Latvijas Universitāte
Fizikas un matemātikas fakultāte
Datorikas nodaļa

Aspektorientētās programmēšanas risinājumi

Kursa darbs

Autors
Uldis Barbans

Vadītājs
Kārlis Čerāns
Dr. dat., asoc. prof.
LU Fizikas un matemātikas fakultāte

Rīga, 2005.

Anotācija

Kursa darbā aplūkota samērā nesen parādījusies programmēšanas paradigma – aspektorientētā programmēšana, kas tiecas papildināt pašlaik praksē plašāk pielietotās – imperatīvo, procedurālo, objektorientēto un vispārīgo programmēšanu, pretendējot uz nākamo lielo soli nozares attīstībā. Darbā ilustrēta pašreizējo paradigmu tehnisko iespēju robeža, vadot uz jauniem un skaistiem risinājumiem aspektorientētajā programmēšanā. Ņemot vērā tās vēl nenobriedušo, plūstošo stāvokli, sniegta vairāku mazsološu jaunievedumu kritika un piedāvāti vairāki jauni risinājumi. Darbs varētu piesaistīt arī ar to, ka ir viens no pirmajiem aspektorientētās programmēšanas materiāliem latviešu valodā.

Abstract

This course paper is devoted to a newly emerged programming paradigm called aspect-oriented programming, that tends to add to the most widely used paradigms such as imperative, procedural, object-oriented and generic programming, thus representing an upcoming advance in the evolution of programming techniques. In my work I illustrate the marginal capabilities of currently available techniques and direct to the transition to sound new solutions in aspect-oriented programming. Given the immature and fluent state of the art, I provide critique to the inferior inventions, and also promote some new conceptual solutions. This work might also attract attention for being one of the earliest aspect-oriented programming resources in Latvian.

Анотация

Курсовая работа посвящается относительно недавно возникшей парадигме программирования – аспектно-ориентированному программированию. Данная парадигма добавляет к обыкновенным парадигмам - императивному, процедурному, объектно-ориентированному и обобщённому программированию, тем самым претендуя на очередный большой шаг в развитие отрасли. В работе иллюстрируется пределы возможностей сегоднящих технологий, направляя к новым и красивым решениям в аспектно-ориентированном программировании. По поводу его несовершённого состояния, даётся критика неудачных нововведений и предлагается несколько новых решений. Ещё работа может привлекать внимание являясь одним из первых материалов по аспектно-ориентированному программированию на латышском языке.

Satura rādītājs

Ievads

Pastāv viedoklis, ka pienācis laiks kārtējam lielajam solim uz priekšu programmēšanas tehnoloģiju attīstības ķēdē no imperatīvās, procedurālās, objektorientētās un vispārīgās (*generic*) programmēšanas uz aspektorientēto programmēšanu.

Orientēšanās uz aspektiem būtībā nozīmē vēlēšanos brīvi rīkoties ar dažādu prasību realizācijām programmatūrā, lai šīs realizācijas varētu efektīvi atkalizmantot, kombinējot lietojumprogrammās. Tam vispirms, šķiet, nepieciešams apzināt iespējas prasības atdalīt, realizēt tās katru atsevišķi. Uzdevums, piemēram, atsevišķi realizēt "pilnekrāna režīmu" – neizvirzot pieņēmumus ne par attēlojamā objekta raksturu, ne par citiem līdztekus iedomājamiem režīmiem un procesiem...

Tā kā aspektorientētā programmēšana ir samērā jauns virziens, tās teorijā un praktiskajos risinājumos vēl ir daudz, ko atklāt, daudz, ko izdarīt pareizi un ko izdarīt nepareizi.

Aspektorientācija nemērķē uz šauri specifiskām nozarēm, tai paredzama vieta starp programmēšanas pamatlīdzekļiem un plašs pielietojums visdažādāko informācijas sistēmu izstrādē. Arī tas piesaista interesi un ir mēģināts uzsvērt darba gaitā.

Šajā darbā:

1. nodaļā esmu analizējis tradicionālo programmēšanas paņēmienu efektivitātes robežas, praksē sastaptās problēmas,
2. nodaļa apskatīti daži progresīvi, varbūt pārsteidzoši risinājumi parastajās paradigmās, kas noderēs, lai ilustrētu pāreju uz aspektorientētās programmēšanas idejām,
3. nodaļa sniedz aspektorientētās programmēšanas definīciju,
4. nodaļa ir pastāvošo aspektorientētās programmēšanas praktisko realizāciju apskats,
5. nodaļa ir pētījums par šo realizāciju piedāvātajiem jaunajiem risinājumiem, kas nav iespējami tradicionālajās programmēšanas paradigmās,
6. nodaļā analizēju izpētītos risinājumus, norādot uz veiksmīgajām iestrādnēm un nepilnībām,
7. nodaļa satur manis paša idejas par iespējamiem jauniem risinājumiem un pētījumu virzieniem,
8. nodaļa vispārīgi ieskicē aplūkotās paradigmas nākotnes izredzes, to, kāda nozīme varētu būt aspektorientētajai programmēšanai, tās vietu datorzinātnēs,
9. nodaļa apraksta ar aplūkoto tēmu saistītas blakusnozares un interesantākos citu autoru darbus, ar kuriem nācies saskarties.

Ierosinājumu mērķis un darba kopējā ievirze ir sniegt lasītājam aptverošu un sakarīgu pārskatu par aspektorientētās programmēšanas iespējamajiem pielietojuma virzieniem, kā arī iekrāt idejas, kas varētu būt noderīgas jaunas, attīstītākas aspektorientētās programmēšanas sistēmas izveidošanai.

Raudzījos uz jau piedāvātajiem risinājumiem samērā kritiski, ņemot vērā pats savas intereses jaunās tehnoloģijas pielietošanā.

Iztirzājums

1. Pastāvošās problēmas

Lai ilustrētu pāreju uz jauno programmēšanas paradigmu, minēšu gan tradicionālajās paradigmās atrisināmas, gan pagrūti atrisināmas, gan neatrisināmas problēmas.

1.1. Koda izkaisītība

Bieži programmatūras koda fragmentā saskaras dažādu veicamo funkciju intereses:

```
MyWidget::MyWidget( QWidget* parent, const char* name )
    : QMainWindow( parent, name )
{
    ...

    QPopupMenu* file = new QPopupMenu(this);
    file->insertItem( tr("E&xit"), qApp, SLOT(quit()),
          QAccel::stringToKey(tr("Ctrl+Q")) );
    menuBar()->insertItem( tr("&File"), file );

    ...
}
```

1. piemērs – grafiskās lietotāja saskarnes elementa implementācija, izmantojot daudzplatformu bibliotēku Qt

`tr()` rūpējas par daudzvalodu saskarnes nodrošināšanu, iejaukta arī īsinājumtaustiņa nodrošināšana.

Iekaisītais kods padara neuzskatāmu pamatproblēmas risinājumu, un tā prasība, kurai veltīts izkaisītais kods, bieži vien rezultātā vispār tiek izpildīta nesistemātiski, jo kopīgs skatījums uz tās realizāciju nav iespējams.

1.2. Koda dublēšanās

Koda fragmenta nokopēšana un ielīmēšana citā vietā, lai ātri realizētu līdzīgu risinājumu un lieki neapgrūtinātu sevi ar abstrakcijas domāšanu, lieliski veicina darba produktivitāti koda rindu izteiksmē, taču sagādā pamatīgas galvassāpes, ja vēlāk šis koda fragments jālabo visās vietās, kur tas nokopēts.

```
    FILE *f;
    char *p;

    f = fopen("piedalies.txt", "w");
    if (f == NULL) {
          return -1;
    }

    p = get_page("http://www.piedalies.lv/");
    if (p == NULL) {
          fclose(f);
          return -2;
```

```
    }

    if (!strstr(p, "Orientēšanās")) {
          free(p);
          fclose(f);
          return -3;
    }

    ...

    free(p);
    fclose(f);
    return 0;
```

2. piemērs – resursu atbrīvošanas koda dublēšanās valodā C

Aizmirst kādā no daudzajām vietām aizvērt failu vai atbrīvot atmiņu ir pārāk viegli.

Humoristiski parādība plaši apspēlēta [UC], kā viena no galvenajām nekvalitatīva programmējuma pazīmēm nopelta [PP, 22].

1.3. Sapiņķētība (neortogonalitāte)

[PP, 29] min eksakti domājošiem cilvēkiem piemēroti tehnisku analoģiju ar helikoptera vadīšanu – proti, lai nosēdinātu helikopteru, nebūt nepietiek pakāpeniski samazināt rotora apgriezienus – izjūk griezes momentu līdzsvars ar stūres rotoru, tas sāk griezt helikopteru vērpetē, piedevām nemainīgs galvenā rotora lāpstiņu leņķis gādā par bīstamu sānsveri... Pareizā procedūra ir, nospiežot kreiso sviru, vienlaikus atbrīvot labo sviru un nospiest labo pedāli.

Ja viena programmas moduļa izmainīšana prasa arī daudzu citu izmainīšanu, ja nav iespējams vienkārši notestēt moduli atsevišķi, bet nepieciešama pārējo daļu līdzdarbība, ja operācijas ietekmē viena otru ar negaidītiem blakusefektiem, tās ir praktiskās koda sapiņķētības izpausmes.

1.4. Par daudz uz priekšu plānošanas

Vēl viens aspekts, kas apgrūtina programmēšanas uzdevuma realizāciju, ir prasības nodrošināt koda uzturamību un paplašināmību nākotnē. Valodā C tas parasti nozīmē neglītu tehnisku makro apzīmējumu parādīšanos, raizes par dažādām skaitļu un simbolu reprezentācijām dažādos datoros u.c.

```
PHP_FUNCTION(set_time_limit)
{
    zval **new_timeout;

    if (PG(safe_mode)) {
          php_error_docref(NULL TSRMLS_CC, E_WARNING, "Cannot set
time limit in safe mode");
          RETURN_FALSE;
    }

    ...
}
```

3. piemērs – PHP moduļa izejas koda fragments valodā C ar makro apzīmējumiem paplašināmības nodrošināšanai

Savādā kārtā ar visiem šiem pasākumiem parasti ilgai nākotnei nepietiek – daudzi paziņojumi par kāda programmprodukta nākamās versijas iznākšanu sākas ar vārdiem "pārrakstīts no sākuma" (*"rewritten from scratch"*), kas jāuztver kā kvalitatīvas izaugsmes pazīme. Līdzīgi dinozauru izmiršanai, vai ne? Saukt šo procesu par paaudžu maiņu būtu nepareizi (spilgts novērojums no [IP]).

1.5. Abstrakcijas apgrūtinājumi

Abstrakcija pati par sevi tuvina izstrādes procesu reālajam pielietojumam. Programmēšanas valodas kļūst aizvien līdzīgākas cilvēka valodai, atvieglojot programmēšanas darbu.

Pastāv priekšstats, ka augstāka līmeņa programmēšanas valodās radītās programmas neizbēgami zaudē zemāka līmeņa valodās rakstītajām veiktspējas ziņā. Priekšstats praksē reizēm apstiprinās:

	P4 2680 Windows 98 MS VC++ 6.0 Release	P4 1680 Red Hat Linux 7.1 g++ 2.96 -O2
1000000 int skaitļu sakārtošana		
qsort() no C standarta bibliotēkas	0.38	0.68
sort() no C++ STL bibliotēkas	0.15	0.20
1000000 int skaitļu nolasīšana no faila		
fscanf() no C standarta bibliotēkas	0.30	0.80
ifstream no C++ STL bibliotēkas	2.27	0.98
1000000 int skaitļu ierakstīšana failā		
fprintf() no C standarta bibliotēkas	1.03	0.52
ofstream no C++ STL bibliotēkas	1.48	0.44

4. piemērs – C un C++ standarta bibliotēku veiktspējas eksperimentāls salīdzinājums (sekundēs)

Kā tas saskan ar programmēšanas valodu radītāju apgalvojumiem, ka "jūs nemaksājat par to, ko neizmantojat" un "valoda pieļauj teju optimālu augsta līmeņa kodu, vienīgi kompilatori vēl nav pietiekami attīstījušies"? ([BS])

2. Pastāvošie risinājumi

2.1. Pareiza strukturēšana

1.1 apskatītajai problēmai bieži ir vienkāršs risinājums – strukturēt kodu, pakārtojot darbības vadoties pēc resursu izmantojuma.

```
FILE *f;
char *p;
int ret = 0;

f = fopen("piedalies.txt", "w");
if (f != NULL) {
      p = get_page("http://www.piedalies.lv/");
      if (p != NULL) {
            if (strstr(p, "Orientēšanās")) {

                  ...

            } else {
                  ret = -3;
            }
            free(p);
      } else {
            ret = -2;
      }
      fclose(f);
} else {
      ret = -1;
}

return ret;
```

5. piemērs – C programmas strukturējums, kas ļauj izvairīties no dublētas resursu atbrīvošanas

AOP materiālos šādu viena apsvēruma noteiktu koda kārtību dēvē par "dominējošās dekompozīcijas tirániju".

Ko darīt, ja izkaisītais kods uzbrūk plašākā frontē – piemēram, nepieciešams vairāku funkciju ieejā pārbaudīt padotā parametra vērtību? Vai nu jāatkārto šis pārbaudes kods katras funkcijas sākumā, vai jāpārveido funkcionālais modelis.

2.2. Objektorientācija, mantošana, polimorfisms

Objektorientācija jau lielākā mērā atbrīvo programmētāju no tehniska, kļūdu nedroša darba.

Piemēram – destruktora jēdziens. Skaidrojot aspektorientētā terminoloģijā - resursu atbrīvošanas aspekts tiek realizēts atsevišķi, kompilatorā, kurš to nekļūdīgi iesaista programmas kodā.

Tāpat – mantošanas iespēja atdala no programmas rūpes par datu tipu paplašināmību (sal. 1.4.) – arī aspekts.

2.3. Daudzkāršā mantošana

Daudzkāršā mantošana (*multiple inheritance*) nodrošina ievērojamu brīvību manipulācijās ar ne tikai tehniskiem, bet arī funkcionalitātes aspektiem.

Varētu uzreiz vēlēties, piemēram, "izklaides centra" aplikācijā tādā veidā mehāniski apvienot "televizoru", "interneta pārlūku" un "mūzikas atskaņotāju", bet

diez vai izdosies tikpat mehāniski pievienot "satura piekļuves ierobežojumus bērniem" vai "balss vadības atbalstu". Tie ir aspekti, kas, šķiet, jāņem vērā jau katras sastāvdaļas realizācijā, līdz ar to tie tiek izkaisīti un nav pārskatāmi.

2.4. Ko devusi vispārīgā programmēšana

Viens no galvenajiem vispārīgās programmēšanas devumiem ir datu struktūru programmēšanas vienkāršošana, atbrīvojot no to vairākkārtīgas realizācijas dažādiem datu tipiem vai atkārtojošamies datu tipu pārveidojumiem vienas abstraktas realizācijas gadījumā. Datu strukturēšanas aspekts tiek realizēts patstāvīgāk.

Vispārīgās programmēšanas iespējas aspektorientācijas nodrošināšanā uzskatāmi demonstrē [URL 1].

2.5. Abstrakciju pielabināšana metaprogrammējot

Funkcionālajai programmēšanai raksturīgas fantastiskas lietas, tiecoties pēc nesamazinātas veiktspējas augstā abstrakcijas līmenī, cilvēki ([URL 2]) iemācījušies veikt arī ar C++ šablonu mehānismu. Iespējams, piemēram, panākt, lai vektoru saskaitīšanas kods no 6. piemēra kompilatora interpretācijā nelīdzinātos 7. piemērā attēlotajam, bet gan izpildītos vienā ciklā bez pagaidu objektu ieviešanas.

```
Vector<double> a, b, c, d;
a = b + c + d;
```

6. piemērs – augsta abstrakcijas līmeņa kods C++

```
double* _t1 = new double[N];
for (int i=0; i < N; ++i)
      _t1[i] = b[i] + c[i];
double* _t2 = new double[N];
for (int i=0; i < N; ++i)
      _t2[i] = _t1[i] + d[i];
for (int i=0; i < N; ++i)
      a[i] = _t2[i];
delete [] _t2;
delete [] _t1;
```

7. piemērs – triviālā pieeja dod neefektīvu rezultātu

```
for (int i=0; i < N; ++i)
      a[i] = b[i] + c[i] + d[i];
```

8. piemērs – vēlamais rezultāts iegūts ar metatransformācijas pielietošanu abstraktajai saskaitīšanas izteiksmei

Skatot pēc būtības, šis risinājums kā reiz ir "kompilatora izglītošana" (skat. izteikumus 1.5.), optimizācijas procedūras (aspekta) iemānīšana ārējā bibliotēkā. Šādi veiktspējas saglabājums tiek iegūts uz izpildkoda izmēra pieauguma rēķina, bet tā jau ir pazīstama dilemma tipiskajos optimizēšanas paņēmienos. Vai tas izklausās pēc ierobežojumiem attiecībā pret netipiskiem paņēmieniem? Nē taču, metaprogrammēšana pieļauj ne jau tikai atgriešanos pie iedomāta izvērstākā koda, bet arī tā pārstrukturēšanu vēl kādā inteliģentā veidā. Šis risinājums jau ļoti vilina uz [IP] aprakstīto enzīmu principu, bet trūkums ir sagatavju metaprogrammu apgrūtinātā lasāmība un no tā izrietošā neuzturamība.

```
// This class encapsulates the "+" operation.
struct plus {
public:
    static double apply(double a, double b) {
        return a+b;
    };
};

// The parse tree node
template<typename Left, typename Op, typename Right>
struct X {
    Left leftNode_;
    Right rightNode_;

    X(Left t1, Right t2)
        : leftNode_(t1), rightNode_(t2)
    { }

    double operator[](int i)
    { return Op::apply(leftNode_[i],rightNode_[i]); }
};

// A simple array class
struct Array {
    Array(double* data, int N)
        : data_(data), N_(N)
    { }

    // Assign an expression to the array
    template<typename Left, typename Op, typename Right>
    void operator=(X<Left,Op,Right> expression)
    {
        for (int i=0; i < N_; ++i)
            data_[i] = expression[i];
    }

    double operator[](int i)
    { return data_[i]; }

    double* data_;
    int N_;
};
```

9. piemērs – transformāciju izpildošā sagatavju metaprogramma

3. Definīcija

Ar aspektorientāciju saistītos jēdzienus definēšu, balstoties uz pazīstamiem programmatūras izstrādes jēdzieniem un ievadā minētajām praksē līdz šim izplatītākajām programmēšanas paradigmām.

3.1. Aspekts

Par **aspektu** sauc jebkuru identificējamu funkcionalitāti vai īpašību, ko realizē programmatūra.

Šis termins idejiski tuvinās programmatūras prasībām, kā uz tām raugās lietotājs, ne tikai tehniskajai realizācijai.

Koda tipiskie pamatelementi, datu struktūras un procedūras, bieži vienlaikus rūpējas par vairākām prasībām – aspektiem. Piemērā 1.1. nodaļā tās ir daudzvalodība un klaviatūras pieejamība.

Ja turklāt šīs prasības konsekventi jārealizē izkaisītas vairākās koda vietās, tad tās sauc par **šķeļošamies aspektiem** (*crosscutting concerns*). Piemēram, kursa darba dokumentālās reprezentācijas struktūras elements – literatūras saraksts – papildus funkcijai sniegt atsauci uz izmantotajiem materiāliem pakļauts arī vairākām šķeļošamies prasībām – tam jābūt latviešu valodā (kā arī pārējām darba daļām), jābūt sakārtotam, jābūt vizuāli pēc priekšrakstiem noformētam un jāparādās satura rādītājā (arī attiecas uz visām darba daļām).

3.2. Aspektu atdalīšana

Aspektu atdalīšana (*SOC – separation of concerns*, bieži arī – modularizēšana) ir galvenais šajā darbā analizētais programmatūras projektējuma principu attīstības virziens – pietuvināšanās lietotāja skatījumam uz programmatūru kā atsevišķu funkciju un īpašību apvienojumu, attālinoties no tehniskās realizācijas ierobežojumu iespaidota domāšanas veida.

Sevišķi ievērības cienīga ir **daudzdimensionālā** – šķeļošos aspektu atdalīšana (*multi-dimensional SOC*, *MDSOC*), jo to, kā parādīts iepriekšējā nodaļā, līdzšinējās programmēšanas paradigmas nerisina (skat. 2.1., 2.3.).

3.3. Aspektorientētā programmēšana

Aspektorientētā programmēšana (turpmāk – AOP) ir programmēšanas tehnoloģiju kopums, kas tiecas tuvināt programmas kodu tās prasību projektējumam, piedāvājot līdzekļus aspektu, tajā skaitā šķeļošos, atdalīšanai kodā.

AOP kā paradigma neizslēdz citu pašlaik populāro paradigmu lietošanu. Līdzīgi kā objektorientētā programmēšana pastāv līdzās procedurālajai un vispārīgā gluži ortogonāli papildina gan iepriekšminētās, gan, piemēram, funkcionālo programmēšanu, AOP arī iesaistās kā ortogonāls papildinājums.

3.4. Piesaistes punkts, piesaistes apraksts, piesaistāmais kods

Aspektu realizāciju kombinēšana izpildāmajā programmā nevar notikt uz minēšanas pamata.

Par **piesaistes punktu** (*join point*) sauc katru aspekta koda reprezentācijas vai tās semantiskā modeļa elementu, kas var tikt mehāniski identificēts, lai realizētu aspektu kombinēšanu.

Piesaistes apraksts (*pointcut*) ir AOP līdzeklis, kas ļauj izvēlēties piesaistes punktu apakškopu, kuriem piekārtot **piesaistāmo kodu** (*advice*), tā realizējot šķeļošos aspektu modulāru realizēšanu.

3.5. Saistīšana

Par **saistīšanu** (*weaving*) sauc procesu, kas kombinē atsevišķu aspektu realizācijas rezultējošajā programmā, identificējot piesaistes punktus un realizējot to mijiedarbību ar piesaistāmo kodu saskaņā ar piesaistes aprakstu.

4. Vēsturisks pārskats

Šajā nodaļā sniegts īss atskats AOP ne pārāk garajā vēsturē.

Jau pirms AOP risinājumu teorētiskā izklāsta palūkosimies arī, kā tie izskatās darbībā, ņemot par piemēriem populārākās realizācijas.

4.1. Pirmsākumi

Koda dublēšanās un komponentu sapiņķētības kritērijus objektorientācijas kontekstā ierosina Karl J. Lieberherr un Ian Holland [AP] 1989. gadā – tā saucamo Dēmetras likumu (*law of Demeter*). Uz tiem orientējas tā saucamā adaptīvā programmēšana (*adaptive programming*), piedāvājot programmu transformācijas metodes modularizācijas veicināšanai. Pazīstamas izstrādes sistēmas Demeter/C++ un Demeter/Java.

AOP jēdzienu ievieš Gregor Kiczales u.c. [AOP 1] 1997. gadā, ierosinot modularizēt ne tikai objektu mijiedarbību, bet arī tos caurvijošos šķeļošos aspektus, izmantojot ne tikai programmu transformācijas, bet arī piesaisti to semantikai.

4.2. AspectJ

Par pirmo AOP risinājumu praksē uzskata 2001. gadā klajā nākušo AOP valodu **AspectJ** ([URL 3]), kas izstrādāta Xerox PARC, paplašinot Java.

Ar plašu izstrādes rīku klāstu un pierastās Eclipse izstrādes vides atbalstu šī ir populārākā un iespējām bagātākā AOP realizācija šobrīd, bet neapšaubāmi ne vienīgā.

4.3. AspectWerkz

AspectWerkz ([URL 4]) izstrādātāji izvēlējušies neveidot jaunu programmēšanas valodu, bet realizēt visu, ko iespējams, tīri valodas Java ietvaros. Aspekti tiek definēti parastu Java klašu veidā. Papildu izteiksmes līdzekļi, protams, ir nepieciešami, šajā gadījumā papildinformācija tiek rakstīta Java komentāros, atsevišķā XML failā vai izmantojot Java 1.5 jauno iespēju – metapiezīmes. Eksperimentālai tehnoloģijai tas ir atzīstams risinājums – saduroties ar problēmu, vismaz Java izejas kods ir lietojams arī standarta Java kompilatorā.

2005. gada janvārī AspectWerkz un AspectJ autori paziņoja par pētījumu un iestrādņu apvienošanu. AspectJ 5 izlaidumā sola apvienot labāko no abu darba grupu rezultātiem.

4.4. JBoss AOP

Arī **JBoss AOP** ([URL 5]), līdzīgi AspectWerkz, neievieš paplašinājumus Java valodā, bet izmanto metapiezīmes vai atsevišķus XML failus aspektu sasaistīšanai.

Interesanta JBoss AOP iespēja ir aspektu piesaistīšana dinamiski Java programmas izpildes laikā. Tādas iespējas paveras, pastāvot labai modularizācijai.

4.5. Hyper/J

IBM savu izpētes virzienu dēvē par daudzdimensiju programmēšanu (*multi-dimensional programming*).

Hyper/J ([URL 6]), atšķirībā no iepriekš minētajām realizācijām, neizšķir kādu galveno kodu un tam pievienojamus aspektus. Hyper/J visi aspekti ir vienlīdzīgas "hiperšķēles", kuras saliekot kopā, iegūst izpildāmo programmu. Šai pieejai noteiktās situācijās ir priekšrocības.

4.6. AspectC++

Ievērojami grūtāk vēlēties ātru AOP ienākšanu zemāka līmeņa programmēšanas valodās, kā C un C++. Tomēr šī kustība arī jau ir aizsākusies.

AspectC++ ([URL 7]) nav realizētas visas populārākās izpildes laikā kontrolējamās aspektu piesaistīšanas iespējas, toties C++ izvēle par bāzes valodu dod iespēju kombinēt AOP ar vispārīgās programmēšanas risinājumiem.

4.7. AOSD konference

Kopš 2002. gada ik gadu notiek Aspektorientētās programmatūras izstrādātāju konference (AOSD, [URL 8]), kuras materiālus var izmantot kā izejas punktu papildus informācijas meklējumos. Ka centralizēta domu apmaiņa ir svarīgs virzītājspēks jauniem risinājumiem programmatūrā, jau pierādījusi pazīstamā Objektorientētās programmēšanas sistēmu, valodu un lietojumu konference (OOPSLA, [URL 9]).

5. AOP risinājumi

Pievērsīsimies šobrīd pieejamajiem AOP risinājumiem teorētiski, vairāk vai mazāk neatkarīgi no realizāciju atšķirībām.

5.1. Koda papildināšana piesaistes punktos

Šķiet, vispopulārākais AOP risinājums koda izkaisītībai, to drudžaini apgūst arī tāda valoda kā PHP (http://www.aophp.net/).

Izvēlētiem piesaistes punktiem, piemēram, funkcijas izsaukumam (`call`), objekta konstruēšanai (`constructor`) u.c. (skat. [URL 10] dažādu rīku atbalstīto piesaistes punktu salīdzinājumu), izmantojot piesaistes aprakstu valodu, piekārto papildus izpildāmo kodu. Konkrētā piesaistes punkta informācija piesaistītajam kodam ir pieejama caur piesaistes punkta API – var nolasīt funkcijai padotos parametrus u.tml.

Šādi var ērti programmai pievienot un atvienot, piemēram, interaktīvu lietotāja atbalstu, trasēšanas un žurnalēšanas funkcijas un citas palīgfunkcijas, kas neprasa iejaukšanos pamatkoda darbībā.

```
// bankas lietojumprogrammas klase
public class Account {
     ...

     public void credit(float amount)  {
           ...
     }

     public void debit(float amount)  {
           ...
     }

     public float getBalance() {
           ...
     }

     ...
}
```

10. piemērs – klase bez izkaisīta palīgfunkciju koda

```
// transakciju žurnalēšanas aspekts
public aspect TransactionLogging {
     // piesaistes apraksts
     public pointcut accountActivities() :
      call(void Account.credit(..)) ||
      call(void Account.debit(..));

     // piesaistāmais kods
     before() : accountActivities {
           // piesaistes punkta API pielietojums
           Signature sig = thisJoinPointStaticPart.getSignature();
           System.out.println("[" + sig.getName() + "]");
     }
}
```

11. piemērs – atsevišķs žurnalēšanas aspekts valodā AspectJ

5.2. Koda modificēšana piesaistes punktos

Lielākais vairums iedomājamo noderīgo aspektu, šķiet, tomēr prasa iejaukšanos pamatkoda darbībā. Piemēram, lai pievienotu datu šifrēšanas aspektu kādai Interneta saziņas lietojumprogrammai, tās pamatkoda plūsma jāpapildina ar pārraidāmo datu izmainīšanu.

Primārais AOP risinājums, kas to nodrošina, ir piesaiste procedūru izsakumiem programmā, ko apraksta ar atslēgas vārdu `around`. Piesaistes punkta API tad piedāvā iespējas operēt ar parametru vērtībām, veikt izsaukuma vietā citu darbību vai turpināt izsaukumu ar `proceed()` metodi.

[URL 11] – piemērs, kā, veiksmīgi kombinējot vispārīgās programmēšanas un AOP līdzekļus AspectC++, realizēt vispārīgu (tipu neatkarīgu) un atkalizmantojamu kešošanas aspektu.

Vienkāršāks piemērs valodā AspectJ:

```
public aspect ExtraParens {
    String around() :
     execution(String node.toString()) {
          String normal = proceed();
          return “(” + normal + “)”;
    }
}
```

12. piemērs – metodes atgriezto simbolu virkni papildina ar iekavām

5.3. Klašu hierarhijas paplašināšana

AOP attīsta jaunā kvalitātē arī 2.3. nodaļā minēto daudzkāršo mantošanu. Esošā klašu hierarhijā iespējams ieviest starpposmus, izvēlēties vecākus, neizmainot oriģinālo izejas kodu vai to pat vispār neizmantojot.

Interesants blakusefekts šim jaunievedumam ir iegūta daudzkāršā mantošana valodā Java.

5.4. Attīstītākie risinājumi

Pašlaik attīstītākie AOP risinājumi piedāvā aprakstīt piesaisti ar izpildes laikā pārbaudāmiem ne pārāk sarežģītiem nosacījumiem attiecībā pret izpildes laika izsaukumu vēsturi (*call stack*). Atslēgas vārds `cflow` ļauj izvēlēties tikai tos piesaistes punktus, kurus sasniedz noteiktas funkcijas izsaukuma ietvaros. Tipisks pielietojums ir izvairīšanās no rekursīvas piesaistītā koda izsaukšanas.

5.5. Kombinētie risinājumi

Kombinējot AOP ar metaprogrammēšanas iespējām, [URL 12] piedāvā elegantāku aspektu piesaistīšanas veidu: izmantojot metapiezīmes (*annotations*), piesaistes punkta definīcija no piemēra 5.1. nodaļā iegūst 14. piemērā parādīto veidu, kas ir kas ir vieglāk atkalizmantojams risinājums.

```
public class Account {
    ...

    @Transactional
    public void credit(float amount)  {
```

```
        ...
    }

    @Transactional
    public void debit(float amount)  {
        ...
    }

    public float getBalance() {
        ...
    }

    ...
}
```

13. piemērs – metapiezīmes

```
    public pointcut accountActivities() :
     call(@Transactional void Account.*(..));
```

14. piemērs – piesaistes apraksts, kas izmanto metapiezīmes

5.6. Ieguvumi faktos

[AOP 1, 6, 13] veiktais izpētes darbs apliecina, ka esošos līdzekļus efektīvi izsmēlušam, no 768 līdz 35213 koda rindām nobriedušam projektam AOP risinājumi var piešķirt gluži vai otro elpu – modularizējot veiktās optimizācijas, koda nepieciešamais apjoms, izrādās, ir vien 1039 rindas. Un kāpēc gan maksāt vairāk?

6. Risinājumu analīze

6.1. Nepilnība

Pirmais iespaids, kas rodas, salīdzinot dažādās realizācijās atšķirīgo piesaistes punktu atbalstu – vai nebūtu iespējama universālāka piesaistes punktu valoda, kas ļautu paplašināt programmu patvaļīgos punktos?

6.2. Sapiņķētība tomēr pāri visam

Kā redzējām piemēros 5. nodaļā, esošie AOP risinājumi ļauj atsaistīt rokas programmētājiem, kuriem līdz šim kaut kas kaut kādu iemeslu dēļ bijis piespiedu kārtā jāraksta noteiktās koda vietās, tādējādi sarežģījot koda struktūru un novēršot uzmanību no veicamā pamatuzdevuma. Tā vietā AOP piedāvā piesaisti procedūras nosaukumam ar dažādiem uz teksta apstrādi orientētiem paņēmieniem (regulāro izteiksmju variācijām). Kaut šo paņēmienu attīstīšana AOP kontekstā ir nozīmīga, šajā darbā to uzskatīšu par paralēlu programmēšanas nozari. Programmas, kuras kontrolē savu struktūru, it sevišķi - apstrādā pašas savu tekstu, ir metaprogrammēšanas izpētes objekts (skat. 9.1.). Gribas izpētīt, cik efektīvi un atvērti risinājumi iespējami AOP tituljomā – programmprodukta dažādo veicamo funkciju (aspektu) modularizēšanā un operācijās ar tiem.

Nez, cik tas tīri izdosies, bet paļaušanos uz to, ka visās vēlamajās aspekta pielietošanas vietās būs programmētāja pašrocīgi piestiprināti karodziņi (kas praksē gan ir efektīvi, skat. 5.5.), uzskatīsim par netīru pieeju.

6.3. Izkaisītība nebūt nebūtu ļaunākais

[URL 13] min analoģiju ar meža neredzēšanu aiz kokiem – projektā izkaisītas koda rindas traucē redzēt attiecīgā aspekta kopainu. Tieši to lielā mērā risina aspektorientētā modularizācija – ļauj nodalīt, apkopot visu programmas koda mežu, kas attiecas uz vienu no daudzajiem programmas pienākumiem, atsevišķā modulī, piedāvājot rīkus koku iegūšanai gatavajā produkcijā. Šajā sakarā gribētos piezīmēt, ka diez vai teksta failu fiziskai atdalīšanai šajā efektā ir galvenā loma – kāpēc gan neglabāt kodu koku veidā un neizmantot rīkus mežu apskatīšanai? Uzsvars jāliek uz "izmantot rīkus"!

6.4. Ļaunā nelasāmība

Modularizētais kods no 5.1. gan ir mazs apjomā, bet jāpiedomā, vai neož pēc "pārāk mazs". Vārdi "`before`" un "`after`" ierauj sevī to informāciju, ko imperatīvajā programmēšanas paradigmā līdz šim tik ļoti ierasts izteikt tieši ar uzskatāmu koda izkārtojumu "pirms" un "pēc". Vai tas ir vēlams? Ja šiem apzīmējumiem grasās pievienoties vēl daudzi citi, vai uzskatāmība netiek zaudēta pavisam? Ja pievienoties negrasās, tad – ar ko gan šie pāris ir tik īpaši, ka vajag tādus ieviest?

6.5. Apguve

Viens no populārākajiem ar cilvēka faktoru saistītajiem kritērijiem izpētītajos AOP risinājumu apskatos ir vieglāka vai grūtāka apguve (*learning curve*). Bet varbūt nebaidīsimies un nodalīsim šo aspektu darba organizācijas līmenī? Derētu panākt, ka par sarežģīto un nepārskatāmo aspektu mijiedarbības programmēšanu, ja tā tāda ir, atbild augstāka līmeņa speciālisti vieni, savukārt lielākā daļa projektā iesaistīto programmētāju darbojas pie atsevišķiem moduļiem ar ierastām tehnoloģijām un neizjūt tādas apguves grūtības, kas traucētu izjust raitas AOP izplatības ieguvumus. Nedaudz svaigu ideju arī šajā virzienā sekos 7. nodaļā.

7. Ierosmes jauniem risinājumiem

7.1. Cik tad var domāt uz priekšu

Cik rindiņu vajadzētu aizņemt programmai, kas izdrukā "Hello, World!"? Domāju, ka vienu.

```
PRINT "Hello, World!"
```

15. piemērs – "Hello, World!" programma valodā QuickBASIC

Kāpēc gan "nopietno" programmēšanas valodu veidotāji iedomājas, ka to lietotājiem pastāvīgi patīk interesēties par to, kāda vērtība programmai jāatgriež operētājsistēmai, lai tā būtu apmierināta, vai par to, ka tā kaut kā jānosauc, pat ja vēl nosaukums nav izdomāts?

```
#include <stdio.h>

int main()
{
    printf("Hello, World!");
    return 0;
}
```

16. piemērs – "Hello, World!" programma valodā C

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SmallestProgram.
PROCEDURE DIVISION.
DisplayGreeting.
    DISPLAY "Hello, World!".
    STOP RUN.
```

17. piemērs – "Hello, World!" programma valodā COBOL

Efektīva modularizācija šajā ziņā būtu tāda, kas ļautu vispār neuztraukties par kādu valodas papildiespēju, tehnisku blakusparādību, neinteresējošu aspektu, tādējādi ļaujot darbā koncentrēties uz veicamo uzdevumu.

Valodā PHP pazīstama "*fopen wrappers*" iespēja ([URL 14]) – izpausmēs ne mazāk aspektorientēts risinājums kā piemēros 5. nodaļā minētie, kas tikai nav tādā vārdā nosaukts. Un tiešām, lai nolasītu failu, kurš atrodas uz tīmekļa servera Lietuvā, nepieciešams ne vairāk kā tiešām nepieciešams – tā pati `file_get_contents()` funkcija un cilvēkam, kurš mācējis izteikt vēlēšanos nolasīt "failu, kurš atrodas uz tīmekļa servera Lietuvā", neizbēgami saprotamais faila nosaukuma pieraksta veids – URL.

```
$file = file_get_contents('http://www.draugams.lt/');
```

18. piemērs – lakoniska tīkla abstrakcija valodā PHP

Nav jāuztraucas par nekādiem sarežģītiem datu pārraides protokoliem, tīkla adaptera izcelsmes valsti un tamlīdzīgiem neiederīgiem aspektiem.

Šī iespēja arī realizēta pietiekami modulāri, lai būtu izslēdzama un ieslēdzama ar vienu rindiņu konfigurācijas failā. Kā tieši realizēta – ar brīvprātīgu darbu un tālredzīgu projektējumu, ko atvieglināt vajadzētu AOP.

7.2. Apguves modularizēšana

Tieši iepriekšējā nodaļā (7.1.) minētie risinājumi liek visvairāk aizdomāties par agrāk (6.5.) apspriestās apguves problēmas minimizēšanas iespējām. AOP vajadzētu būt tai paradigmai, kura ļautu jaunas tehnoloģijas bibliotēku veidā modulāri piesaistīt programmēšanas videi bez nepieciešamības apmācīt tos, kuri neinteresējas par piedāvātajām papildiespējām, vien vēlas, lai tas, ko mācēts paveikt ar iepriekšējām metodēm, bez izmaiņām strādātu arī jaunajā vidē – lai pieredze neietu zudumā (skat. 1.4.).

7.3. Izstrādes vides modularizēšana

Vēl vairāk – interaktīvajai programmēšanas videi kopā ar iepriekšminēto papildiespēju izmantošanu "pēc vajadzības" varētu piešķirt arī projekta vadītāju izstrādātu programmēšanas ieteikumu, vienota stila ieturēšanas kontrolētāja lomu. Pašreiz, atveroties MS Visual Studio konteksta izvēlnei ar visām objekta piedāvātajām metodēm un pazīmēm, var vien paļauties uz savu pieredzi un izstrādātāja saprātīgo vārdu došanu, lai izvēlētos pareizo no dažādām, piemēram, loga pārzīmēšanas metodēm. AOP, ja reiz operē ar jēdzienu "aspekts", vajadzētu pilnveidot šādas priekšāteikšanas sistēmas ar informāciju, kas tieši programmētājam attiecīgajā aspektā piedāvājams.

7.4. Datu vadīta pieeja

Piemērā 5.5. nodaļā ņēmām palīgā metaprogrammēšanas līdzekļus, lai vājinātu neērto piesaisti funkciju nosaukumiem aspekta definēšanā. Tomēr metapazīmju pierakstīšana funkciju definīcijām varētu nelikties pats pilnības kalngals. Tā savā ziņā atkal ir koda dublēšanās, kaut arī aspektā izdalīto divu rindiņu vietā atkārtojas viens vārds.

Kā tad pazīmju pierakstītājs zina, kādas īpašības piemīt attiecīgajai funkcijai? Droši vien, zinot tās saturu. 5.1 un 5.5. funkcijas aplūkojām, nemaz neilustrējot datus, ko tās apstrādā, bet [MM, 102] pietiekami uzsver datu, pretstatā algoritmiem, kā programmēšanas centrālā objekta lomu, lai šāda virspusība darītu uzmanīgu.

Tad nu – vai vienīgi cilvēks spēj pateikt, vai attiecīgā funkcija veic transakciju vai nē? Koncentrēt šīs zināšanas vienā koda fragmentā nozīmētu aprakstīt, kāda izrīcība ar datiem nosaka to, ka tā ir transakcija. Pašreiz AOP valodas cenšas ievest programmēšanas pamatlīdzekļos piešķiršanas un nolasīšanas (*set*, *get*) piesaistes punktu definēšanu pēc iespējas patvaļīgiem datiem. Tālākā attīstībā šie līdzekļi varbūt ļaus arī aprakstīt vispārīgu transakciju un tamlīdzīgas sarežģītākas darbības.

Realitātē izsekot darbībām ar datiem, analizējot programmas tekstu, ir ļoti ierobežotas iespējas. No apstāšanās problēmas algoritmiskās neatrisināmības var izsecināt, ka arī par to, ko funkcija galu galā izdarīs ar tai padotajiem datiem, vispārīgā gadījumā droša slēdziena var nebūt. Praksē viena no populārākajām problēmas izpausmēm ir *aliasing* – spriešanas traucējumi par jebkuru koda fragmentu, kurš satur norādes (*pointers*), jo grūti prognozēt, uz kādiem datiem tās izpildes laikā rādīs. Taču izeja ir – darbībām var sekot līdzi izpildes laikā! To jau dažādos veidos dara Java AOP paplašinājumi, īpaši uzsverot aspektu piesaistīšanu stāvoklim koda plūsmā (skat. 5.4.). Uzskatu, ka būtu svarīgi arī vairāk koncentrēties uz piesaisti datu plūsmā ietvertajai jēgai.

7.5. Pārdefinēšana

AOP dod iespēju bloķēt, respektīvi, mainīt kāda aspekta darbību tādos veidos, kādi nav prātā nākuši oriģinālās programmas autoram. Pašreizējās realizācijās šis aspekts ir tikai funkcijas izsaukums, ko ar `around()` palīdzību var atcelt pavisam vai izpildīt ar mainītām argumentu vērtībām. Bet kādi ir ierobežojumi iespējamajiem citiem programmas "jūtīgajiem punktiem", kuriem derētu pielietot šādus kardinālus pārveidojumus?

Gribētos šo iespēju izmantot, pārskatāmākā veidā definējot, piemēram, 2.5. nodaļā piemērā apskatīto optimizācijas pārveidojumu. Vispār šī iedarbība uz programmas kodu ļoti atgādina tādu OOP pazīstamu jēdzienu kā pārdefinēšana (*overloading*) – piemēram, valodā C++ arī operatoriem un tipu transformācijām jau kādu laiku pieņemts paplašināt uzvedības modeļus, atvieglojot vai, arī tipiski, nesarežģījot (sal. 7.2.) jaunu, spēcīgu programmēšanas līdzekļu pielietošanu. Vai tādā gadījumā nebūtu labi pieturēties arī pie līdzīgām valodas konstrukcijām un meklēt arī vēl citas analoģijas?

7.6. Rīku pielietošana

To, ka ar parasta teksta izteiksmīgumu attīstītām programmēšanas tehnoloģijām ir par maz, var risināt, padarot programmēšanas valodu par interaktīvu programmēšanas vidi.

Paļaušanās uz rīku izmantošanu var atbrīvot no raizēm par valodas sintakses neglītumu, neviennozīmīgumu, dažādo iekavu veidu trūkumu u.tml. – visu neskaidro var skaidri attēlot programmētājam un saņemt skaidras atbildes. Lai taču rīks savos dziļumos programmas teksta reprezentācijai piekārto šo informāciju, cik tik viennozīmīgi iespējams.

[IP] piedāvā pavisam ne uz tekstu orientētu, pilnībā cilvēkam nelasāmu programmas reprezentāciju, kuras rediģēšanas iespējas paplašina līdz ar jaunu programmu (aspektu) izstrādi.

7.7. Pilnība

Teorētiskie pētījumi [AOP 2] piedāvā AOP programmēšanas valodu vispārīgu analītisko bāzi, veiksmīgi modelējot tajā AspectJ un Hyper/J konstrukcijas.

Piemērs, uz kura gribētos novērot iespējamos ieguvumus no aspektu spējas darboties ar pašiem aspektiem, būtu aspektu secības problēma. Pašreizējās realizācijas vairāku vienam punktam piesaistītu aspektu gadījumu risina, ieviešot jaunus atslēgas vārdus, prioritātes un citus nosacījumus.

7.8. Pretrunas

Iespējams, ka visa AOP ved uz milzīgu, nepārskatāmu sistēmu vieglu un bezrūpīgu projektēšanu, bet iespējams arī, ka galarezultāts būs daudz neizdibināmu pretrunu, papildinošajiem aspektiem negaidītos veidos konkurējot un nevēlamos veidos iespaidojot pamata funkcionalitāti.

Kas notiek, ja pamatkoda plūsmā ieviesta aspekta darbībā atgadās kļūda? Tās apstrādi taču pamatkoda rakstītājs var nebūt paredzējis.

Kā, piemēram, attiekties pret `around` aspektu, kurš izlemj neturpināt procedūras izpildi, bet tā ir vitāli nepieciešama pamatkodā realizētajam algoritmam? Parādīsies neparedzama uzvedība? Vai varbūt kaut kādā veidā iespējams tādu aspektu pielietošanu nepieļaut?

8. Risinājumi bez problēmām

Galvenais ieguvums no aizrautības ar pēc iespējas labāku aspektu atdalīšanu ir tas, ka vieglāk tos pēc tam likt kopā visādos derīgos veidos. Tas var radīt augsni jauniem, ērtiem projektēšanas priekšrakstiem (*design patterns*).

Viens no tādiem izplatību gūstošiem priekšrakstiem ir projektēšana ar kontraktiem ([PP, 96]), kas paredz sistemātisku priekšnosacījumu un pēcnosacījumu kontroli programmas moduļu saskarnēm. Šī kontrole palīdz projektēšanā un kļūdu izķeršanā, bet tā nepieder pie programmas pamatuzdevumiem, tātad to būtu vēlams īstenot modulāri, nesarežģījot paša uzdevuma veikšanu – palīdzēt var AOP.

9. Ieskats blakus un nākotnē

9.1. Metaprogrammēšana

Kā vēl viena programmēšanas paradigma, kas attīstās un daudzējādā ziņā saskanīgi papildina AOP, jāmin **metaprogrammēšana** jeb uz valodu orientēta programmēšana ([URL 15], arī [URL 2]). Tās saskatāmie pielietojumi ir programmatūras kvalitātes nodrošināšana, koda un testu ģenerēšana, rīku izstrādes atvieglošana, izstrādes vides un izstrādājamā koda robežu nojaukšana (skat. 2.5.) u.c.

9.2. Pielāgošanās priekšmetapgabalam

Bez modularizācijas vēl viens svarīgs līdzeklis, kurš ļoti atvieglo projektēšanu un programmēšanu, ir **priekšmetapgabalam specifiskas valodas** (*domain-specific languages*, *DSL*). To vērtību uzsver arī [PP, 49]. Praktiski bieži vien no nulles tiek izstrādātas makrovalodas ar pamata iespējām (piemēram, Smarty sagatavju valoda Web lapu izstrādei, http://smarty.php.net/), tomēr visefektīvākā projektētāju un programmētāju sadarbība veidojas, runājot vienā valodā. Tā rodas **priekšmetapgabalam pielāgojamas valodas** (*in-language DSL*, [PP, 53]) jēdziens. Tas nozīmē prasību programmēšanas valodai būt pateicīgai paplašināšanai un pielāgošanai priekšmetapgabalam, ļoti augstā līmenī abstraktai, tai pašā laikā efektīvai no veiktspējas viedokļa. Ne Java, ne C++, par spīti objektorientācijas, operatoru pārlādēšanas un vispārīgās programmēšanas iespējām, par sevišķi pateicīgām šiem mērķiem neuzskata. Lisp un Smalltalk uzskata ([URL 16]).

Minētās īpašības bijušas starp galvenajiem valodas Transframe ([TF]) izstrādes mērķiem. Tā īsteno ļoti paplašināmu sintaksi, ievirza augstā abstrakcijas līmenī objektus, vārdus un atsauces. Projektējuma apraksts sniedz arī interesantas vēsmas objektorientācijas realizācijā, dažas strīdīgas atbildības loģiski noveļot uz programmētāja pleciem ([TF, subtype.htm]). Cilvēku, izstrādājot programmēšanas valodas, tiešām nevajag aizmirst.

9.3. Mērķorientētā programmēšana

Iespējams, AOP gala stacija ir **mērķorientētā programmēšana**, kā latviski varētu tulkot "*intentional programming*" ([IP]). Šo pagaidām neiedzīvināto programmēšanas paradigmu radījis Charles Simonyi, vīrs ar WYSIWYG ("*what you see is what you get*") celmlauža un "ungāru notācijas" autora slavu. Ideja pamatā ir līdzīga AOP idejām (skat. ievadu) – nodrošinot jebkuras programmai izpildāmās prasības (aspekta) modulāru un atvērtu ieprogrammēšanu, tuvināt realizāciju prasību specifikācijai tādā mērā, lai aspektus atkalizmantojot kombinēt būtu tikpat vienkārši kā savirknēt vārdus prasību formulējumā. Tas ļautu izdevušamies aspektiem izdzīvot procesā, kas atgādina dabisko atlasi (sal. 1.4.), uz tiem varētu stabili balstīt arvien jaunus, arvien augstāka līmeņa risinājumus. Citiem vārdiem – vairs nevajadzētu atkal un atkal no jauna izgudrot divriteni.

Microsoft paspārnē iesāktais izpētes projekts pēc 10 gadu darba tika pārtraukts. C. Simonyi nodibināja jaunu uzņēmumu Intentional Software (http://www.intentionalsoftware.com/). Varbūt vēl daudz jaunu prātu būs jāņem palīgā, lai izvirzītais mērķis tiktu sasniegts.

Nobeigums

Aspektorientētā programmēšana palīdz programmatūras izstrādē, sniedzot lielāku rīcības brīvību projektētājiem un novēršot programmētāja darba daudzkāršošanos. Rezultātā paaugstinās darba ražīgums.

Ar šo pētījumu esmu centies gan iepazīties ar specifisko AOP rīku piedāvātajām bāzes iespējām, gan saskatīt aspektorientācijas iespējas lielos vilcienos līdzšinējo paradigmu ietvaros.

Ļoti negribot sevi ierobežot ar citu pētnieku priekšstatiem, kā jau ievadā minēju, esmu centies izvirzīt idejas jauniem AOP risinājumiem. No tām svarīgākās, manuprāt, ir

- mēģināt atbrīvoties no uzspiestiem standartveida "koda ainavas" aprakstiem, kas ne tikai novērš domu no pamatuzdevuma, bet arī samazina aspektu realizāciju savietojamību (7.1.),
- ņemt vērā datu plūsmas, nosakot šķeļošos aspektu saistību (7.4.).

Tā kā praktiskajā programmētāja darbā ikdienā risināmās problēmas esmu paradis analizēt no cilvēka piepūles efektivitātes viedokļa, tad interese par AOP risinājumu īstenošanu un pielietošanu praksē nav atslābusi. Ar to paredzu nodarboties arī zinātniskajos darbos nākotnē.

Literatūras saraksts

[PP] Hunt Andrew, Thomas David. The Pragmatic Programmer. – Addison-Wesley Professional, 1999.
Autoritatīvs efektīvu programmēšanas paņēmienu avots.

[MM] Brooks Frederick P. The Mythical Man-Month. – Addison-Wesley Professional, 1995.
Esejas par darba ražīgumu programminženierijā.

[UC] How To Write Unmaintainable Code [tiešsaiste]. Pieejams Internetā: http://mindprod.com/jgloss/unmain.html
Humoristiska eseja par programmatūras kvalitāti.

[AP] Lieberherr Karl. J., Holland Ian. Assuring good style for object-oriented programs. – IEEE Software, 1989. Pieejams Internetā: ftp://ftp.ccs.neu.edu/pub/demeter/documents/papers/LH89-law-of-demeter.ps
Objektu mijiedarbības sapiņķētības kritizēšanas pirmsākumi.

[AOP 1] Aspect-Oriented Programming / Aut. kol. Gregor Kiczales, John Lamping, Anurag Mendhekar u.c. [tiešsaiste]. Pieejams Internetā: http://www.cs.wm.edu/~coppit/other-papers/kiczales-ECOOP1997-AOP.pdf
AOP pirmavots.

[AOP 2] Clifton Curtis, Leavens Gary T., Wand Mitchell. Parameterized Aspect Calculus: A Core Calculus for the Direct Study of Aspect Oriented Languages. ftp://ftp.cs.iastate.edu/pub/techreports/TR03-13/TR.pdf
Analītisks AOP valodu modelis.

[BS] C++ Answers From Bjarne Stroustrup [tiešsaiste]. Pieejams Internetā: http://www.research.att.com/~bs/slashdot_interview.html
C++ programmēšanas valodas autora atbildes uz kritiskiem jautājumiem.

[TF] Shang David L. Transframe Programming Language [tiešsaiste]. Pieejams Internetā: http://www.visviva.com/transframe/papers/
Dokumentēti jaunas, īpaši elastīgas objektorientētas programmēšanas valodas izstrādes apsvērumi.

[IP] Simonyi Charles. Intentional Programming – Innovation in the Legacy Age [tiešsaiste]. Pieejams Internetā: http://citeseer.ist.psu.edu/rd/77542619%2C539457%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/26368/http:zSzzSzwww.aisto.comzSzroederzSzactivezSzifip96.pdf/simonyi96intentional.pdf
Viss vēlamais programmēšanā.

[URL 1] http://www.aspectc.org/fileadmin/publications/aosd-2004-tut-2x2.pdf
C++ iespējas funkcionalitātes modularizēšanā.

[URL 2] Techniques for Scientific C++ [tiešsaiste]. Pieejams Internetā: http://osl.iu.edu/~tveldhui/papers/techniques/techniques01.html
Augsti abstraktu programmu veiktspējas optimizācijas aspekti.

[URL 3] http://eclipse.org/aspectj/
AspectJ – vispopulārākā AOP valoda, Java paplašinājums.

[URL 4] http://aspectwerkz.codehaus.org/
AspectWerkz – AOP realizācija ar Java standarta līdzekļiem.

[URL 5] http://www.jboss.org/developers/projects/jboss/aop/
JBoss AOP – ar Java standarta līdzekļiem un dinamiskas piesaistīšanas iespējām.

[URL 6] http://www.alphaworks.ibm.com/tech/hyperj/
Hyper/J – simetriska AOP sistēma.

[URL 7] http://www.aspectc.org/
AspectC++ – C++ paplašinājums ar AOP iespējām.

[URL 8] http://www.aosd.net/
Aspektorientētās programmatūras izstrādātāju konference un centrālais informācijas avots.

[URL 9] http://www.oopsla.org/
Objektorientētās programmēšanas sistēmas, valodas un lietojumi.

[URL 10] Kersten Mik. AOP tools comparison [tiešsaiste]. Pieejams Internetā: http://www-128.ibm.com/developerworks/java/library/j-aopwork1/

[URL 11] Lohmann Daniel, Blaschke Georg, Spinczyk Olaf. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++ [tiešsaiste]. Pieejams Internetā: http://www.aspectc.org/fileadmin/publications/gpce-2004.pdf
Vispārīga (tipu neatkarīga), atkalizmantojama kešošanas aspekta implementācija AspectC++.

[URL 12] Laddad Ramnivas. AOP and metadata: A perfect match [tiešsaiste]. Pieejams Internetā: http://www-128.ibm.com/developerworks/java/library/j-aopwork3/

[URL 13] http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp
Intervija ar AOP pamatlicēju Gregor Kiczales.

[URL 14] http://lv.php.net/file_get_contents
Veiksmīgi atdalīti liekie aspekti populārā lietojumā.

[URL 15] Dmitriev Sergey. Language Oriented Programming: The Next Programming Paradigm [tiešsaiste]. Pieejams Internetā: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
Daudzu ar programmēšanu saistīto jaunvārdu apkopojums zem viena nosaukuma, OOP salīdzinot ar akmens laikmetu.

[URL 16] Fowler Martin. Domain Specific Language [tiešsaiste]. Pieejams Internetā: http://www.martinfowler.com/bliki/DomainSpecificLanguage.html
Par programmēšanas valodu pielāgojamības priekšmetapgabalam nozīmi.

Apliecinājums

Ar šo es apliecinu, ka šodien iesniegto kursa darbu es esmu veicis pašrocīgi un esmu izmantojis tikai tajā norādītos palīglīdzekļus.

Uldis Barbans . . . . . .
Rīgā, 2005. g. . . . . .

Kursa darbs izstrādāts
LU Fizikas un matemātikas fakultātes Datorikas nodaļā

Autors
Fizikas un matemātikas fakultātes students
Uldis Barbans . . . . . .
Stud. apl. Nr. DatZ020007
2005. g. . . . . .

Darba vadītājs
Dr. dat., asoc. prof. Kārlis Čerāns . . . . . .
LU Fizikas un matemātikas fakultāte

Darbs iesniegts Datorikas nodaļā 2005. g. . . . . .

Pieņēma sekretāre . . . . . .

Aizstāvēts datorzinātņu kursa pārbaudījumu komisijas sēdē

2005. g. . . . . ar atzīmi . . . . . .

Kursa pārbaudījumu komisija . . . . . .