapodatke koji, korištenjem zajedničkog sistema tipova (*Common Type System*), detaljno opisuju karakteristike razreda ugrađenih u tu biblioteku. To znači da je programski moguće saznati potpunu deklaraciju ugrađenih razreda: njihove nazive, ugrađene članske varijable i njihove tipove, ugrađene članske funkcije i skup njihovih parametara, i, što je ključno, skup sučelja koje dotični razred podržava. Za ilustraciju tog mehanizma, slijedi kratki programski odsječak u kojem je prikazano kako se programski mogu identificirati razredi, ugrađeni u biblioteci razreda, koji zadovoljavaju određeno sučelje:

```
string filePath = „D:\ESOP\Code\debug\StandardOptimizationObjectsLib.dll";
Assembly asmLoadedAssembly = Assembly.LoadFrom(filePath);

Type[] mytypes = asmLoadedAssembly.GetExportedTypes();
foreach (Type t in mytypes)
{
    TypeFilter myFilter = new TypeFilter(MyInterfaceFilter);

    Type[] myInterfaces = t.FindInterfaces(myFilter, "IIterativeAlgorithm");
    if (myInterfaces.Length > 0)
    {
        // ... obavljanje potrebnih operacija s tipom
        IIterativeAlgorithm algor = (IIterativeAlgorithm)Activator.CreateInstance(t);
    }
}
```

*Programski odsječak 42: Dinamička identifikacija tipova u dll-u*

Najprije se pomoću Assembly.LoadFrom() funkcije dinamički učitava biblioteka razreda nakon čega se pomoću članske funkcije GetExportedTypes() saznaje skup tipova (razreda) koji je u njoj ugrađen. Korištenjem funkcije FindInterfaces() se zatim provjerava da li dani tip zadovoljava zadano sučelje.

Pored mogućnosti istraživanja sadržaja biblioteke razreda i doznavanja sučelja koje pojedini tipovi / razredi zadovoljavaju, važna je i mogućnost instanciranja objekata. Ukoliko je poznat tip, odnosno ukoliko postoji varijabla tipa Type koja sadrži podatke o tipu, novi objekt tog tipa je vrlo jednostavno instancirati korištenjem Activator.CreateInstance() funkcije. Opisana funkcionalnost ugrađena u

.NET okruženje je ključna za jednostavno dinamičko iskorištavanje programskih komponenti ugrađenih u dinamičke biblioteke razreda, i kod razvoja ESOP okruženja je ta funkcionalnost iskorištena do maksimuma.

#### 5.6.1.2 Vizualno definiranje optimizacijskog konteksta

Pod vizualnim definiranjem optimizacijskog konteksta podrazumijevamo mogućnost odabira optimizacijskih komponenti, njihovog povezivanje i definiranja odgovarajućih parametara preko vizualnog (GUI) sučelja, bez potrebe za kodiranjem. U prethodnom odjeljku je opisan način na koji se optimizacijske komponente mogu dinamički učitati iz biblioteka razreda. Da bi se omogućilo manipuliranje tim komponentama preko vizualnog sučelja, potrebno je kreirati njihove vizualne reprezentacije, odnosno, potrebno je kreirati skup prezentacijskih kontrola koje će na površini forme reprezentirati pojedine optimizacijske komponente i preko kojih će korisnik-optimizator imati mogućnost podešavanja njihovih parametara i povezivanja unutar optimizacijskog konteksta.

#### 5.6.1.3 Upravljanje s učitanim komponentama

Pored dinamičke identifikacije i učitavanja razreda/objekata iz biblioteka razreda i njihovog vizualnog reprezentiranja na ekranskoj formi, treći bitan aspekt kod ugradnje mogućnosti vizualnog definiranja optimizacijskog konteksta je vezan uz programsko upravljanje s tako kreiranim objektima. Programsko upravljanje objektima znači pozivanje članskih funkcija definiranih u njihovom javnom sučelju. Na primjer, ukoliko dinamički kreiramo neki objekt koji zadovoljava sučelje IIterativeAlgorithm na način kako je to prikazano u prethodnom programskom odsječku:

```
IIterativeAlgorithm algor = (IIterativeAlgorithm)Activator.CreateInstance(t);
```

s kreiranim objektom algor možemo komunicirati samo preko sučelja IIterativeAlgorithm, iako se u stvarnosti radi o objektu nekog konkretnog tipa poput SO_GenerationalGA ili RandomSearch_SO. Stoga nad tako kreiranim objektom možemo pozivati funkcije iz sučelja IIterativeAlgorithm razreda, npr.:

```
algor.SetProblem(...);
algor.Run();
```

Bitno je naglasiti da članske funkcije definirane u razredu SO_GenerationalGA nisu dostupne jer tijekom izvršavanja nije dostupna informacija da se radi o objektu tog tipa. Stoga ovdje vidimo ključnu ulogu sučelja preko kojega je konkretna optimizacijska komponenta, odnosno razred, dinamički kreirana.

### 5.6.2 Opis programskog rješenja

Dio ESOP optimizacijskog okruženja opisan u poglavlju 5.3 je realiziran preko nekoliko biblioteka razreda koje se mogu iskoristiti prilikom gradnje posvećenih programskih rješenja za optimizaciju. Odnosno, za iskorištavanje ugrađenih optimizacijskih komponenti je potrebno izgraditi novu aplikaciju koja će te komponente koristiti statički, na nivou izvornog kôda. ESOP ljuska je, za razliku od tog dijela ESOP okruženja, izgrađena kao samostalna (engl. *stand-alone*) GUI aplikacija. Kod izgradnje ESOP ljuske je primjenjen princip modularne izgradnje na način da je izgrađena funkcionalnost programski realizirana kroz više programskih

komponenti od kojih svaka ugrađuje određeni aspekt funkcionalnosti potrebne za dinamičko definiranje optimizacijskog konteksta / programa i njegovo izvođenje, odnosno provođenje optimizacije.

#### 5.6.2.1 Arhitektura

S obzirom da se opisani skup funkcionalnosti može grupirati u nekoliko dobro definiranih cjelina, ESOP ljuska je izgrađena kroz sljedeći skup programskih modula:

- ESOPApp - aplikacija s GUI sučeljem kroz koju je dostupna cjelokupna funkcionalnost ugrađena u ESOP ljusku.
- DynamicSupportInterfacesLib - biblioteka razreda u kojoj su definirani razredi sučelja koje moraju zadovoljavati optimizacijske komponente ukoliko se želi pružiti mogućnost njihovog iskorištavanje unutar ESOP ljuske.
- OptimizationContextManagementLib - biblioteka razreda u koju je ugrađena funkcionalnost za definiranje optimizacijskog konteksta.
- PresentationLayerLib - biblioteka prezentacijskih komponenti kojima se vizualiziraju pojedine optimizacijske komponente i koje se koriste kod vizualnog definiranja optimizacijskog konteksta.

#### 5.6.2.2 DynamicSupportInterfacesLib

Prvi zadatak kod izgradnje ESOP ljuske je ugraditi funkcionalnost za dinamički rad s objektima/razredima preko kojih su u dinamičkim bibliotekama razreda ugrađene optimizacijske komponente. Potrebno je realizirati dvojaku funkcionalnost. Kao prvo, potrebno je omogućiti pronalaženje i identifikaciju optimizacijskih komponenti u biblioteci razreda. Kako je već opisano, ta identifikacija se obavlja enumeracijom svih tipova/razreda ugrađenih u *dll* i ispitivanjem da li pojedinačni tip zadovoljava neko predefinirano sučelje, preko kojega je određeno i o kakvom se tipu optimizacijske komponente radi. Drugi zadatak je za pronađene optimizacijske komponente iz *dll*-a učitati podatke na osnovu kojih će se kasnije dinamički moći instancirati objekti odgovarajućeg tipa, što je nužno obaviti kod provođenja (izvršavanja) optimizacije.

Način identifikacije razreda ugrađenih u *dll* koji zadovoljavaju neko sučelje je ilustriran programskim odsječkom 42. S obzirom da se optimizacijske komponente u ESOP okruženje ugrađuju upravo preko razreda koji moraju zadovoljiti određena sučelja ugradnja tražene funkcionalnosti je jednostavna. Na način kao što se u programskom odsječku 42 identificiraju ugrađeni optimizacijski postupci preko sučelja IIterativeAlgorithm, moguće je identificirati i ugrađene reprezentacije (preko sučelja IDesignVariable), ugrađene operatore (IOperator) i optimizacijske probleme (sučelje IProblem).

Međutim, kod postavljanja arhitekture cjelokupnog ESOP okruženja ta mogućnost nije iskorištena već je definiran poseban skup sučelja koja optimizacijske komponente moraju zadovoljavati ukoliko se žele iskoristiti unutar ESOP ljuske. Razlog tome je dvojak. Prvi razlog je želja za maksimalnim razdvajanjem funkcionalnosti ESOP okruženja vezanog uz statičko i dinamičko iskorištavanje ugrađenih programskih komponenti. Iako je konačni cilj izgraditi optimizacijsko okruženje (ljusku) koje će pružati kompletnu kontrolu nad definiranjem

optimizacijskog konteksta, zbog različitosti optimizacijskih komponenti i vrlo širokog skupa mogućnosti njihovog međusobnog povezivanja, to je tehnički vrlo teško izvedivo. Stoga se u većini slučajeva kod rješavanja optimizacijskih problema u praksi ipak mora pribjeći programiranju, odnosno izgradnji nove aplikacije u kojoj će se postojeće optimizacijske komponente iskorištavati na nivou izvornog koda. Jednostavno, mogućnost kontrole koju pruža rad s izgrađenim razredima na nivou izvornog kôda su značajno veće u odnosu na definiranje optimizacijskog konteksta putem vizualnog sučelja.

Drugi važan razlog leži u činjenici da korištenje optimizacijskih komponenti na dinamički način, znači unutar ESOP ljuske, povlači dodatne zahtjeve na njihovu ugradnju. Odnosno, ukoliko se optimizacijska komponenta realizirana preko razreda želi iskoristiti unutar ESOP ljuske, morati će se u nju ugraditi dodatna funkcionalnost.

Zbog ovih dvaju razloga je cilj kod izgradnje ESOP okruženja bio razdvojiti te dvije mogućnosti iskorištavanja optimizacijskih komponenti. To razdvajanje je provedeno na način da se u ESOP mogu ugrađivati komponente koje će biti namijenjen samo statičkom iskorištavanju. Kod izgradnje tih komponenti se može posve zanemariti postojanje ESOP ljuske i one se koriste na način kao što je opisano u potpoglavlju 5.4. U tom slučaju korisnik-istraživač ima potpunu kontrolu nad optimizacijskim kontekstom, a negativna strana je što se mora izgraditi novo programsko rješenje, odnosno aplikacija.

Ukoliko se želi izgrađenu optimizacijsku komponentu koristiti unutar ESOP ljuske, tada je njena implementacija složenija jer se mora ugraditi i funkcionalnost na koju će se osloniti ESOP ljuska prilikom rada s tom optimizacijskom komponentom. Ta dodatna funkcionalnost je modelirana preko skupa sučelja koja optimizacijske komponente moraju zadovoljiti i preko kojih se ugrađuje sva potrebna funkcionalnost za njihovo iskorištavanje unutar ESOP ljuske.

#### 5.6.2.2.1 Sučelja za optimizacijske probleme

Sučelja koja moraju zadovoljiti razredi koji predstavljaju optimizacijske probleme su vrlo jednostavna i proizlaze iz različitih vrsta problema. Deklaracije sučelja su sljedeća:

```
public interface class IESOPProblem
{
public:
        virtual IProblemInputDescriptor^   ESOP_GetProblemInputDescription() = 0;
};
public interface class IESOPCompleteProblem : public IESOPProblem
{
};
public interface class IESOPParametrizedProblem : public IESOPProblem
{
public:
        virtual int         getParamNum() = 0;
        virtual String^     getParamName(int ParamInd) = 0;

        virtual void        setParamValue(int Ind, double inVal) = 0;
        virtual double      getParamValue(int Ind) = 0;
};
```

```
public interface class IESOPFileInitProblem : public IESOPProblem
{
public:
	virtual void	InitFromFile(String ^FileName) = 0;
};
```

*Programski odsječak 43: Deklaracije sučelja za dinamički rad s optimizacijskim problemima*

Osnovno je sučelje IESOPProblem koje definira jednu čistu virtualnu funkciju koju moraju ugraditi izvedeni razredi. To je funkcija ESOP_GetProblemInputDescriptor koja vraća opis karakteristika ulaznih varijabli problema. Važnost postojanja te funkcionalnosti će biti jasnija malo kasnije, kod opisa sučelja koje moraju zadovoljavati varijable.

Postojanje tri vrste razreda proizlazi iz tri vrste optimizacijskih problema i različitosti upravljanja s pripadnim objektima koji ih realiziraju. Prvi je IESOPCompleteProblem koji predstavlja kompletni problem koji je za optimizaciju spreman čim se instancira odgovarajući objekt. Druga vrsta su problemi s parametrima. S obzirom da različiti problemi kod realizacije preko razreda definiraju članske funkcije različitog imena za postavljanje svojih parametara (npr. setNumCities() kod TSP-a, odnosno setNumFacilities() kod QAP-a), unutar IESOPParametrizedProblem sučelja je ugrađen skup funkcija kojima se parametrima problema pristupa na generički način. Odnosno, ESOP ljuska će za postavljanje parametara odgovarajućeg objekta koristiti funkciju setParamValue() a razred koji predstavlja konkretan optimizacijski problem će poziv te funkcije proslijediti svojoj odgovarajućoj članskoj funkciji.

Treća vrsta optimizacijskih problema su oni čiji se parametri definiraju na osnovu sadržaja neke datoteke. Ti razredi moraju realizirati funkciju InitFromFile() u koju se ugrađuje programski kod kojim će se inicijalizirati kreirani objekt problema na osnovu sadržaja datoteke čije ime se predaje kao parametar funkciji.

#### 5.6.2.2.2 Sučelja za optimizacijske algoritme

Definiranje svojstava objekta koji predstavljaju optimizacijske postupke predstavlja najznačajniji dio definiranja optimizacijskog konteksta. S obzirom da se i kod izvršavanja dinamički (vizualno) definiranog optimizacijskog konteksta moraju nad tim objektima provesti iste operacije kao i u slučaju statičkog iskorištavanje, za ilustraciju će se najprije prikazati programski odsječak u kojem se vidi kako se s objektima algoritma upravlja na statički način (prenesno iz programskog odsječka 38):

```
// povezivanje s optimizacijskim problemom
Alg->SetProblem(ProbObj);

// povezivanje s kreiranim objektom ProblemInput koji sadrži odabranu reprezentaciju
Alg->SetProblemInput(PI);

// postavljanje parametara algoritma
Alg->setPopulationSize(inPopSize);
Alg->setMutationProbability(inMutProb);
Alg->setCrossoverProbability(inCrossProb);

// definiranje operatora
IFitnessAssignment ^fitAssign = gcnew FitnesAssignment_EqualToObjValue();
```

```
Alg->setFitnessAssignmentOp(fitAssign);

IFitnessSelectionOperator  ^select = gcnew RouleteWheelSelectionFitness(inRnd);
Alg->setFitnessSelectionOp(select);

ICrossoverOperator2To2 ^cross = gcnew RealArray_ArithmeticRecombinationCrossover();
IMutationOperator ^mut              = gcnew RealArray_RandomMutation(inRnd);

Alg->setCrossover(0, cross);
Alg->setMutation(0, mut);

// pokretanje optimizacije
Alg->Run();
```

Na osnovu ovog skupa operacija, definirani su sljedeći razredi sučelja koje moraju zadovoljavati izgrađeni razredi optimizacijskih algoritama kako bi se mogli dinamički iskoristiti unutar ESOP ljuske.

```
public interface class IESOPAlgorithm
{
public:
        virtual void        ESOP_SetProblemInput(IProblemInput ^SolInst) = 0;
        virtual void        ESOP_SetProblemInstance(IESOPProblem ^SolInst) = 0;

        virtual    void     ESOP_Run() = 0;
};

public interface class IESOPCompleteAlgorithm  : public IESOPAlgorithm
{
public:
        virtual int         getParamNum() = 0;
        virtual String^     getParamName(int ParamInd) = 0;

        virtual void        setParamValue(int Ind, double inVal) = 0;
        virtual double      getParamValue(int Ind) = 0;
};

public interface class IESOPTemplateAlgorithm  : public IESOPCompleteAlgorithm
{
public:
        virtual int         getAlgorithmOperatorNum() = 0;
        virtual Type^       getAlgorithmOperatorType(int OperInd) = 0;

        virtual void        setAlgorithmESOPOperator(int OperIndex, IESOPOperator ^oper) = 0;

        virtual int         getDesignVarOperatorNum() = 0;
        virtual Type^       getDesignVarOperatorType(int OperInd) = 0;

        virtual void        setDesignVarESOPOperator(int OpInd, int DVInd,IESOPOperator ^op) = 0;
};
```

*Programski odjsečak 44: Razredi sučelja za algoritme namijenjeni iskorištavanju unutar ESOP ljuske*

Osnovno je sučelje IESOPAlgorithm koje u sebi modelira funkcionalnost koju mora ugraditi svaki razred koji predstavlja konkretan optimizacijski postupak. Funkcije ESOP_SetProblem(), ESOP_SetProblemInstance() i ESOP_Run() se direktno realiziraju

proslijeđivanjem (engl. *forwarding*) odgovarajućim funkcijama iz IIterativeAlgorithm sučelja. Primjer kako to izgleda za SO_GenerationalGA razred je sljedeći:

```
public ref class SO_GenerationalGA : public GeneralGenerationalGA,
                                     public IESOPTemplateAlgorithm,

    virtual void ESOP_SetProblemInput(IProblemInput ^SolutionInst)
                                        { SetProblemInput(SolutionInst); }
    virtual void ESOP_SetProblemInstance(IESOPProblem ^ProblemInst)
                                        { SetProblem(dynamic_cast<IProblem ^>(ProblemInst));}

    virtual void ESOP_Run()    { Run(); }
    // .. ostatak razreda
```

Specifičnosti pojedinih vrsta optimizacijskih postupaka su modelirane definiranjem dva specijalizirana sučelja. IESOPCompleteAlgorithm će naslijediti razredi algoritma kojima je za definiranje njihovog izvršnog konteksta potrebno samo postaviti određeni skup parametara. Primjer takvog algoritma je RandomSearch algoritam.

IESOPTemplateAlgorithm sučelje je namijenjeno razredima koji u svom radu koriste operatore. Operatori algoritma dolaze u dvije varijante. Prvi su operatori koji operiraju u kontekstu cijelog optimizacijskog postupka, čega su tipičani primjeri IFitnessAssignment i IFitnessBasedSelection operatori kod genetičkih algoritama. Druga varijanta su operatori koji djeluju nad reprezentacijom, odnosno skupom varijabli problema. Tipičan primjer takvih operatora su ICrossover i IMutation operatori. S obzirom da se ova vrsta operatora mora pridružiti određenoj varijabli problema, sučelje za rad s njima je nešto drugačije.

I za jednu i za drugu vrstu operatora je definiran sličan skup funkcija. Preko ugradnje funkcije get...OperatorNum() će se definirati broj operatora koji je algoritmu potrebno postaviti, preko funkcije get...OperatorType() će se vratiti konkretan tip operatora, a preko funkcije set...ESOPOperator() će se objektu algoritma pridružiti konkretna instanca objekta operatora.

Kao primjer načina ugradnje ovih funkcija slijedi realizacija ugrađena u razred SO_GenerationalGA razred:

```
    virtual int         getParamNum()                   { return 1; }
    virtual String^     getParamName(int ParamInd)      { return "Population size"; }

    virtual void        setParamValue(int Ind, double inVal) { setPopulationSize((int) inVal); }
    virtual double      getParamValue(int Ind)          { return getPopulationSize(); }

    virtual int         getAlgorithmOperatorNum()       { return 2; }
    virtual Type^       getAlgorithmOperatorType(int OperInd)  {
        if( OperInd == 0 )      return IFitnessAssignmentOperator::typeid;
        else                    return IFitnessBasedSelectionOperator::typeid;
    }
    virtual void setAlgorithmESOPOperator(int OperIndex, IESOPOperator ^oper)    {
        if( OperIndex == 0 )
            setFitnessAssignmentOp(dynamic_cast<IFitnessAssignmentOperator ^>(oper));
        else
            setFitnessSelectionOp(dynamic_cast<IFitnessBasedSelectionOperator ^>(oper));
    }

    virtual int         getDesignVarOperatorNum()               { return 2; }
```

```
        virtual Type^      getDesignVarOperatorType(int OperInd)     {
            if( OperInd == 0 )     return ICrossoverOperator2To2::typeid;
            else                   return IMutationOp::typeid;
        }
        virtual void setDesignVarESOPOperator( int OperIndex, int DesVarIndex,
                                               IESOPOperator ^oper)
        {
            if( OperIndex == 0 )
                setCrossover(DesVarIndex, dynamic_cast<ICrossoverOperator2To2 ^>(oper));
            else
                setMutation(DesVarIndex, dynamic_cast<IMutationOp ^>(oper));
        }
```

*Programski odsječak 45: Realizacija funkcija iz IESOP... sučelja u SO_GenerationalGA razredu*

Na ovaj način se funkcionalnost potrebna za dinamičko iskorištavanje komponente algoritma nadovezuje na već ugrađenu statičku funkcionalnost.

#### 5.6.2.2.3 Sučelja za varijable problema

Sučelje koje moraju zadovoljavati izgrađeni razredi varijabli problema kako bi se mogli iskoristiti unutar dinamičkog okruženja ESOP ljuske je vrlo jednostavno:

```
public interface class IESOPDesignVariable
{
public:
        virtual void        ESOP_SetVariableDescriptor(IDesignVariableDescriptor ^inDesc) = 0;

        virtual int         getParamNum() = 0;
        virtual String^     getParamName(int ParamInd) = 0;

        virtual void        setParamValue(int Ind, double inVal) = 0;
        virtual double      getParamValue(int Ind) = 0;
};
```

*Programski odjsečak 46: Sučelje za varijable namijenjene iskorištavanju unutar ESOP ljuske*

Funkcije za dohvaćanje i postavljanje vrijednosti parametara se prirodno preslikavaju u pripadne funkcije za pojedine vrste varijabli. Tako će Real razred imati dva parametra, donju i gornju granicu intervala vrijednosti, i realizacija tih funkcija će se sastojati u proslijeđivanju poziva funkcijama koje su već ugrađene u taj razred.

Potreba za funkcijoma ESOP_SetVariableDescriptor() proizlazi iz načina incijalizacije varijable. U slučaju statičkog korištenja, ta inicijalizacija se provodi u samom konstruktoru kojem se predaje ProblemInputDescriptor objekt koji je dobiven od objekta problema:

```
// definiramo problem
DeJong_F1_TestProblem ^ProbObj = gcnew DeJong_F1_TestProblem();
ProblemInput ^PI = gcnew ProblemInput(ProbObj->getProblemInputDescription());

// definiramo koristenu reprezentaciju
IRealArray      ^DV1    = gcnew RealArray(dynamic_cast<RealArray_DVDescription ^>
                                  (PI->getProblemInputDesc()->getDesignVariableDesc(0)));
PI->AssignDesignVariable(0, DV1);
```

S obzirom da se ista operacija mora obaviti i nad dinamički kreiranim objektom varijable problema, u sučelje je ugrađena i funkcija ESOP_SetVariableDescriptor() preko koje će se objekt varijable inicijalizirati u slučaju dinamičkog korištenja.

#### 5.6.2.2.4 Sučelja za operatore

Sučelja koja moraju zadovoljavati razredi kojima se ugrađuju operatori su vrlo jednostavna. Postoje dvije vrste operatora: kompletni i operatori s parametrima i oni su modelirani preko sljedeća dva razreda sučelja:

```
public interface class IESOPOperator
{
};

public interface class IESOPParameterOperator : public IESOPOperator
{
public:
        virtual int         getParamNum() = 0;
        virtual String^     getParamName(int ParamInd) = 0;

        virtual void        setParamValue(int Ind, double inVal) = 0;
        virtual double      getParamValue(int Ind) = 0;
};
```

*Programski odsječak 47 : Sučelja za operatore namijenjene iskorištavanju unutar ESOP ljuske*

Značenje funkcija za dohvaćanje i postavljanje vrijednosti parametara i način njihove realizacije je isti kao i za prethodno opisane vrste optimizacijskih komponenti.

### 5.6.2.3 OptimizationContextManagementLib biblioteka

Zadovoljavanjem sučelja opisanih u prethodnom odjeljku, optimizacijske komponente ugrađuju funkcionalnost koja je potrebna za njihovo dinamičko iskorištavanje unutar ESOP ljuske. Unutar biblioteke OptimizationContext-ManagementLib je ugrađena srž ESOP ljuske. Razredi ugrađeni u ovu biblioteku čine objektni model nad kojim djeluju objekti iz prezentacijskog sloja, ugrađeni djelomično u PresentationLayer biblioteku vizualizacijskih komponenti a djelomično u samu ESOP aplikaciju.

Osnovu ove biblioteke čine tri razreda: OptCompLoader, OptCompManager i OptimizationProgram. U OptCompLoader razredu je ugrađena funkcionalnost za ispitivanje sadržaja biblioteka razreda i učitavanje odgovarajućih podataka o tipovima, OptCompManager razred pruža funkcionalnost za rukovanje kolekcijom učitanih optimizacijskhi komponenti, dok OptimizationProgram razred sadrži sve podatke o definiranom optimizacijskom kontekstu.

Glavna funkcionalnost OptCompLoader razreda je realizirana u funkciji LoadESOPComponents koja, koristeći programsku tehniku opisanu u programskom odsječku 42 istražuje sadržaj dll-a i učitava podatke o nađenim optimizacijskim komponentama. Podaci o učitanim komponetama se spremaju u skup LoadedComponent razreda. Osnovni je razred BaseLoadedComponent sa sljedećom deklaracijom:

```
public abstract class BaseLoadedComponent
{
  public string _DLLName;
  public string _Namespace;
  public string _ComponentName;

  // podaci potrebni za rekreiranje dinamičkog objekta za optimizaciju
  public Type _Type;

  public BaseLoadedComponent(   string inDLLName, string inNamespace,
                                string inComponentName, Type inType)
  {
    _DLLName = inDLLName;
    _Namespace = inNamespace;
    _ComponentName = inComponentName;

    _Type = inType;
  }
  public abstract BaseLoadedComponent Clone();
}
```

*Programski odsječak 48 : Bazni razred za reprezentiranje dinamički učitane optimizacijske komponente*

Podaci o imenu biblioteke, prostoru imena unutar kojeg je komponenta definirana i nazivu same optimizacijske komponente se dohvaćaju prilikom istraživanja sadržaja dll-a i pospremaju u odgovarajuće članske varijable. Najvažniji član razreda je _Type članska varijabla u koju se pospremaju podaci o tipu komponente. Važnost ovog podatka proizlazi iz činjenice da se na osnovu njega pomoću Activator.CreateInstance() mogu dinamički kreirati instance tog razreda (objekti).

Iz ovog razreda su izvedeni specijalizirani LoadedComponent razredi za pojedine vrste optimizacijskih komponenti. S obzirom da su izgradnje vrlo slične, dati će se samo primjer odgovarajućih razreda za optimizacijske algoritme.

```
public abstract class BaseLoadedAlgorithm : BaseLoadedComponent
{
  public int ParamNum  {
    get { return _ParamNum; }
    set { _ParamNum = value; _arrParamName = new string[_ParamNum];}
  }
  private  string[]     _arrParamName;
  private  int          _ParamNum;

  public BaseLoadedAlgorithm(string DLLFilePath, string Namespace, string CompName,
                  Type inType, int inParamNum)  : base(DLLFilePath, Namespace, CompName, inType)
  {
    ParamNum = inParamNum;
  }

  public string GetParamName(int Ind)                {     return _arrParamName[Ind];    }
  public void SetParamName(int Ind, string inName){     _arrParamName[Ind] = inName;    }
}

public class LoadedCompleteAlgorithm : BaseLoadedAlgorithm
{
  public LoadedCompleteAlgorithm(string DLLFilePath, string Namespace, string CompName,
                                 Type inType, int inNumParam)
```

```
    : base(DLLFilePath, Namespace, CompName, inType, inNumParam)   {   }

  public override BaseLoadedComponent Clone() { ... }
 }
 public class LoadedTemplateAlgorithm : BaseLoadedAlgorithm
 {
  public int AlgOperNum          {    get { return _AlgOperNum; }   }
  public int DesVarOperNum       {    get { return _DesVarOperNum; }   }

  // podaci o zahtjevanim tipovima operatora
  private int      _AlgOperNum;
  private int      _DesVarOperNum;

  private string[]      _arrAlgOperDesc;
  private string[]      _arrDesVarOperDesc;

  private Type[]        _arrAlgOperType;
  private Type[]        _arrDesVarOperType;

  public LoadedTemplateAlgorithm(string DLLFilePath, string Namespace, string CompName,
                            Type inType, int inParamNum, int inAlgOperNum, int inDesVarOperNum)
                                : base(DLLFilePath, Namespace, CompName, inType, inParamNum)
  {
    _AlgOperNum      = inAlgOperNum;
    _DesVarOperNum = inDesVarOperNum;

    _arrAlgOperDesc       = new string[inAlgOperNum];
    _arrDesVarOperDesc  = new string[inDesVarOperNum];

    _arrAlgOperType       = new Type[inAlgOperNum];
    _arrDesVarOperType  = new Type[inDesVarOperNum];
  }
  public override BaseLoadedComponent Clone()  ... }
  // ... ostatak razreda (get/set funkcije)
}
```

*Programski odsječak 49: Deklaracija razreda BaseLoadedAlgorithm*

Osnovna uloga LoadedComponent razreda je da sadržavaju sve podatke potrebne za potpuni opis odgovarajuće optimizacijske komponente. Na način analogan gore opisanome su izgrađene i komponente za predstavljanje optimizacijskih problema, varijabli i operatora. Pored toga su definirani i razredi koji će predstavljati objekte za snimanje međurezultata i definiranje trajanja optimizacije.

Kako je već spomenuto, skupom učitanih komponenti upravlja OptComManager razred. Njegova izgradnja je jednostavna i svodi se na upravljanje sadržanim kolekcijama odgovarajućih LoadedComponent objekata:

```
public class OptimizationComponentsManager
{
        List<BaseLoadedProblem>     _listLoadedProblems;
        List<BaseLoadedAlgorithm>   _listLoadedAlgorithms;
        List<BaseLoadedOperator>    _listLoadedOperators;
        List<LoadedDesignVariable>  _listLoadedDesignVariables;
        List<LoadedObserverSaver>           _listLoadedObsSaver;
        List<LoadedOptimizationTerminator> _listLoadedOptTerminator;
}
```

Najvažniji razred u OptimizationContextManagementLib biblioteci je razred OptimizationProgram koji sadrži podatke kojima je potpuno definiran optimizacijski kontekst. Preciznije rečeno, OptimizationProgram sadrži podatke na osnovu kojih se kreira *izvršni kontekst* što podrazumijeva kreiranje konkretnih instanci objekata, postavljanje njihovih parametara, njihovo povezivanje i izvršavanje optimizacije.

Optimizacijski kontekst je definiran preko razreda omotača (engl. *wrappers*) koji predstavljaju pojedine vrste optimizacijskih komponenti i pomoću kojih se definira optimizacijski program. Elemente programa čine Base_Problem i njegove specijalizacije Problem_Complete, Problem_Parametrized i Problem_FileInit, Base_Algorithm sa specijalizacijam Algorithm_Complete i Algorithm_Template te ExecutionUnit. Base_Problem i Base_Algorithm predstavljaju komponentu optimizacijskog problema odnosno algoritma. Ovi razredi sadrže referencu na odgovarajući LoadedComponent objekt koji sa svoje strane sadrži podatke o konkretnoj optimizacijskoj komponenti koju dani element programa predstavlja.

Uloga ExecutionUnit omotača je centralna i on predstavlja vezu između problema i algoritma. Preciznije govoreći, unutar jednog optimizacijskog programa se može ugraditi proizvoljan broj problema i algoritama, a kreiranjem ExecutionUnit objekta koji povezuje određeni problem i algoritam, odnosno odgovarajuće elemente programa, se u biti definira primjena danog algoritma na dani problem. ExecutionUnit sadrži podatke koji detaljno opisuju kako će se obaviti primjena algoritma na dani problem. Ti podaci uključuju podatke o trajanju optimizacije i praćenju međurezultata ali i podatke o odabranoj reprezentaciji za varijable problema i podatke o odabranim operatorima koje će algoritam koristiti prilikom obavljanja optimizacije.

Način definiranja optimizacijskog programa se najbolje vidi na primjeru definiranja optimizacijskog konteksta za rješavanje DeJong_F1 problema pomoću generacijskog genetičkog algoritma. Izvršavanje ovog programa je u *run-time*u posve istovjetno izvršavanju programskog odsječka 38 u kojem je optimizacijski kontekst statički definiran.

```
static void Solve_DeJongF1_SOGenerationalGA(OptimizationComponentsManager compManager)
{
 OptimizationComponentsManager   compManager = new OptimizationComponentsManager();

 // učitavamo optimizacijske komponente iz dll-a
 compManager.LoadComponentsFromDLL
                        (@"D:\ESOP\debug\ESOP.StandardOptimizationObjectsLib.dll");
 compManager.LoadComponentsFromDLL
                        (@"D:\ESOP\debug\ESOP.BaseResultsLib.dll");

 // Kreiramo optimizacijski program
 OptimizationProgram esopProgram = new OptimizationProgram();

 // Iz OptCompManagera dohvaćamo potrebne komponente
 LoadedCompleteProblem refProblemComp = compManager.GetProblemComponentByName
            ("DeJong_F1_TestProblem") as LoadedCompleteProblem;
 LoadedTemplateAlgorithm refAlgComp = compManager.GetAlgorithmComponentByName
            ("SO_GenerationalGA") as LoadedTemplateAlgorithm;

 LoadedDesignVariable refDesVar = compManager.GetDesignVariableComponentByName
            ("RealArray") as LoadedDesignVariable;
```

```
LoadedCompleteOperator refFitAssignOper = compManager.GetOperatorComponentByName
            ("FitnesAssignment_EqualToObjValue") as LoadedCompleteOperator;
LoadedCompleteOperator refSelectionOper = compManager.GetOperatorComponentByName
            ("RouleteWheelSelectionFitness") as LoadedCompleteOperator;
LoadedParameterOperator refCrossoverOper = compManager.GetOperatorComponentByName
            ("RealArray_ArithmeticRecombinationCrossover") as LoadedParameterOperator;
LoadedCompleteOperator refMutationOper = compManager.GetOperatorComponentByName
            ("RealArray_RandomMutation") as LoadedCompleteOperator;

LoadedOptimizationTerminator refOptTerm = compManager.GetOptTerminatorComponentByName
            ("TerminateOnIterationNum");

LoadedObserverSaver refObs1 = compManager.GetObserverSaverComponentByName
            ("CurrObjectiveValueSaver");

// Zatim kreiramo odgovarajuće elemente programa
Problem_Complete      problemWrapper = new Problem_Complete(refProblemComp);
Algorithm_Template    algWrapper = new Algorithm_Template(refAlgComp);

Operator_Complete      operFitAssign = new Operator_Complete(refFitAssignOper);
Operator_Complete      operSelection = new Operator_Complete(refSelectionOper);
Operator_Parametrized operCrossover = new Operator_Parametrized(refCrossoverOper);
Operator_Complete      operMutation = new Operator_Complete(refMutationOper);

OptimizationTerminator  optTerminator = new OptimizationTerminator(refOptTerm);
ObserverSaver           obsSaver1 = new ObserverSaver(refObs1);

esopProgram.AddProblemObject(problemWrapper);
esopProgram.AddAlgorithmObject(algWrapper);

// kreirati ExecutionUnit
ExecutionUnit newEU = esopProgram.AddExecutionUnit(problemWrapper, algWrapper);

// Kreiramo kompletni kontekst za execution unit
// definiramo vrijeme optimizacije
optTerminator.DurationParameter = 20;      // 20 iteracija
newEU.AlgContext.SetOptimizationTerminator(optTerminator);

// dodajemo skup observer objekata
newEU.AlgContext.AddObserverSaver(obsSaver1);

// postavljamo odabranu reprezentaciju
newEU.SetProblemDesignVar(0, refDesVar);

// postavljamo parametre algoritma
newEU.AlgContext.SetParamValue(0, 20);      // population size

// postavljamo odabrane operatore
(newEU.AlgContext as TemplateAlgorithmContext).SetAlgorithmOperator(0, operFitAssign);
(newEU.AlgContext as TemplateAlgorithmContext).SetAlgorithmOperator(1, operSelection);

(newEU.AlgContext as TemplateAlgorithmContext).SetDesignVarOperator(0, 0, operCrossover);
(newEU.AlgContext as TemplateAlgorithmContext).SetDesignVarOperator(1, 0, operMutation);

// pokreće se izvršavanje
newEU.Execute();
```

*Programski odsječak 50 : Definiranje optimizacijskog programa kroz programski kôd*

Izvršavanje definiranog optimizacijskog programa se obavlja pozivom funkcije Execute() nad odgovarajućim objektom tipa ExecutionUnit. Osnovni zadatak funkcije Execute() je da kreira konkretne instance objekata, postavi njihove parametre, poveže ih i pokrene proces optimizacije. Način njenog rada se najbolje vidi iz samog programskog kôda:

```
public class ExecutionUnit
{
  BaseProblem         _refProblem;
  BaseAlgorithm       _refAlg;

  ProblemRepresentation       _ownSelectedRepresentation;
  AlgorithmContext            _ownAlgContext;

  public void Execute()
  {
     IESOPProblem prob = (IESOPProblem)Activator.CreateInstance
                                                  (_refProblem._ownCompDef._Type);

     IProblemInput PI = new ProblemInput(prob.ESOP_GetProblemInputDescription());

     // sada treba kreirati varijablu i pridružiti je u PI
     for (int i = 0; i < SelectedRepresentation.DesignVarNum; i++)
     {
       IESOPDesignVariable desVar = (IESOPDesignVariable)Activator.CreateInstance
                                        (SelectedRepresentation.GetSelectedDesVar(i)._Type);
       desVar.ESOP_SetVariableDescriptor( prob.ESOP_GetProblemInputDescription().
                                                                  getDesignVariableDesc(i));
       PI.AssignDesignVariable(0, desVar as IDesignVariable);
     }

     IESOPAlgorithm alg = (IESOPAlgorithm)Activator.CreateInstance
                                                    (_refAlg.GetAlgorithmObject()._Type);

     alg.ESOP_SetProblemInstance(prob);
     alg.ESOP_SetProblemInput(PI);

     // postaviti parametre algoritma
     for (int i = 0; i < AlgContext.AlgParamNum; i++ )
       (alg as IESOPCompleteAlgorithm).setParamValue(i, _ownAlgContext.GetParamValue(i));

     if (alg is IESOPTemplateAlgorithm)     {
         // postaviti operatore algoritma ...
     }

     // kreiramo i dodjeljujemo optimization terminator ...

     // kreiramo i dodjeljujemo observer savere  ....

     alg.ESOP_Run();         // pokrecemo optimizaciju

     OptimizationResult Res = (alg as IIterativeAlgorithm).GetResultRef() as OptimizationResult;
   }
  }
}
```

*Programski odsječak 51 : Funkcija Execute()*

Proučavanjem navedenih programskih odsječaka se odmah postavlja pitanje – gdje je tu vizualno sučelje? Odgovor je jednostavan – nema ga. Ili, preciznije rečeno, objekti ugrađeni u OptimizationContextManagementLib predstavljaju objektni model preko kojega je realizirana funkcionalnost potrebna za dinamički rad s optimizacijskim komponentama, ali bez realizacije vizualnih, odnosno bolje rečeno, prezentacijskih aspekata te izgradnje. Kao primarni cilj kod definiranja arhitekture ESOP ljuske je postavljeno arhitekturno razdvajanje slojeva objektnog modela i prezentacijskog sloja, u skladu s osnovnim postavkama objektno-orijentiranog oblikovanja [Mayer1988]. Razredi izgrađeni u ovoj biblioteci nemaju veze s objektima preko kojih će se oni vizualizirati u prezentacijskom sloju, odnosno unutar ESOP ljuske. Objekti prezentacijskog sloja su ugrađeni u PresentationLayer biblioteku i samu aplikaciju koja predstavlja izvršnu aplikaciju ESOP ljuske.

#### 5.6.2.4 ESOPApp aplikacija i PresentationLayer biblioteka

Vizualno sučelje ESOP ljuske je izgrađeno kroz izvršnu aplikaciju koja je izgrađena kao standardna Windows aplikacija u kojoj se interakcija s korisnikom obavlja kroz GUI sučelje. Osnovna funkcionalnost je korisniku dostupna kroz sustav izbornika preko kojih se obavljaju određene akcije prilikom korištenja aplikacije. Osnovne akcije definirane kroz stavke u pripadnim izbornicima su:

- kreiranje novog optimizacijskog programa
- učitavanje / snimanje optimizacijskog programa
- učitavanje optimizacijskih komponenti
- pregled rezultata

Sadržajem izbornika nije definiran potpuni skup funkcionalnosti biblioteke jer je određeni skup funkcionalnosti dostupan korisniku kroz ostale elemente GUI sučelje. Pored izbornika, vizualno sučelje čine komponenta koja vizualizira skup učitanih optimizacijskih komponenti i skup prezentacijskih komponenti koje su ugrađene u PresentationLayer biblioteku i preko kojih se obavlja vizualizacija optimizacijskog programa.

##### 5.6.2.4.1 Komponenta za prikaz učitanih optimizacijskh komponenti

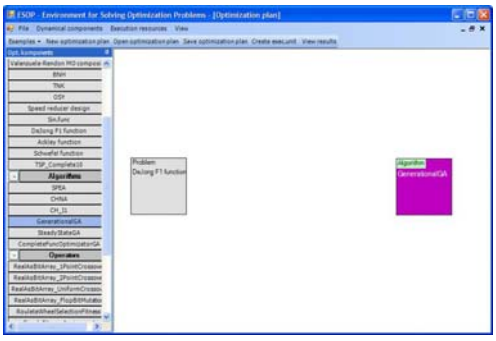
Ova komponenta je izgrađena u stilu trake s alatima (engl. *toolbar*) gdje je svaka učitana optimizacijska komponenta predstavljena preko Button vizualizacijske kontrole. Osnovna zadaća ove komponente, nazvane ctrlOptComponentsToolbar, je da vizualizira sadržaj OptCompManager objekta koji sadrži podatke o učitanim optimizacijskim komponentama. ctrlOptComponentsToolbar sadrži referencu na instancu objekta OptCompManager, koji je dio ESOPApp aplikacije, i prilikom učitavanja novih optimizacijskih komponenti obavlja osvježavanje kontrole, odnosno kreiranje novih Button kontrola kao reprezentacija za novoučitane optimizacijske komponente. Druga uloga ctrlOptomponentsToolbar komponente je da u interakciji s formom za vizualizaciju optimizacijskog programa (opisanom u sljedećem odjeljku) omogući dodavanje pojedinih optimizacijskih komponenti u optimizacijski program.

#### 5.6.2.4.2 PresentationLayer biblioteka

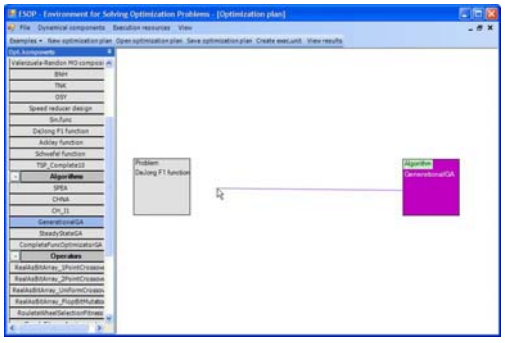
U ovoj biblioteci su ugrađeni razredi koji predstavljaju vizualizacijske kontrole za određene objekte definirane u objektnom modelu koji je realiziran u OptimizationContextManagementLib biblioteci.

Osnovni element ove biblioteke je forma frmOptimizationProgram preko koje se vizualizira kompletan optimizacijski program. Forma frmOptimizationProgram u sebi sadrži referencu na OptimizationProgram objekt što joj omogućava vizualizaciju sadržaja definiranog optimizacijskog programa na ekran. Međutim, nužno je i postojanje suprotnog smjera komunikacije, od objekta prema formi, kako bi se prilikom promjena u objektnom modelu mogao ažurirati sadržaj prezentacijskog sloja. Ta funkcionalnost je postignuta korištenjem *Model-View-Controller* obrasca oblikovanja. Za svaki objekt u objektnom modelu se definira razred *kontrolera* preko kojega se programski pokreću sve akcije vezane uz objekt. Kontroler ima referencu i na konkretan objekt i na vizualizacijsku kontrolu preko koje se taj objekt reprezentira u prezentacijskom sloju. Detaljan opis *Model-View-Controller* obrasca se može naći u [Gamma2004].

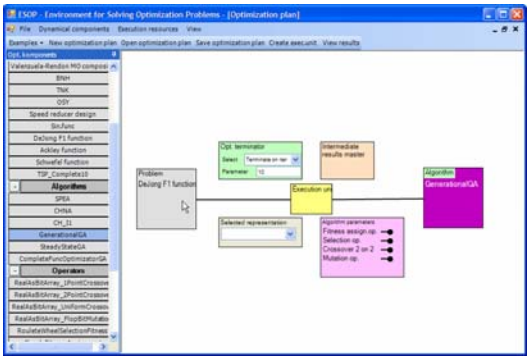
Pored forme za vizualizaciju optimizacijskog programa drugi važan element PresentationLayer biblioteke čini skup vizulizacijskih kontrola preko kojih se vizualiziraju određeni elementi optimizacijskog programa. Za svaki element programa je izgrađena odgovarajuća korisnička kontrola koja se prilikom stavljanja tog elementa u optimizacijski program instancira unutar forme frmOptimizationProgram na njenoj radnoj površini (engl. *client area*). Pojedine kontrole pružaju funkcionalnost povezanu s elementom programa kojeg predstavljaju. Osnovnu ulogu, u skladu s dizajnom objektnog modela kojim je programski modeliran optimizacijski program, ima ctrlExecutionUnit kontrola koja vizualizira ExecutionUnit objekt. Kreiranje pojedine jedinice izvršavanja se obavlja jednostavnim povezivanjem kontrola koje predstavljaju problem i algoritam. Prilikom kreiranja ctrlExecutionUnit kontrole se obavlja i kreiranje kontrola koje predstavljaju ostale aspekte optimizacijskog programa: definiranje trajanja optimizacije, odabir reprezentacije za ulazne varijable problema, odabir operatora te odabir međurezultata koji će se pratiti tijekom izvođenja optimizacije. Proces kreiranja jedinice izvršavanja je prikazan na slikama 5.20 – 5.23.



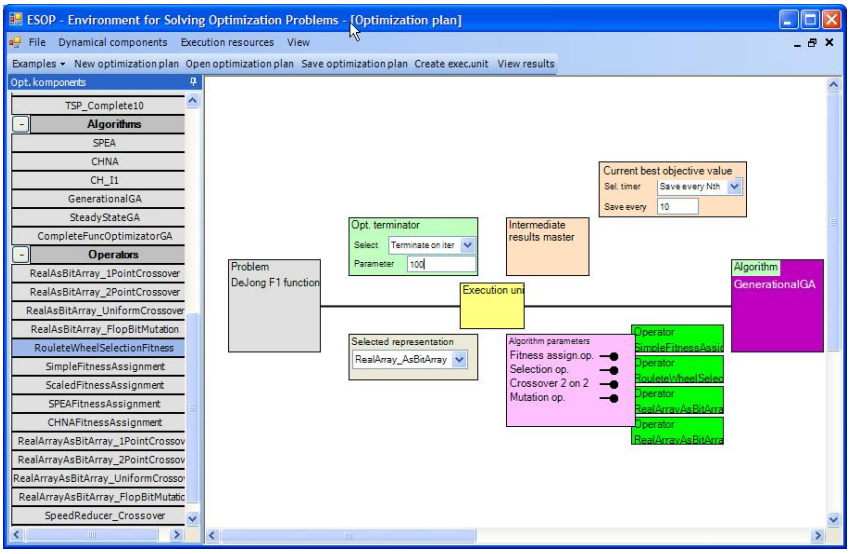
**Slika 5.20 Dodavanje problema i algoritma**



**Slika 5.21 Povezivanje problema s algoritmom (kreiranje izvršne jedinice)**

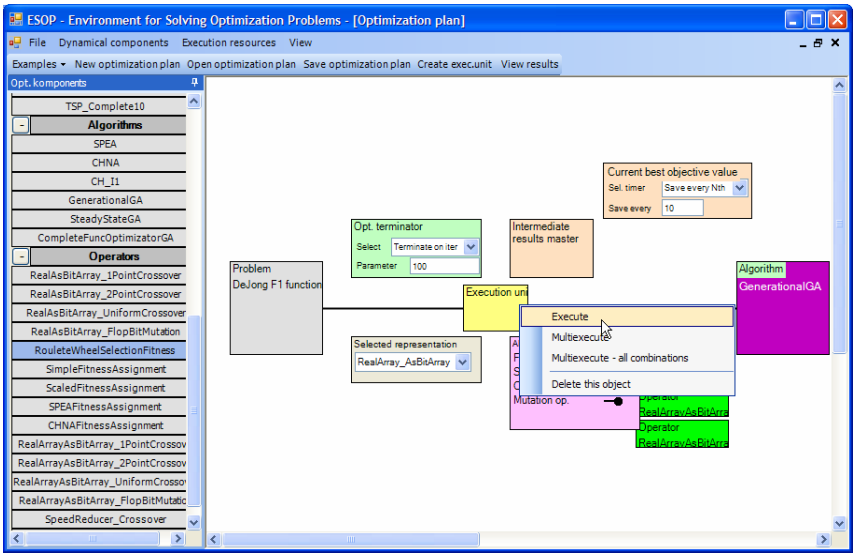


**Slika 5.22 Automatski kreirani skup vizualnih kontrola povezanih s izvršnom jedinicom**




**Slika 5.23 Potpuno definiran optimizacijski program**

Pokretanje optimizacije se obavlja preko *pop-up* menija ugrađenog u ctrlExecutionUnit kontrolu, kako je prikazano na slici 5.24.




**Slika 5.24 Pokretanje optimizacije**

Nakon obavljanja optimizacije, što se u konačnici svodi na poziv funkcije Execute() iz ExecutionUnit razreda, rezultati optimizacije se mogu vidjeti kroz frmResults formu, koja je također ugrađena u PresentationLayer biblioteku. Ona u sebi sadrži rezultate svih optimizacija provedenih u pripadnom optimizacijskom programu, preko liste OptimizationResults objekata, i vizualizira ih preko List kontrole. Klikom na odgovarajući redak se otvara odgovarajuća kontrola iz ResultsVisualizationControls biblioteke (frmResultSO, frmResultMO, ...) nakon čega je moguće daljnje proučavanje dobivenih rezultata, kako je to opisano odjeljku 5.3.3.

## 5.6.3 Prikaz korištenja

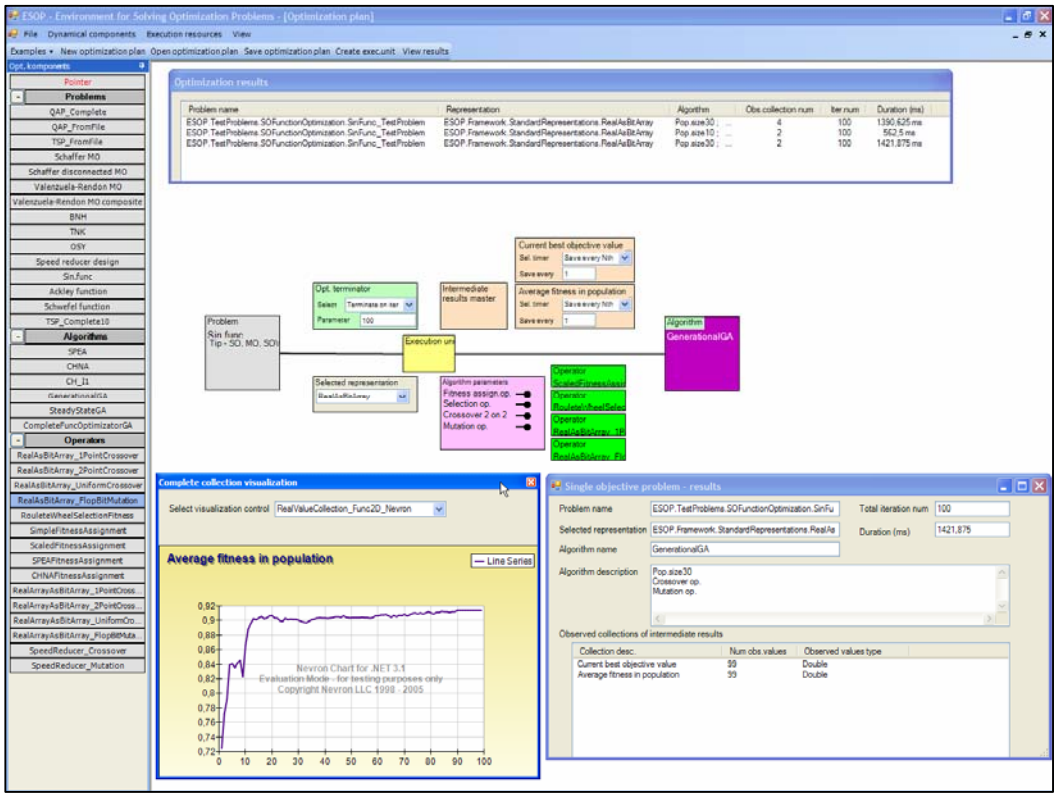
Izgrađena ESOP ljuska omogućava vrlo jednostavno definiranje optimizacijskog programa preko vizualnog sučelja. Ukoliko su potrebne optimizacijske komponente već izgrađene i učitane u ESOP ljusku, optimizator može provesti cjelokupni proces optimiranja bez napuštanja vizualnog okruženja i bez potrebe za dodatnim programiranjem.

Zahvaljujući fleksibilnoj i proširljivoj arhitekturi ESOP razvojne okosnice, izgradnja novih komponenti je maksimalno jednostavna i uz korištenje Visual Studio razvojnog okruženja i njegovih mogućnosti za automatizirano kreiranje infrastrukture programskog rješenja pomoću čarobnjaka zahtjeva minimalan trud i poznavanje razvojnog okruženja od strane korisnika (optimizatora).

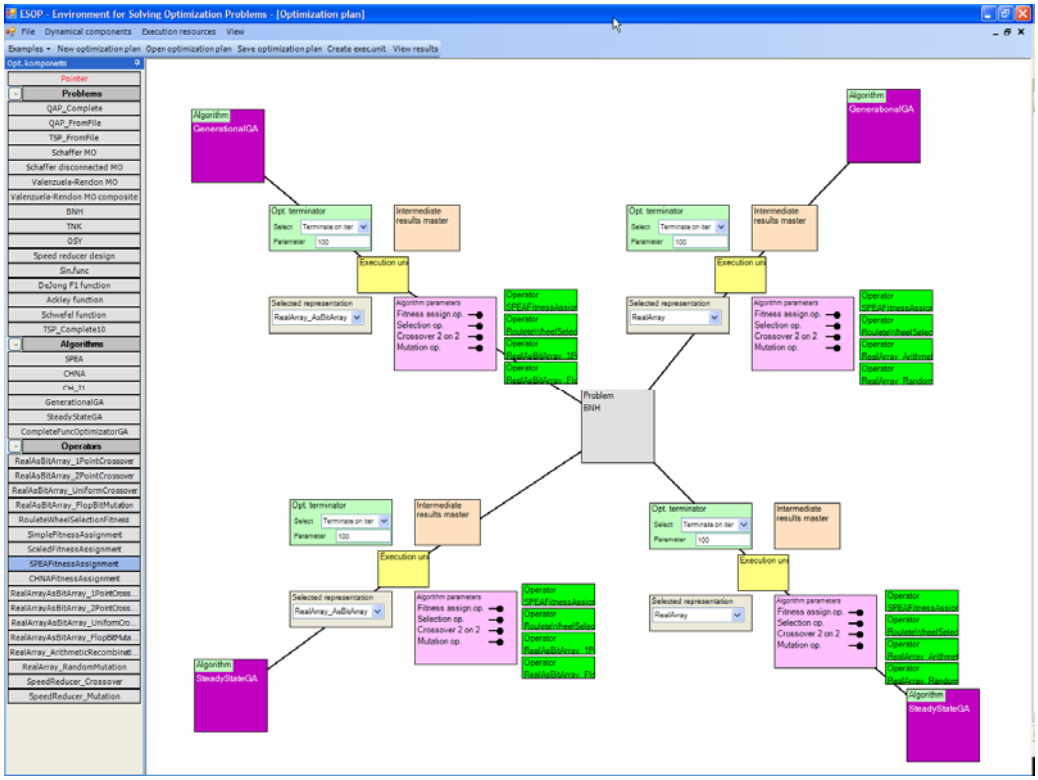
Mogućnost korištenja različitih jezika unutar .NET razvojnog okruženja za ugradnju novih optimizacijskih komponenti je dodatna prednost za korisnike okruženja jer omogućava korištenje programskih jezika koji su svojom sintaksom jednostavniji i samim time lakši za savladavanje od C++-a, a broj podržanih jezika za razvoj u .NET okruženju se povećava velikom brzinom.

Ugrađene mogućnosti za vizualizaciju i usporedbu rezultata i međurezultata optimizacije su dostupne jednim klikom miša. Prikaz dijela vizualnih komponenti ugrađenih u ESOP ljusku je dan na slici 5.25 gdje su prikazane forme za vizualizaciju rezultata i međurezultata kod primjene generacijskog genetičkog algoritma na problem jednokriterijske optimizacije.

Mogućnost jednostavnog uspoređivanja različitih optimizacijskih postupaka je ilustrirana na slici 5.26 gdje je je prikazan jedan složeni optimizacijski program u kojem su definirane četiri izvršne jedinice za rješavanje BNH višekriterijskog problema nastale kombiniranjem dvaju vrsta genetičkih algoritama (GenerationalGA i SteadyStateGA) i dvije vrsta reprezentacija (RealArray i RealArray_AsBitArray). Uz već opisanu mogućnost vizualizacije rezultata i međurezultata optimizacije i mogućnost podešavanja parametara optimizacije kroz vizualno sučelje, posao istraživača je značajno pojednostavljen.




**Slika 5.25 Prikaz ESOP okruženja 1 – komponente za vizualizaciju rezultata**




**Slika 5.26 Prikaz ESOP okruženja 2 – složeni optimizacijski program s primjenom različitih algoritama na rješavanje danog optimizacijskog problema**

# 6. Primjena ESOP-a na problem određivanja optimalnih parametara mehaničke strukture broda

U prvih pet poglavlja ove disertacije je dan teorijski opis domene optimizacije te je detaljno opisano izgrađeno ESOP optimizacijsko okruženje i prikazana njegova primjena na rješavanje standardnih problema optimizacije. U ovom poglavlju će se prikazati primjena razvijenog optimizacijskog okruženja na rješavanje složenog problema iz prakse.

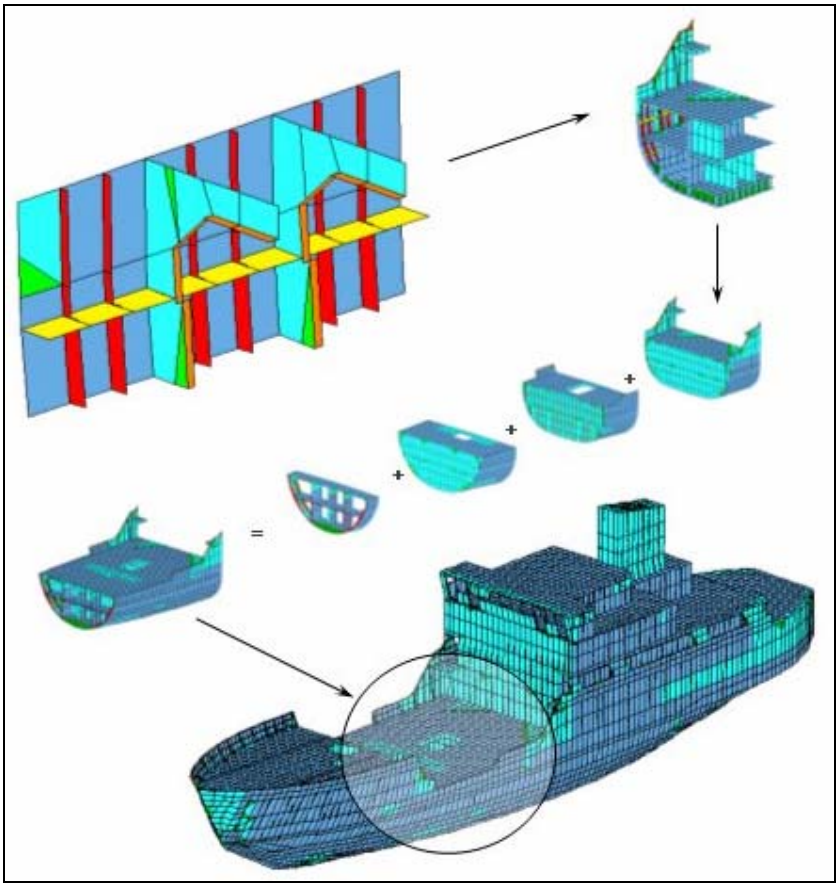
Problem za rješavanje kojega će se iskoristiti ESOP okruženje je problem optimizacije parametara mehaničke strukture broda. Rješavanje ovog problema optimizacije je izvedeno u sklopu tehnologijskog projekta „Integrirani programski sustav za brodske konstrukcije“ Ministarstva znanosti i tehnologije, financiranog u iznosu 1.478.500,00 kn [MZT2006]. Kao osnovni cilj projekta je postavljeno povećanje konkurentnosti hrvatske brodogradnje kroz implementaciju integriranog programskog sustava za: (a) definiranje funkcionalno efikasne konstrukcije broda, te (b) ubrzanje izrade projektne dokumentacije.

Glavni izvođač na projektu je bila skupina istraživača s Katedre za konstrukciju plovnih objekata na Fakultetu strojarstva i brodogradnje, na čelu s prof.dr.sc. Vedranom Žanićem. Kao podizvođač je na projektu sudjelovala i skupina istraživača sa Zavoda za primjenjeno računarstvo u sastavu prof.dr.sc. Damir Kalpić, prof.dr.sc Vedran Mornar i mr.sc. Zvonimir Vanjak sa zadatkom razvoja programske komponente za provođenje višekriterijske optimizacije korištenjem genetičkih algoritama.

## 6.1 Problem definiranja konstrukcije broda

Projektiranje brodske konstrukcije je vrlo složen tehnički i tehnološki problem i detaljan opis izlazi van okvira ove disertacije. Dobar pregled se može naći u [Žanić2002] a u ovom potpoglavlju će se dati samo okvirni pregled tog problema koliko je nužno za razumijevanje opisa razvoja programske komponente za optimizaciju pomoću genetičkih algoritama.

Projektiranje brodskih konstrukcija se danas gotovo univerzalno provodi korištenjem programskih alata koji omogućavaju izradu trodimenzionalnog modela broda i ispitivanje karakteristika tako definirane konstrukcije u virtualnom okruženju. Prednost takvog pristupa je značajno smanjenje troškova projektiranja. Model broda se gradi korištenjem dobro definiranih osnovnih mehaničkih struktura (vojevi, ukrepe, nosači i paneli u brodostrojarskoj terminologiji), za koje su poznate fizikalne karakteristike i njihov odziv na različite vrste opterećenja, nakon čega se tako izgrađeni model podvrgava proračunavanju naprezanja korištenjem metode konačnih elemenata (engl. *Finite Element Method* – FEM). Prikaz jednog takvog modela, zajedno s načinom razlaganja na osnovne strukture je dan na slici 6.1.



**Slika 6.1 Prikaz modela broda (preuzeto iz [MAESTRO2006])**

Osnovni zadatak kod projektiranja broda je stoga određivanje vrijednosti parametara kojima su te strukture potpuno definirane. Tipični primjeri tih parametara su širina, visina i debljina ukrepe i čvrstoća korištenog materijala a vrijednosti tih parametara je potrebno podesiti tako da brodska konstrukcija bude optimalna.

Najčešće se postavljaju dva osnovna kriterija optimalnosti, čime određivanje optimalnog skupa vrijednosti parametara prelazi u domenu višekriterijske optimizacije. Prvi kriterij optimalnosti je minimizacija težine cjelokupne strukture a drugi je maksimizacija sigurnosti konstrukcije. Prvi kriterij proizlazi iz direktne veze između težine strukture, koja je određena količinom čelika potrebnog za njenu izgradnju, i cijene izgradnje broda. S obzirom na veliku konkurenciju na području brodogradnje koja doista predstavlja globaliziran biznis, postizanje što manjih troškova je imperativ uspjeha na globalnom tržištu.

S time je konfliktan drugi kriterij optimizacije, a to je postizanje što veće sigurnosti konstrukcije broda. Sigurnost broda se proračunava preko definiranih konstrukcijskih i dizajnerskih kriterija na osnovu postavljenih parametara mehaničke strukture. Pojednostavljeno rečeno, proračunom sigurnosti za danu konstrukciju se dobije mjera koliko je ta konstrukcija otporna na različita naprezanja.

Pored ova dva osnovna kriterija optimalnosti, kod projektiranja konstrukcije broda postoji i veliki skup ograničenja kojima su definirani zahtjevi koje ta konstrukcija mora zadovoljavati. Neka od tih ograničenja modeliraju jednostavne

funkcijske odnose između različitih parametara mehaničke strukture, a neka su složenije prirode i proizlaze iz proračuna pomoću metode konačnih elemenata.

Uzevši sve navedeno u obzir, vidljivo je da problem nalaženja optimalnih parametara mehaničke strukture broda predstavlja složeni problem višekriterijske optimizacije s ograničenjima.

## 6.2 Izgradnja programskog rješenja

Kako je u uvodu već navedeno, kao osnovni zadatak skupine istraživača na FER-u je postavljena izgradnja programske komponente za rješavanje problema optimizacije mehaničkih parametara brodske konstrukcije. S obzirom da izračunavanje vrijednosti definiranih funkcija cilja i ograničenja (izlaz optimizacijskog problema) zahtijeva proračun mehaničkih naprezanja korištenjem metode konačnih elemenata, arhitektura programskog rješenja je nešto drugačija u odnosu na primjere prikazane u prethodnim poglavljima ove disertacije.

U do sada prikazanim primjerima korištenja ESOP-a, iskorištavanje ESOP optimizacijskog okruženja je podrazumijevalo izgradnju aplikacije koja će za provođenje optimizacije koristiti optimizacijske komponente ugrađene u biblioteke razreda koji su dio ESOP-a. S obzirom na široki skup optimizacijskih komponenti koji je standardno ugrađen u ESOP, u velikom broju slučajeva se zadatak korisnika-istraživača svodi na definiranje razreda / objekta kojim će se definirati zadani optimizacijski problem. Osnovna uloga tog objekta je da na osnovu zadanih vrijednosti ulaznih varijabli proračuna pripadni skup vrijednosti funkcija cilja i zadanih ograničenja.

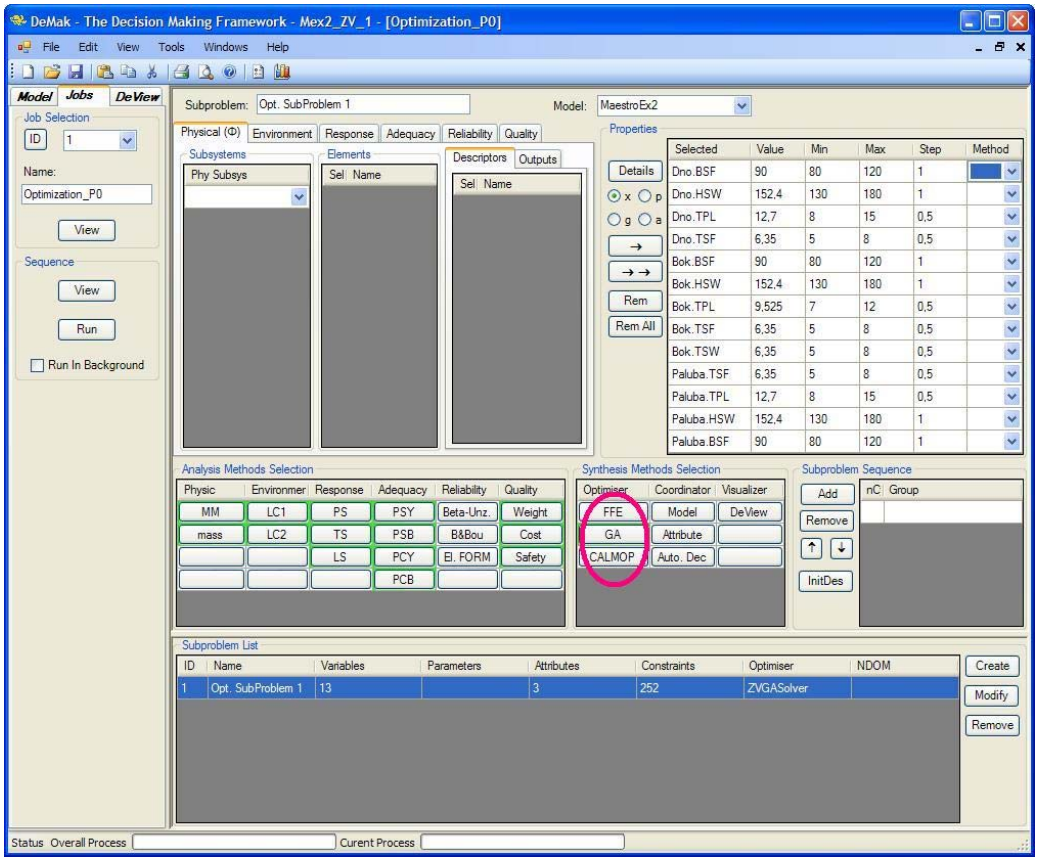
U ovom slučaju bi to značilo ugradnju cjelokupne funkcionalnosti za provođenje FEM proračuna u taj razred. Uz to bi u tu komponentu bilo potrebno ugraditi i mogućnost definiranja cijele brodske konstrukcije (što predstavlja ulazni podatak prilikom provođenja proračuna pomoću FEM metode). Kako je izgradnja programskog rješenja s takvom funkcionalnošću upravo i postavljena kao cilj cjelokupnog projekta „Integrirani programski sustav za brodske konstrukcije“ pokazala se potreba za kreiranjem nešto drugačije arhitekture.

### 6.2.1 Arhitektura programskog rješenja

Kako je prethodno opisano, komponenta za optimizaciju parametara mehaničke strukture brodske konstrukcije mora imati pristup funkcionalnosti za proračunavanje mehaničkih naprezanja pomoću FEM metode. Ta funkcionalnost je ugrađena u DeMak aplikaciju, koje je razvila skupina istraživača s Fakulteta strojarstva i brodogradnje.

DeMak (skraćenica od engl. *Decision Maker*) predstavlja okruženje za potporu donošenju odluka kod odabira parametara mehaničke strukture brodske konstrukcije. DeMak okruženje je izgrađeno kao .NET aplikacija i predstavlja ljusku unutar koje projektant broda provodi proces optimiranja brodske konstrukcije. Unutar DeMak ljuske je ugrađena funkcionalnost za definiranje optimizacijskog problema, što podrazumijeva učitavanje opisa konstrukcije broda (znači, učitavanje trodimenzionalnog strukturnog modela broda) a za proračunavanje FEM metodom se oslanja na CREST programsku komponentu koja je realizirana u Fortran

programskom jeziku. Samo definiranje konstrukcije se obavlja pomoću MAESTRO sustava [MAESTRO2006]. Prikaz DeMak ljuske s učitanim problemom optimizacije jednog panela (osnovni dio strukture brodske konstrukcije) je dan na slici 6.2.




**Slika 6.2 Prikaz DeMak ljuske**

Osnovni zadatak DeMak ljuske je pružiti konstruktoru mogućnost provođenja optimizacije brodske konstrukcije kroz jedno sučelje. Stoga je arhitektura samog DeMak-a postavljena fleksibilno, s mogućnošću ugrađivanja različitih optimizacijskih postupaka. Optimizacijski postupci se implementiraju kao zasebne programske komponente koje sa DeMak aplikacijom komuniciraju preko skupa definiranih sučelja. Na slici 6.2. je označena dio prozora DeMak ljuske unutar kojega se vidi skup ugrađenih optimizacijskih komponenti.

## 6.2.2 Interakcija optimizacijskih komponenti s DeMak ljuskom

Ugradnja optimizacijskog postupka za rješavanje opisanog problema optimizacije se svodi na izgradnju razreda izvedenog iz apstraktnog razred Optimizer koji je definiran u DeMakNET.dll biblioteci i koji ima sljedeću definiciju:

```
public abstract class Optimizer
{
  protected List<Attribute> attributes;
  protected List<Constraint> constraints;
  protected List<DesignVariable> designVariables;
```

```
  protected List<InitialDesign> initialDesigns;
  protected int numModelRuns;
  protected OptimizerControl optControl;
  protected OptOutputMulticriterial optOutput;
  protected List<OuterConstraint> outerConstraints;
  protected SubProbSimModel spsmodel;

  public Optimizer();
  public Optimizer(OptimizerControl _optimizerControl);

  public int NumModelRuns { get; }
  public OptimizerControl OptimizerControl { get; set; }
  public OptOutputMulticriterial OptOutput { get; }

  public Attribute GetAttribute(int i);
  public Constraint GetConstraint(int i);
  public int GetNumAttributes();
  public int GetNumConstraints();
  public int GetNumOuterConstraints();
  public int GetNumVariables();
  public DesignVariable GetVariable(int i);

  public abstract int Optimize();
  public abstract int Optimize(OptimizerStatusInfo optStatus,
                               BackgroundWorker bgWork,
                               DoWorkEventArgs e);

  public void ResetNumModelRuns();
  public int RunFirstCheck();
  public int RunSimulation();
  public void SetOptimizationComponents(List<DesignVariable> dv, List<Attribute> att,
                                        List<Constraint> con, List<InitialDesign> inDes,
                                        List<OuterConstraint> outConstr);
  public void SetSubProbSimulationModel(SubProbSimModel _spsmodel);
}
```

Unutar ovog apstraktnog razreda je ugrađena cjelokupna infrastruktura potrebna za definiranje optimizacijskog problema. Članske varijable sadrže popise ulaznih varijabli (ovdje nazvanih dizajn varijablama), funkcija cilja (atributi) i ograničenja a proračunavanje izlaza se provodi pozivom funkcije RunSimulation(). Unutar te funkcije je ugrađen programski kôd koji korištenjem metode konačnih elemenata proračunava naprezanja u konstrukciji broda te na osnovu toga izračunava vrijednosti funkcija cilja i ograničenja.

Graditelj novog optimizacijskog postupka mora samo u svom izvedenom razredu definirati implementaciju apstraktne funkcije Optimize(), odnosno unutar te funkcije ugraditi algoritam za provođenje optimizacije.

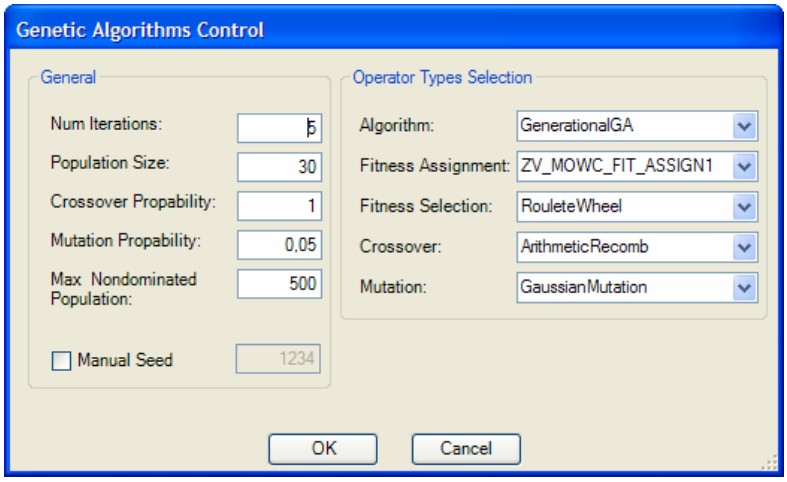
## 6.2.3 Ugradnja optimizacijskih komponenti

S obzirom da je izrada cjelokupnog ESOP optimizacijskog okruženja potaknuta upravo rješavanjem ovog problema, zbog čega je i naglasak u opisu izgradnje stavljen na rješavanje višekriterijskih problema, optimizacijske komponente potrebne za rješavanje su već ugrađene u ESOP (i opisane ranije u prethodnim poglavljima disertacije).

Stoga je i izgradnja programske komponente za optimizaciju pomoću genetičkih algoritama bila vrlo jednostavna. Izgrađena je biblioteka razreda ZVGASolver.dll unutar koje su definirana dva razreda: DeMakProblem i ZVGASolver.

Razred DeMakProblem je izveden iz razreda ProblemMOWC i predstavlja optimizacijski problem, dok je razred ZVGASolver izveden iz razreda Optimizer i predstavlja komponentu unutar koje je realiziran optimizacijski postupak. Izgradnja DeMakProblem razreda je vrlo jednostavna i slijedi programske tehnike opisane kod proširivanja ESOP-a s novim problemima (odjeljak 5.5.1.1), uz jednu specifičnost. Naime, kako je optimizacijski problem definiran sadržajem gore opisanog razreda Optimizer, razred DeMakProblem mora imati referencu na razred ZVGASolver preko koje će dohvatiti potrebne podatke (broj i vrsta ulaznih varijabli, definicija funkcija cilja i postavljenih ograničenja) i obaviti proračunavanje izlaza (pozivom funkcije RunSimulation()).

Razred ZVGASolver ne predstavlja standardni ESOP razred optimizacijskog algoritma (koji su karakterizirani izvođenjem iz IterativeAlgorithm apstraktnog razreda) već programsku komponentu koja koristi razrede ugrađene u ESOP a koji realiziraju pojedinačne optimizacijske postupke. ZVGASolver stoga predstavlja konfigurabilnu optimizacijsku komponentu unutar koje se kreira optimizacijski kontekst odabirom odgovarajući objekata optimizacijskog algoritma i njemu pripadnih operatora. Stoga je izgrađen poseban razred ZVGAControl preko kojega DeMak okruženje omogućava odabir pojedinih komponenti koje se odabiru preko forme (koja je dio DeMak ljuske) prikazane na slici 6.3.



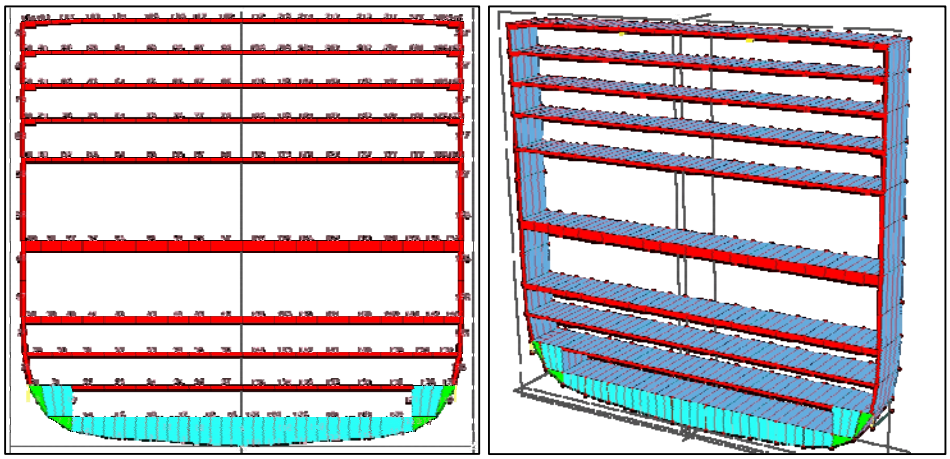

**Slika 6.3 Forma za upravljanje parametrima optimizacijske komponente**

Unutar implementacije funkcije Optimize() se na početku kreiraju svi potrebni optimizacijski objekti (u skladu sa sadržajem ZVGAControl objekta) nakon čega se pokreće proces optimizacija (pozivom funkcije Run() nad kreiranim objektom optimizacijskog algoritma). Na kraju se rezultati prenose u DeMak okruženje korištenjem razreda OptOutputMulticriterial, koji je definiran unutar DeMak okruženja. Izgradnja cjelokupne opisane programske komponente je obavljena u otprilike 700 linija programskog koda.

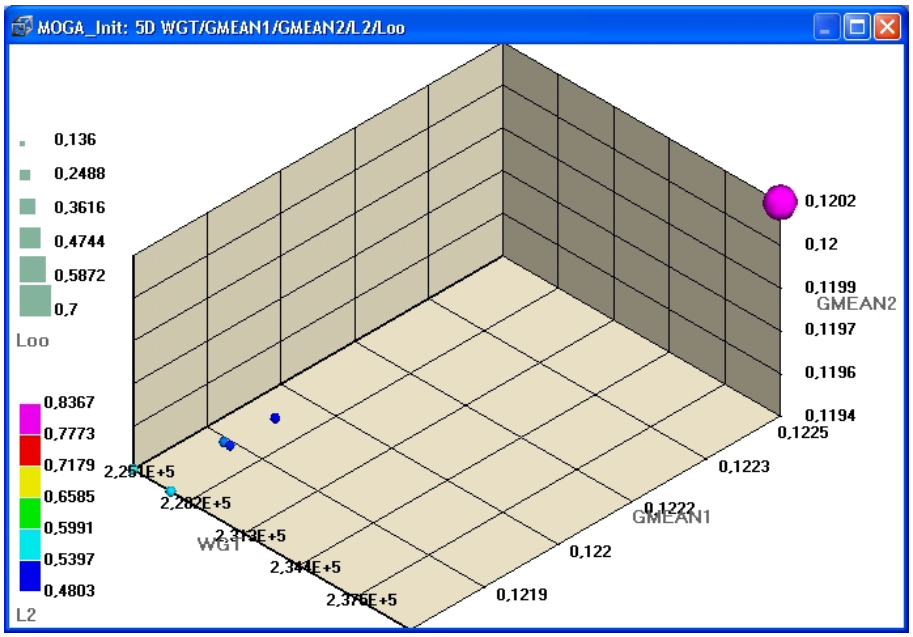
## 6.3 Rezultati

Razvijena programska komponenta za optimizaciju pomoću genetičkih algoritama je unutar DeMak ljuske primijenjena za rješavanje različitih problema. S obzirom da je DeMak ljuska projektirana na način da podržava učitavanje i optimiranje različitih vrsta mehaničkih konstrukcija, genetički algoritmi su primjenjeni za rješavanje problema različite složenosti.

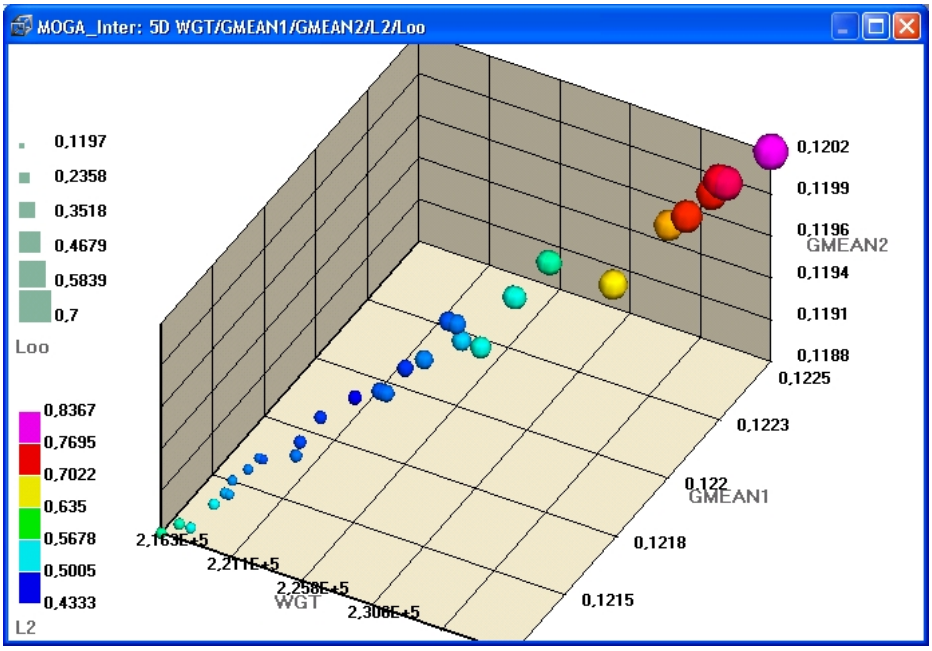
Od optimizacije jednostavnog panela, što je predstavljalo problem s 13 varijabli, 3 funkcije cilja i 252 ograničenja, do optimizacije cijelog presjeka broda što je predstavljalo vrlo složen problem sa 635 varijabli, 3 funkcije cilja i 2500 ograničenja. Izgled fizičke konstrukcije tog presjeka je dan na slici 6.4, gdje se vidi raspored pojedinačnih mehaničkih elemenata koji čine konstrukciju, dok je na slikama 6.5 – 6.7 prikazan razvoj skupa Pareto rješenja tijekom optimizacije.



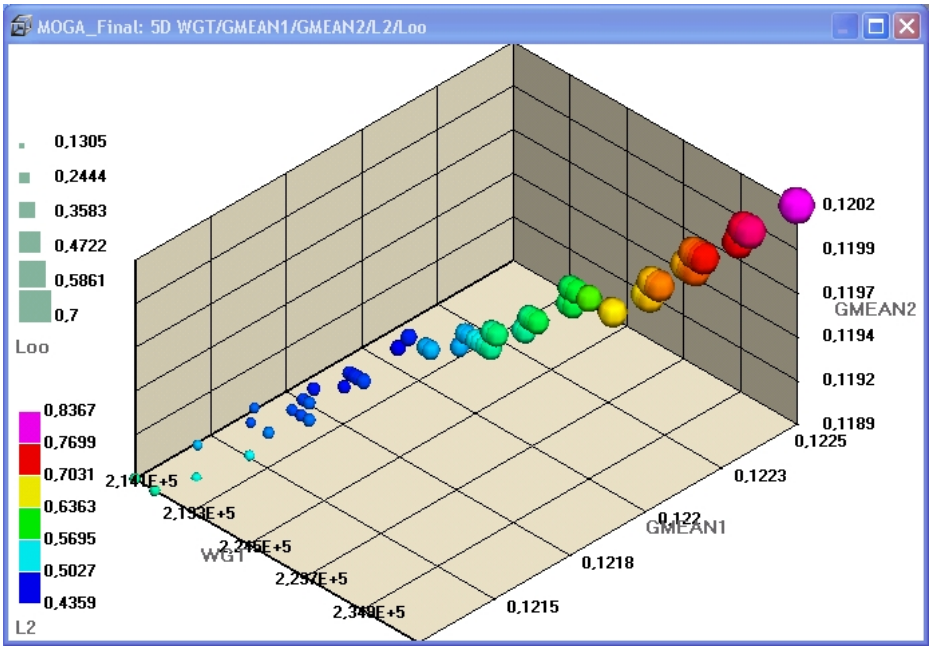
**Slika 6.4 Izgled konstrukcije broda**




**Slika 6.5 Skup Pareto rješenja u početku optimizacije**




**Slika 6.6 Skup Pareto rješenja pri sredini optimizacije**




**Slika 6.7 Skup Pareto rješenja na kraju optimizacije**

# 7. Zaključak

U okviru ove disertacije je na temelju analize domene optimizacije kroz koju su identificirani pojedini elementi te domene i njihove interakcije, te na temelju toga izrađenom konceptualnom modelu, oblikovano i izgrađeno općenito optimizacijsko okruženje ESOP (***E****nvironment for* ***S****olving* ***O****ptimization* ***P****roblems*).

Analizom postojećih okruženja za optimizaciju je ustanovljeno da u pojedinim aspektima ne zadovoljavaju potrebe istraživača kod provođenja eksperimentalne analize heurističkih optimizacijskih postupaka i njihove primjene na probleme u praksi. Kao primarni nedostatak postojećih optimizacijskih okruženja se pokazuje nepostojanje vizualnog sučelja kroz koje se može obavljati proces optimizacije. S obzirom da je većina tih rješenja izgrađena kroz biblioteke razreda, iskorištavanje u njih ugrađenih optimizacijskih komponenti zahtjeva izradu novog programskog rješenja što od strane istraživača zahtjeva poznavanje, često i naprednih tehnika programiranja. S obzirom na veliki broj različitih razvojnih okruženja u kojima su izgrađena postojeća rješenja, istraživači se često suočavaju sa situacijom da prikladno postojeće programsko rješenje nije iskoristivo zbog nepoznavanja razvojnog okruženja u kojem je ono razvijeno.

Pored navedenoga, većina postojećih optimizacijskih okruženja, iako ne i sva, omogućava primjenu samo jedne vrste optimizacijskih postupaka. S obzirom na dobro poznati „*No Free Lunch*“ teorem [Wolpert1997] koji dokazuje da ne postoji optimizacijski postupak koji bi bio najbolji za sve vrste optimizacijskih problema, posvećenost samo jednoj vrsti optimizacijskih postupaka je značajan nedostatak postojećih programskih rješenja.

Znanstveni doprinos je ostvaren klasifikacijom koncepata iz cjelokupne domene optimizacije, što obuhvaća optimizacijske probleme i postupke optimizacije primjenjive na njihovo rješavanje. Na osnovu analize njihovih karakteristika i njihovih međusobnih interakcija, formiran je konceptualni model domene koji je poslužio kao temelj za oblikovanje općenitog optimizacijskog okruženja.

Korištenjem tehnika objektno-orijentirane paradigme je izgrađena razvojna okosnica koja omogućava ugradnju širokog skupa različitih optimizacijskih problema i optimizacijskih postupaka. Oblikovan je, i u vidu ESOP ljuske izgrađen, univerzalni prezentacijski sloj kroz koji se definiranje cjelokupnog procesa optimizacije obavlja preko vizualnog sučelja, i u koji su ugrađene mogućnosti za vizualizaciju i analizu rezultata i međurezultata optimizacije.

Razvijeno ESOP optimizacijsko okruženje je primijenjeno na rješavanje složenog problema optimizacije parametara mehaničke strukture brodske konstrukcije u sklopu tehnologijskog projekta „Integrirani programski sustav za brodske konstrukcije“. Iskorištavanjem ugrađenih optimizacijskih komponenti za provođenje višekriterijske optimizacije pomoću genetičkih algoritama problem je efikasno riješen uz minimalan trud uložen u izgradnju specifičnih programskih komponenti vezanih konkretno uz taj problem.

Unatoč tome što je rad na ovoj disertaciji rezultirao zaokruženim i u praktičnoj primjeni testiranim okruženjem za optimizaciju, postoje i mnoge dodatne teme za daljnja istraživanja.

S obzirom na numeričku zahtjevnost provođenja optimizacije putem heurističkih algoritama, zanimljiv i potencijalno vrlo koristan pravac istraživanja predstavlja ugrađivanje mogućnosti za paralelizirano provođenje optimizacije, bilo na više računala povezanih u mreži bilo na višeprocesorskom računalu. Uzevši u obzir da su mnogi heuristički optimizacijski postupci bazirani na operiranju nad populacijom rješenja, potrebne prilagodbe u izrađenom konceptualnom i objektnom modelu ne bi bile prevelike, pogotovo ukoliko se uzme u obzir kvaliteta podrške za distribuirano računarstvo u .NET razvojnom okruženju.

Imajući u vidu konstantnu potrebu za uspoređivanjem performansi različitih algoritama na testnim problemima, dodatni pravac istraživanja bi mogao ići u smjeru oblikovanja i izgradnje centralnog repozitorija optimizacijskih komponenti koje bi bile lako dostupne i lako ugradive u različita optimizacijska i razvojna okruženja. Iako dijeljenje izvršnih programskih komponenti predstavlja složeni izazov, poglavito zbog povezanih sigurnosnih aspekata, moderna razvojna okruženja (.NET), koja su od početka projektirana s ciljem garantiranja sigurnosti izvršavanja u skladu sa sigurnosnim kontekstom, omogućavaju, barem teorijski, rješavanje i tog problema, a korištenje XML-a kao univerzalnog formata za razmjenjivanje podataka omogućava, uz definiranje odgovarajućih struktura formata, razmjenjivanje definicija optimizacijskih problema i postignutih rezultata optimizacije.

Mogući pravac daljnjeg istraživanja je i razvoj *jezika specifičnog za domenu* (engl. *Domain Specific Language*). Izgrađeno ESOP okruženje pruža vizualno sučelje za definiranje optimizacijskih programa i sa stajališta jednostavnosti korištenja je to izvrsno rješenje. Međutim, korisnik je u tome donekle ograničen jer se optimizacijski programi mogu definirati samo kroz predefinirane elemente vizualnog sučelja. Oblikovanje i izgradnja jezika koji bi sadržavao skup jednostavnih programskih konstrukta za definiranje optimizacijskih programa i istovremeno omogućavao potpuni pristup izgrađenim programskim komponentama predstavlja vrlo obećavajući pravac istraživanja koji, barem po mišljenju autora, zahtjeva vlastitu disertaciju.

# 8. Literatura


[Ackley1987] D.H. Ackley, *A Connectionist Machine for Genetic Hillclimbing*, Kluwer, Boston, 1987.

[Anić1999] V.Anić, I.Goldstein, *Rječnik stranih riječi*, Novi Liber, Zagreb, 1999.

[Arenas2002a] M. G. Arenas, B. Dolin, J. J. Merelo, P. A. Castillo, I. Fdez de Viana, i M. SchÄonauer. *JEO: Java evolving objects*. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* 2002.

[Arenas2002b] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. Merelo, B. PÄachter, M. Preus, i M. Schonauer. *A framework for distributed evolutionary algorithms*. U *Parallel Problem Solving from Nature (PPSN VII)*, volume 2439 of *Lecture Notes in Computer Science*, pp. 665-675. Springer-Verlag, 2002.

[Atallah1999] M.J.Atallah, editor, *Algorithms and Theory of Computation Handbook*, CRC Press, Boca Raton, Florida, 1999.

[Avison1998] D.E.Avison, G.Fitzgerald, *Information Systems development: methodologies, techniques and tools*, 2.izd. McGraw-Hill, 1998.

[Baker1987] J.E. Baker, *Reducing bias and inefficiency in the selection algorithm*, u Proc. of the 2nd International Conference on Genetic Algorithms and Their Applications, Lawrence Erlbaum, Hillsdale, New Jersey, 1987.

[Baranović2003] M. Baranović, M. Borčić, D. Hunjet, V. Kalafatić, D. Kranjčec, J. Mesarić, B. Peh, *Informacijski sustav visokih učilišta*, MZT, Zagreb, 2003.

[Beck1999] K. Beck. *Extreme Programming Explained*. AddisonWesley, 1999.

[Beni1989] G. Beni, U. Wang. *Swarm intelligence in cellular robotic systems*. U NATO Advanced Workshop on Robots and Biological Systems, Il Ciocco, Tuscany, Italy, 1989.

[Bersini1996] H. Bersini, M. Dorigo, S. Langerman, G. Seront, L. Gambardella. *Results of the First International Contest on Evolutionary Optimisation,* Technical Report IRIDIA/96-18, IRIDIA, Universit'e Libre de Bruxelles, 1996.

[Bleuler2003] S. Bleuler, M. Laumanns, L. Thiele, i E. Zitzler, *PISA --- a platform and programming language independent interface for search algorithms*, u Evolutionary Multi-Criterion Optimization, EMO 2003.

[Bishop1989] J.M. Bishop, *Stochastic Searching Networks*, Proc. 1st IEE Conf. on Artifical Neural Networks, pp 329-331, London, 1989.

[Bisschop1993] J.J. Bisschop, R. Entriken, *AIMMS: The Modeling System.* Paragon Decision Technology, Haarlem, Nizozemska, 1993.

[Burkard1999] R.E.Burkard i E.Çela, *Linear assignment problems and extensions,* u Handbook of Combinatorial Optimization Vol.4 (D.-Z. Du i P.M.Pardalos, eds.), Kluwer Academic Publishers, Dordrecht, 1999.

[Burkard1994] R.E.Burkard, E.Çela, B.Klinz, *On the biquadratic assignment problem*, u Quadratic Assignment and Related Problems, P.Pardalos i H.Wolkowicz, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science 16, pp. 117-145, AMS, Providence, Rhode Island, 1994.

[Burkard1997] R.E. Burkard, S. Karisch, i F. Rendl. QAPLIB { *a quadratic assignment problem library*. J. Global Optimization, 10, pp. 391-403, 1997.

[Cahon2004] S. Cahon, N. Melab, i E. G. Talbi. ParadisEO: *A framework for the reusable design of parallel and distributed metaheuristics*. *Journal of Heuristics - Special Issue on New Advances on Parallel Meta-Heuristics for Complex Problems*, 10:357{380, 2004.

[Cela1998] E. Cela. *The Quadratic Assignment Problem: Theory and Algorithms*. Kluwer, 1998.

[Cerny1985] V. Cerny, *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm*, J. Opt. Theory Appl., Vol. 45, 1985.

[Coello2001] Coello Coello C.A. *A Short Tutorial on Evolutionary Multiobjective Optimization*. Proceedings of the 1st International Conference on Evolutionary Multi-Criterion Optimization, EMO 2001.

[Davis1985] L. Davis, *Applying Adaptive Algorithms to Epistatic Domains*, Proc. of the International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1985.

[DeJong1975] K.A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, University of Michigan, doktorska disertacija, 1975.

[Dorigo2004] M. Dorigo i T. Stützle, *Ant Colony Optimization*, MIT Press, 2004.

[Edgeworth1881] F.Y.Edgeworth, *Mathematical Physics*, P.Keagan, London, 1881.

[Eiben2003] A.E.Eiben i J.E.Smith, *Introduction to Evolutionary Computing*, Springer-Verlag, Berlin, 2003.

[Emmerich2001] M. Emmerich i R. Hosenberg. *TEA: A C++ library for the design of evolutionary algorithms*. Technical report, University Dortmund, 2001.

[Evans2004] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, Boston, 2004.

[Fertalj2006] K. Fertalj, *Folije za predavaja iz predmeta Projektiranje informacijskih sustava*, FER, 2006.

[Festa2001] P. Festa i M.G.C. Resende. *GRASP: An annotated bibliography*. U P. Hansen i C.C. Ribeiro, eds., *Essays and Surveys on Metaheuristics*, pp. 325-367. Kluwer Academic Publishers, 2001.

[Fourer1993] R. Fourer, D. M. Gay, i B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press/Wadsworth Publishing Company, Belmont, CA, 1993.

[Fowler1999] M. Fowler. *Refactoring: Improving the Design of Existing Code*. AddisonWesley, 1999.

[Gagne2002] C. Gagne i M. Parizeau. *Open BEAGLE: A new versatile C++ framework for evolutionary computations*. U *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2002*, page 888, 2002.

[Gamma2004] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Garey1979] M.R. Garey, D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H.Freeman, New York, 1979.

[Glover1989] F. Glover. *Tabu search*. Part I. Orsa J. Comput., 1989, 190-206.

[Goldberg 1991] D.E. Goldberg i K. Deb, *A Comparative Analysis of Selection Schemes Used in Genetic Algorithms*, u G.J.E. Rawlins, *Foundations of Genetic Algorithms*, pp. 69-93, San Mateo, SAD, Morgan Kaufmann Publishers, 1991.

[Goldberg1985] D.E. Goldberg i Jr.R. Lingle, *Alleles, loci and the traveling salesman problem*, Proceedings of an International Conference on Genetic Algorithms, u J.J.Grefenstette, (ed.), Lawrence Erlbaum Associates, Hillsdale 1985.

[Gutin2002] G. Gutin i A.P. Punnen. *Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, 2002.

[Holland1975] J.H.Holland, *Adaptation in Natural and Aritficial Systems*, University of Michigan Press, Ann Arbor, 1975.

[Jacobson1999] I. Jacobson, G. Booch i J. Rumbaugh, *The Unified Software Development Process*. Addison Wesley Publishing Company, 1999.

[Keijzer2001] M. Keijzer, J. J. Merelo, G. Romero i M. SchÄonauer. *Evolving objects: A general purpose evolutionary computation library*. U *Proceedings of the 5th International Conference on Evolutionary Algorithms (EA 01)*, 2001.

[Kirkpatrick1983] S.Kirkpatrick, C.D.Gelatti i M.P.Vecchi, *Optimization by simulated annealing*, Science 220, pp. 671-680, 1983.

[Koopmans1957] T.C.Koopmans i M.J.Beckmann, *Assignment problems and location of economic activities*, Econometrica 25, pp.53-76, 1957.

[Koulmas1994] C. Koulmas, S.R. Antony, i R. Jaen, *A Survey of Simulated Annealing Applications to Operations Research Problems*, Omega International Journal of Management Science Vol. 22, 41-56, 1994.

[Kurpati2002] A. Kurpati, S. Azarm i J.Wu, *Constraint handling improvements for multiobjective genetic algorithms*, Struct. Multidisc. Optim. (23), Springer-Verlag, pp. 204–213, 2002.

[Larman2005] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, NJ, second edition, 2005.

[Lawler1985] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan i D.B. Shmoys, *Sequencing and scheduling: Algorithms and complexity*, Logistics of Production and Inventory (S.C. Graves, A.H.G. Rinnooy Kan and P.H. Zipkin, eds.), North-Holland, 1993.

[Lawler1963] E.L.Lawler, *The quadratic assignment problem*, Management Science 9, pp. 586-599, 1963.

[Lin1973] S. Lin, B. W. Kernighan, *An Effective Heuristics Algorithm for the Traveling Salesman Problem*, Operations Research 21, pp. 498-516, 1973.

[Luke2002] S. Luke. *ECJ: A java-based evolutionary computation and genetic programming system*, http://www.cs.umd.edu/projects/plus/ecj/, pristup prosinac 2005.

[Lyman2003] P. Lyman, H.R. Varian, *How Much Information*, (http://www.sims.berkeley.edu/research/projects/how-much-info/index.html), 2003, pristup ožujak 2006.

[Maciaszek2001] L. Maciaszek, "*Requirements Analysis and Systems Design: Developing Information Systems with UML*", Addison-Wesley, Toronto 2001.

[MAESTRO] http://www.proteusengineering.com/maestro.htm, pristup u lipnju 2006.

[Martin2003] R.C. Martin, *Agile Software Development: Principles*, Patterns, and Practices. Prentice Hall, 2003.

[Mühlenbein1991] H. Mühlenbein, D. Schomisch i J. Born, *The Parallel Genetic Algorithm as Function Optimizer*, *Parallel Computing,* 17, pp. 619-632,1991.

[Mühlenbein1993] H.Mühlenbein, i D. Schlierkamp-Voosen, *Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization,* Evolutionary Computation, Vol. 1, pp. 25-49, 1993.

[Mayer1988] B. Mayer, *Object-Oriented Software Construction*, Hertfordshire, England, Prentice Hall, 1988.

[MZT2006] Pregled tProjekata „*Integrirani programski sustav za brodske konstrukcije*“, http://www.mzos.hr/tprojekti/prikaz_det.asp?offset=45&ID=4031, pristup lipanj 2006.

[Nevron2006] www.nevron.com, pristup veljača 2006.

[Oliver1987] I.M. Oliver, D.J. Smith, i J.R.C. Holland, *A Study of Permutation Crossover Operators on the Traveling Salesperson Problem*, Proc. of the Second International Conference on Genetic Algorithms and their Applications (pp. 224-230). Grefenstette, J.J.(ed.), Lawrence Erlbaum Associates, Hillsdale, 1987.

[Papadimitriou1982] C. Papadimitriou i K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall Inc., New York, 1982.

[Pareto1896] V. Pareto, *Cours D'Economie Politique*, Vol. I i II, F.Rogue, Lausanne, Švicarska, 1896.

[Reinelt1991] G. Reinelt: *TSPLIB-A Traveling Salesman Problem Library*, ORSA Journal on Computing, Vol. 3, pp. 376--384, 1991.

[Richter2002] J. Richter, *Applied Microsoft .NET Framework Programming*, Microsoft Press, Redmond, 2002.

[Suganthan2005] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-P. Chen, A. Auger i S. Tiwari, *Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization*. Technical Report 2005005, Nanyang Technological University, Singapore and IIT Kanpur, India, May 2005.

[Syswerda1989] G. Syswerda, *Uniform crossover in genetic algorithms*. u J.D. Schaffer, Proc. of the Third International Conference on Genetic Algorithms, San Mateo, California, SAD, Morgan Kaufmann Publishers, pp. 2-9, 1989.

[Syswerda1991] G.Syswerda, *Schedule Optimization using Genetic Algorithms*, u L. Davis (editor), Handbook of Genetic Algorithms, Van Nostrand Reinhold Library, 1991.

[Törn1989] A. Törn i A. Zilinskas, *Global Optimization* (u *Lecture Notes in Computer Science*; Vol.350), Springer-Verlag, Berlin, 1989.

[Vanjak2001] Z. Vanjak, *Metode rješavanja kvadratičnog problema pridruživanja*, magistarski rad, FER, 2001.

[Van Hentenryck1999] P. Van Hentenryck, L. Michel, P. Laborie, W. Nuijten, i J. Rogerie. *Combinatorial Optimization in OPL Studio*. In Proc. of the 9th Portuguese Conference on Articial Intelligence International Conference (EPIA'99), Evora, Portugal, September 1999.

[Veldhuizen2000] D. Van Veldhuizen and G. Lamont. *Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art*. Evolutionary Computation, 8(2):125--147, 2000.

[Vlissides1996] J. Vlissides, *Protection, Part I: The Hollywood Principle*, C++ Report, Vol. 8, No. 2, SIGS, 1996.

[Wagner2004] S. Wagner, M. Affenzeller., *HeuristicLab -- A Generic and Extensible Optimization Environment*, Proceedings of IBERAMIA 2004, Springer, New York.

[Wall1996] M. Wall. *GAlib: A C++ library of genetic algorithm components*. Technical report, MIT Mechanical Engineering Dept. (http://lancet.mit.edu/ga), 1996, pristup travanj 2006.

[Wirfs-Brock2002] R. Wirfs-Brock i A. McKean, *Object Design: Roles*, *Responsibilities, and Collaborations*, Addison Wesley, 2002.

[Zitzler2001] E. Zitzler, M. Laumanns i L. Thiele. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001.

[Whitley2000] D. Whitley, *Permutations*, u T. Bäck i dr., *Evolutionary Computation 1: Basic Algorithms and Operators,* pp. 274-284, Institute of physics Publishing, Bristol, 2000.

[Wikipedia2006a] Wikipedia, http://en.wikipedia.org/wiki/Objective_function, pristup ožujak 2006.

[Wikipedia2006b] Wikipedia, http://en.wikipedia.org/wiki/Algorithm, pristup ožujak 2006.

[Wolpert1997] D. H. Wolpert i W. G. Macready. *No free lunch theorems for optimization*. IEEE Transactions on Evolutionary Computation, 1(1):67--82, April 1997.

[Žanić2002] V. Žanić, A. Rogulj, T. Jančijev, S. Bralić, J. Hozmec, *Methodology for Evaluation of Ship Structural Safety*, Proc. of the 10th International Congress IMAM 2002*,* / Mavrakos,S.A; Spyrou,K (ur.).Crete, Greece : Hellenic Institute of Marine Technology, 2002.

# Sažetak

Rješavanje složenih optimizacijskih problema koji se javljaju u praksi zahtjeva primjenu optimizacijskih postupaka realiziranih preko računalnih algoritama. Iako postoje kvalitetni komercijalni alati za optimiranje određenim metodama, poglavito primjenjivi na rješavanje linearnih programa, sve veća potreba za rješavanjem nelinearnih i NP-teških problema zahtjeva primjenu heurističkih optimizacijskih postupaka koji, za razliku od egzaktnih metoda, ne garantiraju nalaženje globalno optimalnog rješenja, ali uglavnom u razumnom vremenu mogu generirati dovoljno dobra približna rješenja.

Međutim, postojeća programska podrška za optimiranje heurističkim postupcima ne zadovoljava. Iskorištavanje tih programskih rješenja često zahtjeva poznavanje naprednih tehnika programiranja a univerzalan im je nedostatak nepostojanje vizualnog sučelja za provođenje procesa optimiranja te nemogućnost vizualizacije rezultata i provođenja njihove analize i usporedbe, što je od velike važnosti kod heurističkih metoda optimizacije koje zahtijevaju precizno podešavanje njihovih parametara.

Na temelju provedene klasifikacije domene područja optimizacije izgrađen je konceptualni model unutar kojega su definirani i modelirani pojedini elementi iz domene i njihove interakcije. Na osnovu tog konceptualnog modela je oblikovano i izgrađeno ESOP optimizacijsko okruženje unutar kojeg je primjenom naprednih tehnika objektno-orijentirane paradigme ugrađena razvojna okosnica koja omogućava ugradnju širokog skupa različitih optimizacijskih postupaka.

Uz to je izgrađena i ESOP ljuska kao univerzalni prezentacijski sloj kroz koju se definiranje cjelokupnog procesa optimizacije obavlja preko vizualnog sučelja, i u koju su ugrađene mogućnosti za vizualizaciju i analizu rezultata i međurezultata optimizacije.

Razvijeno ESOP optimizacijsko okruženje je primijenjeno na rješavanje složenog problema optimizacije parametara mehaničke strukture brodske konstrukcije. Usprkos velikom broju varijabli i ograničenja postavljenih u problemu, primjenom genetičkih algoritama ugrađenih u ESOP okruženje su dobiveni kvalitetni rezultati.

# Ključne riječi

Optimizacija
Višekriterijska optimizacija
Optimizacijska okruženja
Objektno oblikovanje
Strukturna optimizacija

# Summary

Solving of difficult optimization problems requires the application of optimization methods implemented as computer algorithms. Although there exist a number of commercial tools for certain optimization techniques, especially for linear programming, the necessity to solve nonlinear and NP-hard problems introduces heuristic optimization methods, which, in contrast to exact methods, do not guarantee globally optimal solution, but they usually find satisfactory solutions in reasonable time.

However, the existing optimization packages based on heuristic methods are not fully satisfactory. Their use often requires knowledge of advanced programming techniques and they generally suffer from lack of graphical user interface and absence of tools for visualization of analyses and comparisons, which are very important aspects of heuristic optimization techniques, due to sensitivity upon their parameter values.

Based on classification of the problem domain, a conceptual model was developed consisting of elements from the general domain of optimization and defining their mutual interactions. Based on the conceptual model, ESOP optimization environment has been designed and implemented. ESOP framework, which serves as the foundation of ESOP environment upon which a GUI interface is built, was developed using advanced object-oriented techniques and it can support different kinds of optimization methods.

The ESOP shell was implemented as a universal presentation layer with graphical user interface, with installed components for visualization and analysis of final and intermediate results of optimization.

The ESOP environment has been used for solving a complex problem of finding optimal values for parameters in ship mechanical structure. Notwithstanding the large number of variables and constraints in the problem, the application of genetic algorithms implemented in ESOP environment has yielded good results.

# Keywords

Optimization
Multicriterial optimization
Optimization environments
Object design
Structural optimization

# Opis života

Zvonimir Vanjak rođen je 9.listopada 1974. u Sukošanu kraj Zadra. U istom mjestu je pohađao i osnovnu škola, a 1989. upisuje Matematičko-informatički obrazovni centar u Zadru. Za vrijeme pohađanja srednje škole aktivno sudjeluje na državnim natjecanjima iz fizike i informatike a u trećem razredu osvaja treće mjesto na Državnom natjecanju iz fizike. Srednju školu završava s odličnim uspjehom te biva proglašen za učenika generacije.

Fakultet elektrotehnike i računarstva upisuje 1993. godine. Na prvoj godini je nagrađen Nagradom Josipa Lončara za postignuti uspjeh a fakultet završava četiri godine kasnije s prosjekom ocjena 4.56. Nakon obrane diplomskog ispita na Fakultetu elektrotehnike i računarstva, odmah se zapošljava na Zavodu za primijenjenu matematiku, Grupa računarskih znanosti, kao znanstveni novak, a istovremeno upisuje i poslijediplomski studij na području Primijenjenih računarskih znanosti, kojeg završava u srpnju 2001. godine obranivši magistarski rad pod nazivom «Metode rješavanja kvadratičnog problema pridruživanja».

Na 554. sjednici Fakultetskog vijeća Fakulteta elektrotehnike i računarstva od 18.siječnja 2006. prihvaćena mu je tema doktorske disertacije pod naslovom „Okruženje za rješavanje optimizacijskih problema“.

Za vrijeme rada na Zavodu za primijenjenu matematiku sudjeluje u nekoliko projekata: optimizaciji rasporeda ispitnih rokova na Fakultetu elektrotehnike i računarstva, izradi Sustava za automatsku provjeru znanja i anketiranja studenata, te projektu prebacivanja Sustava za subvencioniranu prehranu studenata na .NET tehnologiju. Kao vanjski suradnik je u Institutu za nuklearnu tehnologiju sudjelovao na razvoju programske podrške za analizu podataka dobivenih ispitivanjem metodom vrtložnih struja, s posebnim naglaskom na ispitivanja cijevi parogeneratora, reaktorske posude te reaktorske kape u nuklearnim elektranama.

Na Fakultetu elektrotehnike i računarstva sudjeluje u izvođenju nastave iz predmeta Programiranje, Algoritmi i strukture podataka, Programske paradigme i jezici, Operacijska istraživanja te Objektno-orijentirano programiranje. Primarni znanstveni interes mu je istraživanje područja objektno orijentiranog oblikovanja s ciljem što efikasnije izgradnje informacijskih sustava. Objavio je nekoliko znanstvenih radova na domaćim i međunarodnim konferencijama.

# Biography

Zvonimir Vanjak was born on 9th of October 1974, in Sukošan near Zadar. He completed his secondary school education at the Mathematical and Informatical Education Center in Zadar. During his secondary schooling he was active participant in state–wide school contests in physics and informatics and in third grade he won a bronze medal in physics contest. He completed his secondary school with honors.

In 1993. he enrolled at the Faculty of Electrical Engineering and in his freshmen year he won Josip Lončar Award. He graduated in 1997. specializing in Computing and since January 1998. he has been employed at the Department of Applied Computing working as a researcher and assistant. He defended his master thesis titled „Methods for Solving Quadratic Assignment Problem“ in July of 2001.

On the 554$^{th}$ regular assembly of the Faculty Scientific and Educational Council held on 18$^{th}$ of January the proposed title of his doctoral thesis „Environment for Solving of Optimization Problems“ was approved.

During his work at the Department of Applied Computing he collaborated at several projects and as associate to Institute for nuclear technology he participated in development of software for analysis of eddy current data for use in nuclear industry.

At the Faculty of Electrical Engineering and Computing he was involved in education of courses Programming, Algorithms and Data Structures, Programming Paradigms and Languages, Operational Research and Object-oriented programming. His scientific interest is in the field of object-oriented design, with emphasis on efficient development of information systems. He is author or co-author of several scientific publications.