Projektni uzorci ponašanja (Behavioral design patterns-DP)

Šablonski metod (eng. *Template Method*) – klasni uzorak ponašanja

Namena:

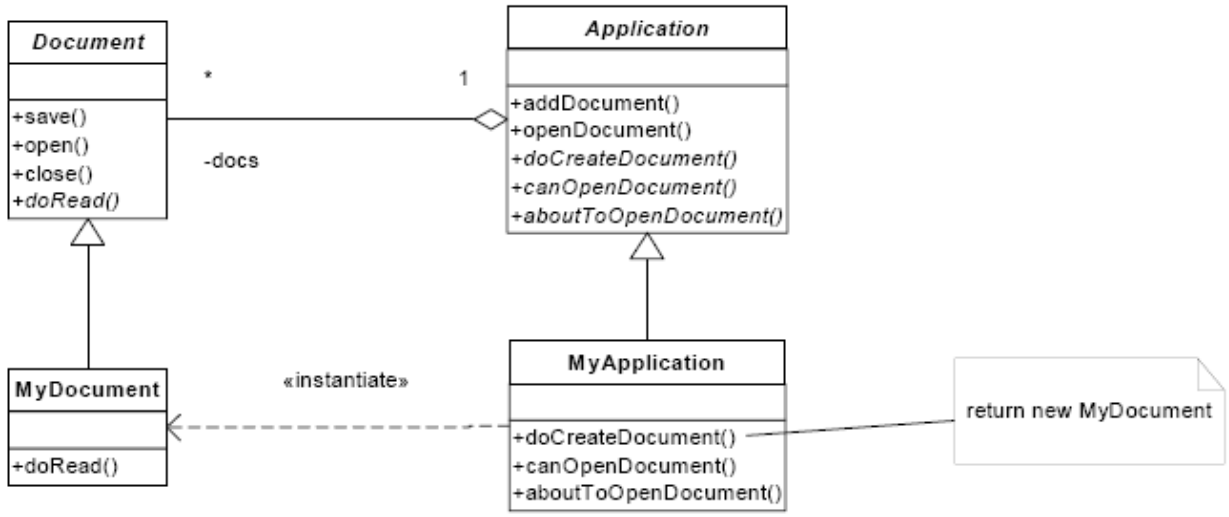
-definiše strukturu (skelet) nekog algoritma, pri čemu se definicija određenih koraka delegira potklasama
- omogućava potklasama da redefinišu određene korake algoritma bez izmene strukture algoritma

Primenljivost

Ovaj DP treba koristiti:
– da se implementiraju invarijantni delovi algoritma jednom, a da se ostave potklasama za implementaciju delovi koji mogu varirati
– za "refactoring to generalize". Naime, kada postoji zajedničko ponašanje za potklasae, onda klasa može obezbediti podrazumevanu implementaciju za određene delove. Tada se u okviru različitih potklasa implementiraju samo oni delovi koji su karakteristični. Ovim se smanjuje dupliranje koda.

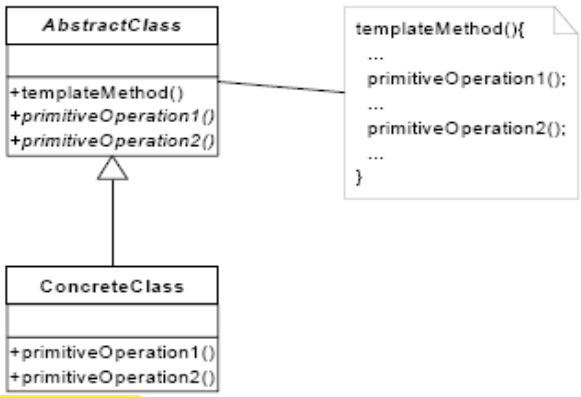
Kratak opis problema

– razmatra se radni okvir za razvoj aplikacije koji obezbeđuje klase Application i Document



– klasa Application je odgovorna za otvaranje postojećih dokumenata
– dokumenti se čuvaju u nekom eksternom formatu (fajlu)
– objekt klase Document reprezentuje informaciju u dokumentu nakon što je učitana iz fajla
– aplikacije koje se grade iz radnog okvira mogu da naslede klase Application i Document (na primer, aplikacija za crtanje definiše DrawApplication i DrawDocument potklase)
– apstraktna klasa Application definiše algoritam za otvaranje dokumenta (algoritam je implementiran u openDocument operaciji)
– openDocument definiše korake za otvaranje dokumenta
– metod openDocument se naziva *Template Method*
– *Template Method* definiše algoritam sastavljen od apstraktnih operacija
– potklase definišu operacije da obezbede konkretno ponašanje
– definisanjem koraka algoritma, *Template Method* fiksira njihov redosled
– potklase prilagođavaju korake algoritma svojim potrebama

DIJAGRAM PRIKAZA



Učesnici:

AbstractClass klasa

- implementira šablon (skelet) algoritma koji poziva primitivne operacije
- deklariše apstraktne primitivne operacije za korake algoritma koje će potklase definisati

ConcreteClass klasa

- implementira primitivne operacije koje obavljaju delove algoritma specifične za potklase

Saradnja:

ConcreteClass implementira varijantne korake algoritma za AbstractClass

Prednosti:

– fundamentalna tehnika za code reuse (zajedničko ponašanje u klasama radnog okvira)
– vodi invertovanoj kontrolnoj strukturi (holivudski princip: "Ne zovite nas, mi ćemo zvati Vas", roditeljska klasa zove operacije dece, a ne obrnuto)
– izbegava se rizik da izvedena klasa iz redefinisane operacije ne pozove potrebnu operaciju roditeljske klase
– mogu se neke primitivne operacije realizovati u apstraktnoj klasi kao rudimentarne *hook* operacije koje obezbeđuju podrazumevano ponašanje

Bliski DP:

– *Factory Method* (npr. doCreateDocument) se često poziva iz *Template Method*
– *Template Method* varira deo algoritma, a *Strategy* varira ceo algoritam

PRIMERI:

1. Ovaj primer ilustruje dizajnerski šablon *Template method* koji obezbeđuje skelet poziva metoda koje zajedno čine algoritam. Jedan ili više koraka mogu biti različiti u svakoj od podklasa. Podklase će implementirati te korake bez uticaja na sekvencu poziva koraka.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         AbstractClass c;

         c = new ConcreteClassA();   c.TemplateMethod();

         c = new ConcreteClassB();    c.TemplateMethod();
```

```
            // cekanje na korisnicki unos
            Console.Read();
        }
    }

    // "AbstractClass"
    abstract class AbstractClass
    {
        public abstract void PrimitiveOperation1();
        public abstract void PrimitiveOperation2();

        // "Template method"
        public void TemplateMethod()
        {
            PrimitiveOperation1();
            PrimitiveOperation2();
            Console.WriteLine("");
        }
    }

    // "ConcreteClass"

    class ConcreteClassA : AbstractClass
    {
        public override void PrimitiveOperation1()
        {
            Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
        }
        public override void PrimitiveOperation2()
        {
            Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
        }
    }

    class ConcreteClassB : AbstractClass
    {
        public override void PrimitiveOperation1()
        {
            Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
        }
        public override void PrimitiveOperation2()
        {
            Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
        }
    }
}
```

```
Izlaz
```

ConcreteClassA.PrimitiveOperation1()
ConcreteClassA.PrimitiveOperation2()

ConcreteClassB.PrimitiveOperation1()
ConcreteClassB.PrimitiveOperation2()

2. Ovaj primer iz stvarnog sveta pokazuje kako šablonski metod Run() obezbeđuje sekvencu poziva različitih metoda. Implementacije ovih metoda su u klasi CustomerDataObject. Ova podklasa implementira metode Connect, Select, Process i Disconnect.

```
using System;
```

```
using System.Data;
using System.Data.OleDb;

namespace ConsoleApplication2
{
   class Program
   {
      static void Main(string[] args)
      {
         DataAccessObject dao;

         dao = new Categories();          dao.Run();

         dao = new Products();          dao.Run();
         Console.Read(); // ceka se unos korisnika
      }
   }

   abstract class DataAccessObject
   {
      protected string connectionString;

      protected DataSet dataSet;

      public virtual void Connect()
      {
         // ako je mdb datoteka na putanji c:\
         connectionString =
           "provider=Microsoft.JET.OLEDB.4.0; " +
           "data source=c:\\nwind.mdb";
      }

      public abstract void Select();
      public abstract void Process();

      public virtual void Disconnect()
      {
         connectionString = "";
      }

      // "Template Method"
      public void Run()
      {
         Connect();          Select();          Process();          Disconnect();
      }
   }

   // "ConcreteClass"
   class Categories : DataAccessObject
   {
      public override void Select()
      {
         string sql = "select CategoryName from Categories";
         OleDbDataAdapter dataAdapter = new OleDbDataAdapter(sql, connectionString);

         dataSet = new DataSet();
         dataAdapter.Fill(dataSet, "Categories");
      }

      public override void Process()
      {
         Console.WriteLine("Categories ---- ");
```

```
            DataTable dataTable = dataSet.Tables["Categories"];
            foreach (DataRow row in dataTable.Rows)
            {
                Console.WriteLine(row["CategoryName"]);
            }
            Console.WriteLine();
        }
    }

    class Products : DataAccessObject
    {
        public override void Select()
        {
            string sql = "select ProductName from Products";
            OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
              sql, connectionString);

            dataSet = new DataSet();
            dataAdapter.Fill(dataSet, "Products");
        }

        public override void Process()
        {
            Console.WriteLine("Products ---- ");
            DataTable dataTable = dataSet.Tables["Products"];
            foreach (DataRow row in dataTable.Rows)
            {
                Console.WriteLine(row["ProductName"]);
            }
            Console.WriteLine();
        }
    }
}
```

```
Categories ----
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Products ----
Chai
Chang
Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
Grandma's Boysenberry Spread
Uncle Bob's Organic Dried Pears
Northwoods Cranberry Sauce
Mishi Kobe Niku
```

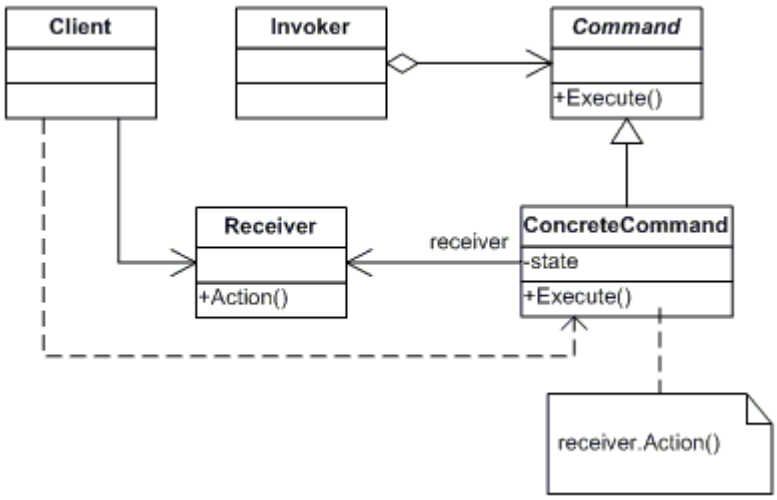
Zadatak za vežbu: Klasa StudentSelectionSort definiše metodu void sort(List<Student> list) za sortiranje liste elemenata tipa Student. Metoda sort je šablon metoda i koristi poziv metode int compare(Student e1, Student e2) da utvrdi odnos između e1 i e2. Metoda compare vraća -1 za e1<e2, 0 za e1==e2 i 1 za e1>e2.
Klasa Student ima atribute: int brojIndeksa, String ime, String Prezime. Napisati klase:

StudentSelectionSort i Student.
Sledeće klase nasleđuju StudentSelectionSort i redefinišu metodu compare:
StudentSortIndeks - compare treba da poredi studente po broju indeksa,
StudentSortIme - compare treba da poredi studente po imenu,
StudentSortPrezime - compare treba da poredi studente po prezimenu.
Sortirati proizvoljan niz elemenata po sva tri kriterijuma. Ispisati sortiran niz na konzoli.
Uraditi isti zadatak primenom strategy dizajn šablona.

Komanda (engl. Command) – objektni uzorak ponašanja

Definicija
Enkapsulira zahtev kao objekat što omogućuje parametrizaciju klijenta različitim zahtevima, nizovima poruka i omogućava realizaciju operacija nad kojima je moguće izvršiti „undo“ operaciju.

UML Dijagram klasa



Učesnici

Command (Command) Deklariše interfejs za izvršenje operacija
ConcreteCommand (CalculatorCommand)
Definiše vezu između objekta klase Receiver i akcije koja mu se upućuje
Implemetira „Execute“ metode pozivajući odgovarajuće operacije u klasi Receiver
Client (CommandApp)
Kreira objekat klase ConcreteCommand i postavlja objekat koji će prihvatiti njegove pozive (Receiver)
Invoker (User) Traži da mu komanda obradi zahtev
Receiver (Calculator) Zna kako da izvršava operacije koje su povezane sa zahtevima

PRIMERI

1. Ovaj primer ilustruje dizajnerski šablon Command koji čuva zahteve kao objekte koji klijentu omogućavaju izvršenje ili opozivanje zahteva.

```
using System;
namespace Komanda
{
  // test aplikacija
```

```
class MainApp
{
  static void Main()
  {
    Receiver receiver = new Receiver();
    Command command = new ConcreteCommand(receiver);
    Invoker invoker = new Invoker();

    // Inicira i izvršava komandu
    invoker.SetCommand(command);
    invoker.ExecuteCommand();

    Console.Read();
  }
}

// "Command"

abstract class Command
{
  protected Receiver receiver;

  // Konstruktor
  public Command(Receiver receiver)
  {
    this.receiver = receiver;
  }

  public abstract void Execute();
}

// "ConcreteCommand"

class ConcreteCommand : Command
{
  // Konstruktor
  public ConcreteCommand(Receiver receiver) :
    base(receiver)
  {
  }

  public override void Execute()
  {
    receiver.Action();
  }
}

// "Receiver"

class Receiver
{
  public void Action()
  {
    Console.WriteLine("Pozvano Receiver.Action()");
  }
}

// "Invoker"

class Invoker
{
  private Command command;

  public void SetCommand(Command command)
  {
    this.command = command;
  }

  public void ExecuteCommand()
  {
```

```
      command.Execute();
    }
  }
}
```

```
Izlaz
Pozvano Receiver.Action()
```

2. Ovaj realan slučaj prikazuje projektni obrazac Command upotrebljen za unapređenje jednostavnog kalkulatora, na taj način što omogućuje neograničen broj „undo“ i „redo“ operacija. Obratite pažnju da je u jeziku C# „operator“ ključna reč, a u ovom primeru je iskorišćen kao promenljivu tako što ispred njega je postavljen znak '@' .

```
using System;
using System.Collections;

namespace KomandaRealanSlucaj
{
  // test aplikacija
  class MainApp
  {
    static void Main()
    {
      // kreiranje korisnika i omogucavanje da korisnik vrsi izracunavanja
      User user = new User();

      user.Compute('+', 100);
      user.Compute('-', 50);
      user.Compute('*', 10);
      user.Compute('/', 2);

      // Undo 4 komande
      user.Undo(4);

      // Redo 3 komande
      user.Redo(3);

      Console.Read();
    }
  }

  // "Command"

  abstract class Command
  {
    public abstract void Execute();
    public abstract void UnExecute();
  }

  // "ConcreteCommand"

  class CalculatorCommand : Command
  {
    char @operator;
    int operand;
    Calculator calculator;

    // Konstruktor
    public CalculatorCommand(Calculator calculator,
      char @operator, int operand)
    {
      this.calculator = calculator;
      this.@operator = @operator;
      this.operand = operand;
    }

    public char Operator
    {
```

```
    set{ @operator = value; }
  }

  public int Operand
  {
    set{ operand = value; }
  }

  public override void Execute()
  {
    calculator.Operation(@operator, operand);
  }

  public override void UnExecute()
  {
    calculator.Operation(Undo(@operator), operand);
  }

  // Undo metod vraca inverziju od @operator
  private char Undo(char @operator)
  {
    char undo;
    switch(@operator)
    {
      case '+': undo = '-'; break;
      case '-': undo = '+'; break;
      case '*': undo = '/'; break;
      case '/': undo = '*'; break;
      default : undo = ' '; break;
    }
    return undo;
  }
}

// "Receiver"

class Calculator
{
  private int curr = 0;

  public void Operation(char @operator, int operand)
  {
    switch(@operator)
    {
      case '+': curr += operand; break;
      case '-': curr -= operand; break;
      case '*': curr *= operand; break;
      case '/': curr /= operand; break;
    }
    Console.WriteLine(
      "Tekuca vrednost = {0,3} (nakon {1} {2})",
      curr, @operator, operand);
  }
}

// "Invoker"

class User
{
  // Inicijalizacije
  private Calculator calculator = new Calculator();
  private ArrayList commands = new ArrayList();

  private int current = 0;

  public void Redo(int levels)
  {
    Console.WriteLine("\n---- Redo {0} nivoa ", levels);
    // Redo operacije
    for (int i = 0; i < levels; i++)
```

```
      {
        if (current < commands.Count - 1)
        {
          Command command = commands[current++] as Command;
          command.Execute();
        }
      }
    }

    public void Undo(int levels)
    {
      Console.WriteLine("\n---- Undo {0} nivoa ", levels);
      //  Undo operacije
      for (int i = 0; i < levels; i++)
      {
        if (current > 0)
        {
          Command command = commands[--current] as Command;
          command.UnExecute();
        }
      }
    }

    public void Compute(char @operator, int operand)
    {
      // Kreirati command operaciju i izvrsenje
      Command command = new CalculatorCommand(
        calculator, @operator, operand);
      command.Execute();

      // Dodati command na undo list
      commands.Add(command);
      current++;
    }
  }
}
```

Izlaz

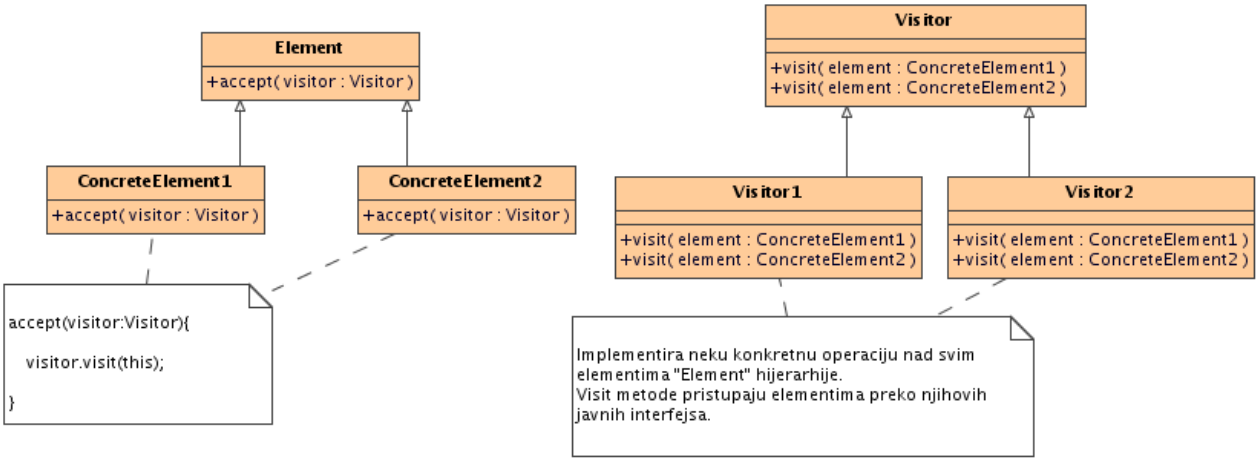
```
Tekuca vrednost = 100 (nakon + 100)
Tekuca vrednost =  50 (nakon - 50)
Tekuca vrednost = 500 (nakon * 10)
Tekuca vrednost = 250 (nakon / 2)

---- Undo 4 nivoa
Tekuca vrednost = 500 (nakon * 2)
Tekuca vrednost =  50 (nakon / 10)
Tekuca vrednost = 100 (nakon + 50)
Tekuca vrednost =   0 (nakon - 100)

---- Redo 3 nivoa
Tekuca vrednost = 100 (nakon + 100)
Tekuca vrednost =  50 (nakon - 50)
Tekuca vrednost = 500 (nakon * 10)
```

Posetilac (engl. Visitor) – objektni uzorak ponašanja

1. Definiše operacije koje se izvode nad elementima strukture objekata bez izmene klasa elemenata nad kojima se operacija izvodi.
2. Na osnovu tipova dva objekta poziva operaciju - Double dispatch.



Osnovni učesnici su:

Apstraktni element hijerarhije objekata (klasa **`Element`**)
generalizacija svih klasa nad kojima se izvodi operacija. Definiše operaciju `accept` koja prihvata `Visitor`-a

Konkretni elementi hijerarhije (klase **`ConcreteElement1,ConcreteElement2...`**)
definišu konkretne elemente klase nad kojima se vrši operacija. Svaki element hijerarhije mora definisati `accept` metodu na sličan način.

Apstraktni **`Visitor`**
definiše `visit` metode za svaki konkretni element hijerarhije koju "posećuje".

Konkretni Visitor (klase **`Visitor1, Visitor2 ...`**)
Svaki konkretni *Visitor* implementira jednu operaciju za svaki element hijerarhije nad kojom se operacija vrši. Redefiniše sve `visit` metode nadklase.

Visitor DP koristi polimorfizam dva puta. Prvi put prilikom poziva `accept` metode elementa da bi se pozvala prava accept metoda konkretne klase a drugi put prilikom poziva `visit` metode da bi se pozvala ispravna metoda konkretnog Visitor-a. Ovaj šablon omogućava implementaciju *Double Dispatch* mehanizma kod programskih jezika koji omogućavaju *Single Dispatch*.

Java PRIMERI

1.
// 1. Dodati accept(Visitor) metod hijerarhiji "element"

/* C++, Java su tzv. dynamic uni-dispatch jezici. Kod njih Visitor tehnikom

double dispatch pomaze da se klasama dodaju operacije , a da se klase ne menjaju.

Ovde je metod accept primer double dispatch operacije,

tj.operacije dvokriterijumske otpreme, jer izvrsavanje operacije accept zavisi od vrste zahteva, ali i od tipa Posetioca

```
(interface Visitor) i tipa Elementa (interface Clan)
*/
// 2. Kreirati "visitor" osnovnu klasu i  visit() method za svaki tip "element"
// 3. Kreirati "visitor" izvedenu klasu za  svaku "operaciju" koja se izvodi nad  "elementima"
// 4. Klijent kreira "visitor" objekte i prosledjuje ga pri pozivu accept()

interface Clan {
  public void accept( Visitor v ); // prvi dispatch     // 1. accept(Visitor)
}                                          //    interface

class Ovaj implements Clan {
  public void   accept( Visitor v ) { v.visit( this ); } // 1. accept(Visitor)
  public String ovajj()             { return "Ovaj"; }   //    implementacija
}
class Neki implements Clan {
  public void   accept( Visitor v ) { v.visit( this ); }
  public String neko()              { return "Neki"; }
}
class Ostali implements Clan {
  public void   accept( Visitor v ) { v.visit( this ); }
  public String preostali()         { return "Ostali"; }
}

interface Visitor {                        // 2. "visitor" kreiranje
  public void visit( Ovaj item ); // drugi dispatch //    osnovna klasa sa
  public void visit( Neki item );                  //    visit() metodom za
  public void visit( Ostali item );                //    svaki "element"
}                                          //    tip

class UpVisitor implements Visitor {          // 3.kreiranje "visitor"
  public void visit( Ovaj item ) {          //   izvedene klase za
    System.out.println( "Up nad " + item.ovajj() ); } // svaku "operaciju"
  public void visit( Neki item ) {
    System.out.println( "Up nad" + item.neko() ); }  //nad  "elementima"
  public void visit( Ostali item ) {
    System.out.println( "Up nad " + item.preostali() ); }
}

class DownVisitor implements Visitor {
  public void visit( Ovaj item ) {      System.out.println( "Down nad " + item.ovajj() ); }
  public void visit( Neki item ) {      System.out.println( "Down nad " + item.neko() ); }
  public void visit( Ostali item ) {      System.out.println( "Down nad " + item.preostali() ); }
}

class Visitor1 {
```

```
  public static Clan[] list = { new Ovaj(), new Neki(), new Ostali() };
  public static void main( String[] args ) {
    UpVisitor    up   = new UpVisitor();            // 4.Klijent kreira
    DownVisitor  down = new DownVisitor();          //  "visitor" objekte
    for (int i=0; i < list.length; i++)             //i svaki prosledjuje
      list[i].accept( up );                         // pozivima accept()
    for (int i=0; i < list.length; i++)
      list[i].accept( down );
}
 }
```

2.

```
//Ilustracija mehanizma  "double dispatch" koji Visitor implementira

//Poreklo termina "double dispatch"
//Naime,OO poruke manifestuju "single dispatch" <=> operacija koja se izvrsava zavisi od:vrste zahteva, tipa primaoca

//Postoje jezici koji direktno podrzavajau "double dispatch" (CLOS).
//Postoje jezici poput Java , C++ koji podrzavaju "single dispatch"

//Jezici koji podrzavaju double dispatch ili multi dispatch umanjuju potrebu za Visitor design pattern-om.

//"double dispatch" <=>operacija koja se izvrsava zavisi od :    vrste zahteva,tipa DVA primaoca
//Operacija accept karakteristicna za Visitor DP je double dispatch.
//Zavisi od tipa Posetioca i tipa Elementa,omogucavajuci posetiocima da zahtevaju razlicite operacije za svaku klasu elementa.

//virtual ? Java?
// virtual =odlaze razresavanje do vremena izvrsavanja programa
// Ako zelimo da deklarisemo ovakvu  "funkciju":    void process( virtual Base object1, virtual Base object2 )
// koja primenjuje algoritam zasnovan  nad  tipovima 2 objekta koja poticu iz jedne hijerarhije, onda postoji problem sto kljucna rec
// "virtual" moze da se ne upotrebi za zahtev dinamickog vezivanja  objekta   koji se prosledjuje.

// Konkretno, kada se pozove  process1()  nad prvim  objektom,
// njegov tip, npr. A, B ili C postaje "poznat" za vreme izvrsavanja,
// ali je tip drugog objekta,  virtual Base, jos uvek  nepoznat.
// Tada se process2() poziva nad drugim objektom, i identitet (samim tim i  tip)  prvog objekta (this) se prosledjuje kao argument.
//Tok kontrole se sada usmerava ka onom mestu  gde je poznat tip oba  objekta(i , naravno, identitet oba  objekta) .

//Umesto statickog vezivanja operacija u interfejs Element,onda izrazavajuci se u terminima dijagram prikaza za Visitor DP,moguce
//je "ucvrstiti" operacije u Visitor-u, i upotrebiti Accept koji bi obavio vezivanje u vreme izvrsavanja, sto je i potrebno.
//Prosirivanje interfejsa Element dobija se dodavanjem nove potklase klase Visitor, umesto mnogo novih potklasa klase Element.
public class VisitorSingle {
interface Base {
  void process1( Base secondObject );
  void process2(  A   firstObject  );
  void process2(  B   firstObject  );
```

```
  void process2(  C   firstObject  );
}
static class A implements Base {
  public void process1( Base drugi )  { drugi.process2( this ); }
    /* Dakle, kada se pozove  process1()  nad prvim  objektom,  njegov tip, npr. A, B ili C postaje  "poznat" za vreme izvrsavanja.
   Potom se process2() poziva nad drugim  objektom, i identitet (samim tim i  tip)  prvog objekta (this) se prosledjuje kao argument.
    */

  public void process2( A prvi ) {     System.out.println( "prvi  A, drugi jeste A" ); }
  public void process2( B prvi ) {     System.out.println( "prvi  B, drugi jeste A" ); }
  public void process2( C prvi ) {     System.out.println( "prvi  C, drugi jeste A" ); }
}

static class B implements Base {
  public void process1( Base drugi ) { drugi.process2( this ); }
  public void process2( A prvi ) {     System.out.println( "prvi A, drugi jeste B" ); }
  public void process2( B prvi ) {     System.out.println( "prvi B, drugi jeste B" ); }
  public void process2( C prvi ) {     System.out.println( "prvi C, drugi jeste B" ); }
}

static class C implements Base {
  public void process1( Base drugi ) { drugi.process2( this ); }
  public void process2( A prvi ) {     System.out.println( "prvi  A, drugi jeste C" ); }
  public void process2( B prvi ) {     System.out.println( "prvi  B, drugi jeste C" ); }
  public void process2( C prvi ) {     System.out.println( "prvi  C, drugi jeste C" ); }
}

public static void main( String[] args ) {
  Base niz[] = { new C(), new A(), new B() };
  for (int i=0; i < niz.length; i++)
    for (int j=0; j < 3; j++)        niz[i].process1( niz[j] );
}
}
```

Kratak opis problema

Kompajler vrši uobičajeno transformaciju programa u neku internu strukturu podataka.(najčešće sintaksno stablo).

Kreiranje sintaksnog stabla je tek početak prevođenja programa.

- Kompajler prolazi kroz stablo i generiše mašinski kod.
- Optimizujući kompajler , takođe prođe kroz stablo i zameni delove stabla nekim drugim delovima.
- Da bi se olakšalo testiranje, zgodno je obezbediti mehanizam koji prolazi kroz stablo i štampa na ekranu sve čvorove stabla.
- Proveravanje korektnosti programa takođe zahteva prolazak kroz stablo.

Svi ovi problemi imaju jednu zajedničku crtu: kompajler prolazi kroz sve delove stabla i uradi nešto nad njima.

Ideja1 (intuitivni prilaz problemu): Neka je *Cvor* osnovna klasa koja reprezentuje čvor sintaksnog stabla. Ideja je deklarisati metod u klasi *Cvor* , koji obavlja željenu operaciju na čvoru u stablu, a onda u svim podklasama se može redefinisati ovaj metod da obavi operaciju koja odgovara konkretnom čvoru.

Mane ideje 1:

- prilično komplikovana implementacija u slučaju da je hijerarhija klasa velika i da postoji potreba za obavljanjem mnogo različitih operacija nad elementima strukture podataka (generisanje koda, štampanje, optimizacije...)
- otežano održavanje programa

Mehanizam ideje 2 – upotreva Visitor DP:

Kreirati apstraktnu klasu koja definiše generalni izgled "posetioca".

```
public abstract class Visitor{
   public void Obrada(Cvor c);
   public void Obrada(Izraz i);
. . .
}
```

I , naravno, svakoj klasi u hijerarhiji dodati metod koji omogućava "posetiocu" da "poseti" dati element i prosledi mu sve potrebne informacije:

```
public void Visit(Visitor v) {
  v.Obrada(this); }
```

Na primer, ako se želi odštampati sintaksno stablo i ako promenljiva *root* sadrži koren stabla onda se preduzima:

```
Visitor v = new Stampati();
root.Visit(v);
```

gde implementacija "posetioca" za štampanje sintaksnog stabla:

```
public class Stampati extends Visitor {
  public void Obrada(Izraz i) {
  i.expr.Visit(this);
  System.out.println(";"); }


public void Obrada(AditivniIzraz i) {
  System.out.print("(");
i.left.Visit(this);
System.out.print(" + ");
i.right.Visit(this);
System.out.print(")"); }
...

}
```

Prednosti primene Visitor DP-a

1. kompletan kod koji definiše operacije nad pojedinačnim elementima definisan je na jednom mestu (*Stampati*).
2. mogućnost jednostavnih modifikacija(u smislu dodavanja novih operacija)
   Npr. ako postoji potreba za operacijom koja radi sve kao i *Stampati*, samo što npr. kod dodeljivanja ne štampa "=" nego "->", dovoljno je definisati podklasu klase *Stampati* i u njoj redefinisati metod Obrada*(VisitorOznacavanja)*. Ali, kod Visitor-a, nekada nije dovoljno definisati novu operaciju samo dodavanjem novog posetioca, već je za dodavanje nove operacije nužno promeniti više klasa (npr. ako je funkcionalnost takva da je raspoređena u više klasa, onda se menja svaka od klasa).
3. poseta različitih hijerarhija klasa je osobina Visitor-a da može da "poseti" objekte koji nemaju zajedničku roditeljsku klasu (što, na primer, nije svojstvo Iterator DP).
4. grupisanjem srodnih, a razdvajanjem različitih operacija, moguĆe je da se svaka struktura podtaka, specifična za algoritam sakrije u posetiocu. Jer, srodno ponašanje se grupiše u posetiocu, umesto da se raspoređuje u klase koje definišu strukturu podataka. Različiti skupovi ponašanja bivaju razdvojeni u sopstvene klase posetilaca, čime se pojednostavljuju klase koje definišu elemente, kao i algoritmi definisani u visitor-ima.
5. tokom "posete" elemenata strukture objekata, Visitor akumulira izvesna stanja. Ako li se ne koristi Visitor, onda bi operacije akumuliranja stanja obavljale globalne promenljive, ili bi se kao dodatni argument operacijama koje vrše obilazak, predavalo stanje.

Mana Visitor Dp-a

Visitor DP nije naročito pogodan u situacijama kada se hijerarhija klasa koje čine elemente strukture podataka često menja u toku razvoja programa.

Npr. kada bi morala da se doda neka klasa u *Cvor* hijerarhiji, npr.*BitwiseIzraz*, tada bi bilo nužno promeniti svaku klasu u *Visitor* hijerarhiji i definisati *metod Obrada(BitwiseIzraz i)* u svakoj od tih klasa.

Primeri korišćenja Visitor DP-a

- Mark Linton je osmislio ime "visitor" u specifikaciji "Fresco Application Toolkit" (kompanija X Consortium) iz januara 1993.
- Jslint je alatka koja statistički skanira Java kod ne bi li našla mesta osetljiva po pitanju sigurnosti. Sami autori su predložili da pošto Jslint parsira source datoteku u sintaksno drvo, onda ono može da se obilazi korišćenjem Visitor design pattern-a. Ovaj pattern enkapsulira svaku operaciju nad sintaksnim drvetom u jedan objekat nazvan Visitor, dozvoljavajući korisnicima da definišu nove operacije nad drvetom bez izmene elemenata drveta. Na taj način se enkodira svaka (sigurnosna) rupa u jedan Visitor koji obilazi drvo raščlanjivanja tragajuci za instancama koje su sporne.

  Sama alatka Jslint može da se modifikuje da skanira kod na drugim jezicima putem ova tri koraka:

  1. zameniti Java gramatiku
  2. napisati novi skup Visitor klasa
  3. dodati novi korisnički interfejs
- Jedna od primena Visitor DP-a je u okviru grep usluga nekih softverskih proizvoda, koji u fajlu pretražuju zadati string.

- Visitor DP (sa metodima accept() i visitConcreteElement()) je još jedan od mehanizama koji omogućuju eliminaciju naredbi višestrukog granjanja.
- Klasa visitor-a se sreće i kod Smalltalk-80 pod imenom ProgramNodeEnumerator koja se upotrebljava za algoritme koji vrše ispitivanja izvornog koda.

Bliski DP

Interpretator DP
Visitor DP se može iskoristiti radi obavljanja interpretacije.

Composite DP
U sistemu koji koristi Composite DP može da se upotrebi Visitor DP da bi se obavile operacije nad pojedinačnim komponentama.