Šabloni projektovanja

Sumarizacija

Nasleđivanje klase je jedan od osnovnih koncepata OOP. Uz pomoć nasleđivanja može se definisati nova vrsta objekata na osnovu stare klase. Tako se dolazi do novih implementacija sa vrlo malo rada nasleđujući osobine starih klasa u sistemu. Ako se nasleđivanje koristi na pravi način, sve klase koje je nasleđuju imaće isti interfejs. Na taj način sve klase mogu da odgovore zahtevu interfejsa ove apstraktne klase. Ako se objektima rukuje isključivo na osnovu interfejsa definisanih u apstraktnoj klasi, postižu se dve stvari:

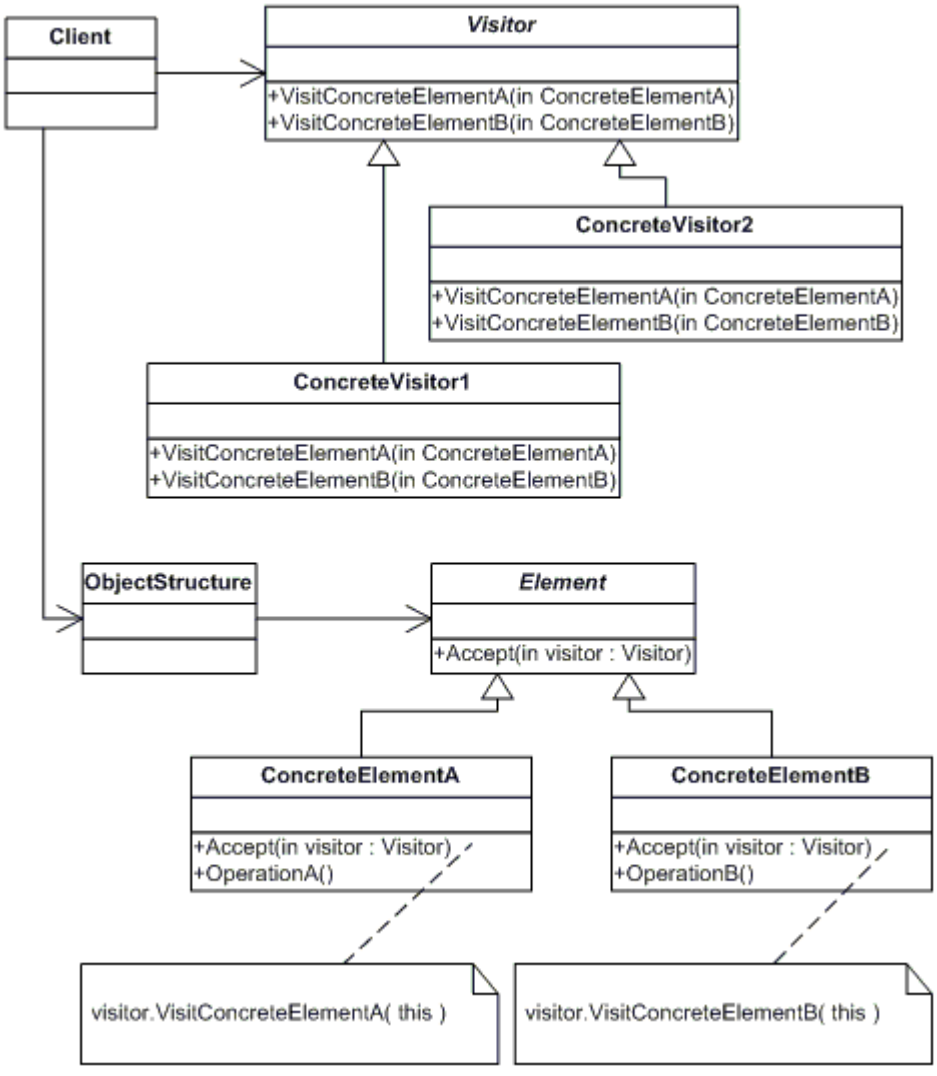
• Klijenti ne moraju da budu svesni tipa objekta koji koriste sve dok se objekti drže interfejsa koji klijenti očekuju.

• Klijenti ne moraju da budu svesni ni klase kojom se implementiraju objekti. Klijenti imaju samo apstraktnu klasu kojom je definisan interfejs.

Ovo do te mere smanjuje zavisnost od implementacije među podsistemima, da dovodi do sledećeg principa višestruke upotrebe u OO projektovanju: **Programirati prema interfejsu, a ne prema klasi.**
Naravno, negde u sistemu mora da se instanciraju konkretne klase, a to se postiže upotrebom šablona projektovanja kao što su: Factory method, Singleton,...
Šabloni omogućavaju da sistem bude napisan prema interfejsima, a ne prema implementacijama.

Zadaci

1. Napisati u C# program koji koristi Visitor nad dva objekta koji prolaze kroz listu zaposlenih i vrše istu operaciju nad svim instancama klase KonkretniZaposleni. Ova dva posetioca definišu različite operacije – jedna postavlja broj slobodnih dana, a druga visinu plate.



Elementi

Klase i/ili objekti koje učestvuju u ovom obrascu su:

- **Visitor (Visitor)**
  - deklariše operaciju Visit za svaku klasu iz ConcreteElement familije. Ime operacije i njen „potpis“ identifikuju klasu koja je poslala zahtev Visit posetiocu. Ovo omogućava posetiocu da jednoznačno odredi klasu objekta koji posećuje. Nakon toga posetilac može da pristupa elementima direktno kroz dostupni interfejs.
- **ConcreteVisitor (PlataVisitor, OdmorVisitor)**
  - implementira svaku operaciju koju deklariše klasa Visitor. Svaka operacija implementira jedan fragment algoritma koji je definisan odgovarajućom klasom ili objektom. Klase iz familije ConcreteVisitor obezbeđuju kontekst za algoritam i čuvajući njegovo lokalno stanje. Ovakvo stanje najčešće služi da akumulira rezultat tokom obilaska strukture.
- **Element (Element)**
  - definiše operaciju Accept koja preuzima objekat Visitor kao argument.
- **ConcreteElement (KonkretniZaposleni)**
  - implementira operaciju Accept koja prihvata objekat Visitor kao argument
- **ObjectStructure (Zaposleni)**
  - može prolaziti kroz strukturu elemenata
  - može da obezbeđuje interfejs visokog nivoa koji dozvoljava Visitor objektima da posećuju elemente

  može da bude ili obrazac Composite ili kolekcija kao što je npr. lista ili skup

```
using System;
using System.Collections;

namespace Zad1
{
 class MainApp
 {
  static void Main()
  {
   // postavljanje kolekcije zaposlenih
   Zaposleni e = new Zaposleni();
   e.Attach(new Radnik());
   e.Attach(new Poslovodja());
   e.Attach(new Upravnik());

   //  'POSETA' zaposlenih sa dva visitora: povišica plate, ispis dana odmora
   e.Accept(new PlataVisitor());
   e.Accept(new OdmorVisitor());
   Console.Read();
  }
 }

 // "Visitor"
 interface IVisitor
 {   void Visit(Element element);}

 // "ConcreteVisitor1", povišica plate
```

```
class PlataVisitor : IVisitor
{
  public void Visit(Element element)
  {
    KonkretniZaposleni employee = element as KonkretniZaposleni;

    // postaviti povišicu prihoda zaposlenog za 10%
    employee.Income *= 1.10;
    Console.WriteLine("{0} {1} - nova plata: {2}", employee.GetType().Name,
employee.Name, employee.Income);
  }
}

// "ConcreteVisitor2", ispis dana odmora
class OdmorVisitor : IVisitor
{
  public void Visit(Element element)
  {     KonkretniZaposleni employee = element as KonkretniZaposleni;

    // ispis dana odmora zaposlenog
    Console.WriteLine("{0} {1} – novi odmor, iznos u danima: {2}", employee.GetType().Name,
employee.Name, employee.VacationDays);
  }
}

class Radnik : KonkretniZaposleni
{
  public Radnik() : base("Pera", 25000, 14){ } // Konstruktor
}

class Poslovodja : KonkretniZaposleni
{
  public Poslovodja() : base("Mika", 35000, 16){  } // Konstruktor
}

class Upravnik : KonkretniZaposleni
{
  public Upravnik() : base("Laza", 45000, 21){  } // Konstruktor
}

// "Element"
abstract class Element
{    public abstract void Accept(IVisitor visitor);}

// "ConcreteElement"
class KonkretniZaposleni : Element
{
  string name;
  double income;
  int vacationDays;

  // Konstruktor
  public KonkretniZaposleni(string name, double income,  int vacationDays)
  {     this.name = name;     this.income = income;
    this.vacationDays = vacationDays;
  }
  // Polja
```

```
  public string Name
  {      get{ return name; }
         set{ name = value; }
  }

  public double Income
  {
    get{ return income; }
    set{ income = value; }
  }

  public int VacationDays
  {
    get{ return vacationDays; }
    set{ vacationDays = value; }
  }

  public override void Accept(IVisitor visitor)
  {
    visitor.Visit(this);
  }
 }

 // "ObjectStructure"
class Zaposleni
 {
  private ArrayList zaposleni = new ArrayList();

  public void Attach(KonkretniZaposleni employee)
  {      zaposleni.Add(employee);}

   public void Detach(KonkretniZaposleni employee)
  {      zaposleni.Remove(employee);}

  public void Accept(IVisitor visitor)
  {
    foreach (KonkretniZaposleni e in zaposleni){        e.Accept(visitor); }
    Console.WriteLine();
  }
 }
}
```

Izlaz

```
Radnik Pera- nova plata: 27500
Poslovodja Mika- nova plata: 38500
Upravnik Laza- nova plata: 49500

Radnik Pera– novi odmor, iznos u danima: 14
Poslovodja Mika– novi odmor, iznos u danima: 16
Upravnik Laza– novi odmor, iznos u danima: 21
```

Primeri upotrebe za GoF obrasce koje nismo detaljnije obrađivali

2. Obrazac Fasada, primer iz stvarnog sveta
Ovaj primer pokazuje upotrebu obrasca Facade kao objekta klase MortgageApplication (hipoteka) koji obezbeđuje uprošćeni interfejs ka skupu klasa koje imaju ulogu da odrede kreditnu sposobnost klijenta.

```
using System;
namespace Zad2
{
  class MainApp
  {
    static void Main()
    {
      // Fasada
      Mortgage mortgage = new Mortgage();

      // Računanje hipoteke za kupca
      Customer customer = new Customer("Pera Peric");
      bool eligable = mortgage.IsEligible(customer,125000);

      Console.WriteLine("\n" + customer.Name +   ", status zahteva za pozajmicu:  " + (eligable ? "odobreno
" : "odbacen"));
      Console.Read(); // Čekanje na korisnički unos
    }
  }

  // "Subsystem ClassA"
  class Bank
  {
    public bool HasSufficientSavings(Customer c, int amount)
    {
      Console.WriteLine("Provera stednje, klijent  " + c.Name);
      return true;
    }
  }

  // "Subsystem ClassB"
  class Credit
  {
    public bool HasGoodCredit(Customer c)
    {
      Console.WriteLine("Provera kredita, klijent  " + c.Name);
      return true;
    }
  }

  // "Subsystem ClassC"
  class Loan
  {
    public bool HasNoBadLoans(Customer c)
    {
      Console.WriteLine("Provera pozajmica, klijent " + c.Name);
      return true;
    }
  }

  class Customer
  {
    private string name;
    public Customer(string name){      this.name = name;}
    public string Name{      get{ return name; } }
  }
```

```
// "Facade"
class Mortgage
{
  private Bank bank = new Bank();
  private Loan loan = new Loan();
  private Credit credit = new Credit();

  public bool IsEligible(Customer cust, int amount)
  {
    Console.WriteLine("{0} zahteva {1:C} na pozajmicu\n", cust.Name, amount);

    bool eligible = true;

    // Provera kreditne sposobnosti
    if (!bank.HasSufficientSavings(cust, amount))
    {       eligible = false;}
    else if (!loan.HasNoBadLoans(cust))
    {       eligible = false;}
    else if (!credit.HasGoodCredit(cust))
    {       eligible = false;}

    return eligible;
  }
 }
}
```

Izlaz
Pera Peric zahteva $125,000.00 na pozajmicu

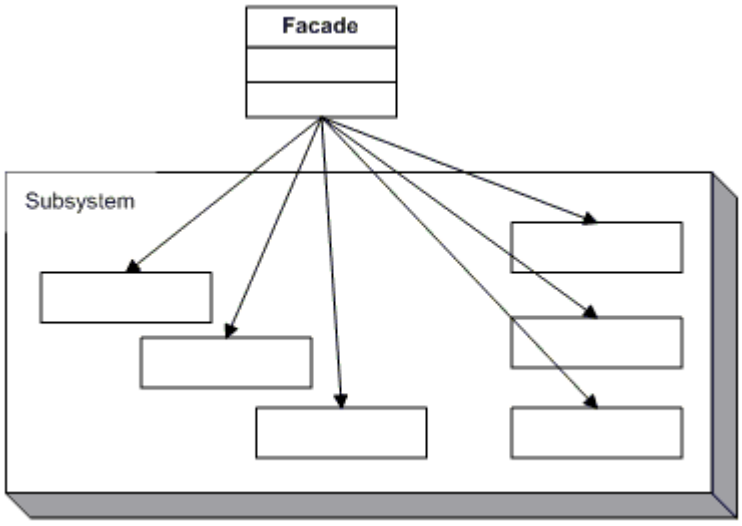
Provera stednje, klijent Pera Peric
Provera pozajmica, klijent Pera Peric
Provera kredita, klijent Pera Peric

Pera Peric, status zahteva za pozajmicu: odobreno

Fasada (Facade), strukturni objektni DP

Definicija: Obezbeđuje jedinstveni interfejs ka celom podsistemu. Obrazac Fasada definiše interfejs na višem nivou koji olakšava upotrebu podsistema.

UML Dijagram klasa obrasca Fasada



Elementi

Klase i/ili objekti koje učestvuju u ovom obrascu su:

- **Facade (MortgageApplication)**
  - zna koje klase podsistema treba da prihvate spoljni zahtev
  - delegira klijentski zahtev odgovarajućem objektu iz podsistema
- **Subsystem classes (Bank, Credit, Loan)**
  - implementiraju funkcionalnost podsistema
  - obrađuju zahteve koji im dolaze od objekta klase Facade
  - nemaju pojma o fasadi i ne čuvaju nikakvu referencu na nju

3. **Interpreter**

Ovaj primer iz stvarnog sveta prikazuje upotrebu obrasca Interpreter koji je upotrebljen za konverziju rimskih u arapske brojeva.

```
using System;
using System.Collections;

namespace Zad3
{

 class MainApp
 {
  static void Main()
  {
   string rimski = "MCMXXXVII";
   Context context = new Context(rimski);

   // klijent gradi   'stablo parsiranja' , raščlanjuju se hiljade, stotine, desetice, jedinice
   ArrayList tree = new ArrayList();
   tree.Add(new ThousandExpression());
   tree.Add(new HundredExpression());
   tree.Add(new TenExpression());
   tree.Add(new OneExpression());

   // klijent poziva operaciju Interpret za terminalne simbole, klase hiljade, stotine, desetice,
//jedinica
   foreach (Expression exp in tree)
   {       exp.Interpret(context);      }

   Console.WriteLine("{0} = {1}", rimski, context.Izlaz);
   Console.Read();
  }
 }

 // "Context"
 class Context
 {
  private string input; // rimski
  private int Izlaz; // vrednost input-a u arapskim ciframa

  // Konstruktor
  public Context(string input)    {      this.input = input;}
```

```
  // Svojstva
  public string Input
  {
    get{ return input; }
    set{ input = value; }
  }

  public int Izlaz
  {
    get{ return Izlaz; }
    set{ Izlaz = value; }
  }
}

// "AbstractExpression"
abstract class Expression
{
  public void Interpret(Context context) // operacija obrade terminalnih simbola hiljada, stotina,..
  {
    if (context.Input.Length == 0)
      return;

    if (context.Input.StartsWith(Nine()))  // ako context.Input pocinje sa CM, XC, IX
    {
      context.Izlaz += (9 * Multiplier()); // Multiplier = 1000, 100, 10, 1
      context.Input = context.Input.Substring(2);
    }
    else if (context.Input.StartsWith(Four())) // ako context.Input pocinje sa CD, XL, IV
    {
      context.Izlaz += (4 * Multiplier());
      context.Input = context.Input.Substring(2);
    }
    else if (context.Input.StartsWith(Five())) // ako context.Input pocinje sa D, L, V
    {
      context.Izlaz += (5 * Multiplier());
      context.Input = context.Input.Substring(1);
    }

    while (context.Input.StartsWith(One())) // ako context.Input pocinje sa M, C, X, I
    {
      context.Izlaz += (1 * Multiplier());
      context.Input = context.Input.Substring(1);
    }
  }

  public abstract string One();
  public abstract string Four();
  public abstract string Five();
  public abstract string Nine();
  public abstract int Multiplier();
}

// Racunanje hiljada,  rimski numeral M
// "TerminalExpression"

class ThousandExpression : Expression
{
```

```
  public override string One() { return "M"; }
  public override string Four(){ return " "; }
  public override string Five(){ return " "; }
  public override string Nine(){ return " "; }
  public override int Multiplier() { return 1000; }
}

// Racunanje stotina, rimski numerali C, CD, D, CM
// "TerminalExpression"

class HundredExpression : Expression
{
  public override string One() { return "C"; }
  public override string Four(){ return "CD"; }
  public override string Five(){ return "D"; }
  public override string Nine(){ return "CM"; }
  public override int Multiplier() { return 100; }
}

// Racunanje desetica, rimski numerali X, XL, L, XC
// "TerminalExpression"

class TenExpression : Expression
{
  public override string One() { return "X"; }
  public override string Four(){ return "XL"; }
  public override string Five(){ return "L"; }
  public override string Nine(){ return "XC"; }
  public override int Multiplier() { return 10; }
}

// Racunanje jedinica: I, II, III, IV, V, VI, VI, VII, VIII, IX
// "TerminalExpression"

class OneExpression : Expression
{
  public override string One() { return "I"; }
  public override string Four(){ return "IV"; }
  public override string Five(){ return "V"; }
  public override string Nine(){ return "IX"; }
  public override int Multiplier() { return 1; }
 }
}
```

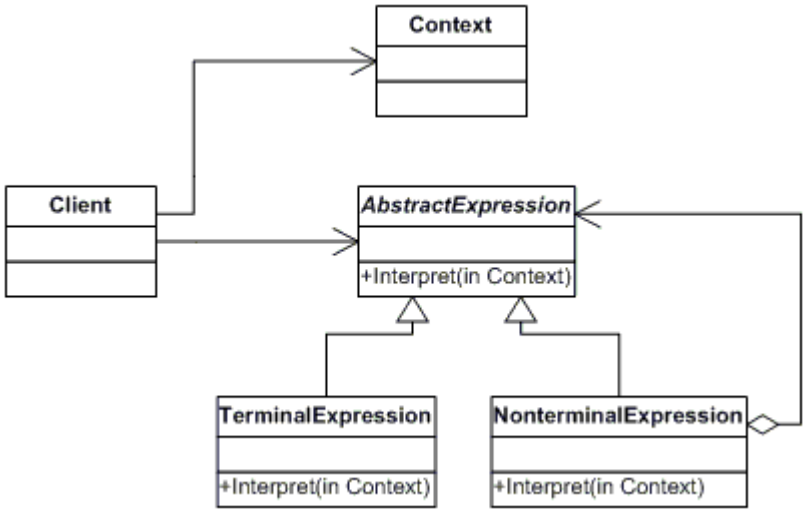
Izlaz

```
MCMXXXVII = 1937
```

Interpretator, klasni obrazac ponašanja

Definicija: Služi da za dati jezik, definiše reprezentaciju njegove gramatike zajedno sa pravilima interpretiranja rečenica.

UML Dijagram klasa



Elementi

Klase i/ili objekti koje učestvuju u ovom obrascu su:

- **AbstractExpression (Expression)**
  - Deklariše interfejs za izvršenje operacija
- **TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)**
  - Implementira operaciju Interpret koja je asocirana uz terminalne simbole gramatike.
  - Za svaki terminalni simbol potrebna je po jedna instanca terminalnog simbola iz rečenice.
- **NonterminalExpression (not used)**
  - Po jedna ovakva klasa je potrebna za svako od pravila tipa R ::= R1R2...Rn koje postoji u gramatici
  - Vodi računa o instancama tipa AbstractExpression za svaki od R-ova (R1...Rn.)
  - Implementira operaciju Interpret za neterminalne simbole u gramatici. Interpretacija tipično zove sebe samu rekurzivno za svaki od R-ova.
- **Context (Context)**
  - Sadrži globalne informacije potrebne interpretatoru
- **Client (InterpreterApp)**
  - Gradi apstraktno sintaksno stablo koje predstavlja određenu rečenicu u jeziku koji je definisan gramatikom. Pomenuto sintaksno stablo je sačinjeno od instanca klasa NonterminalExpression i TerminalExpression.

Poziva operaciju Interpret

4. Obrazac State

Ovaj primer iz stvarnog sveta prikazuje upotrebu obrasca State koji dozvoljava da se objekat klase Account (Račun) ponaša drugačije u zavisnosti od količine raspoloživih sredstava. Razlika u ponašanju je delegirana kroz objekte RedState, SilverState i GoldState. Ova stanja opisuju račun koji je u minusu, tek otvoreni račun i račun sa „dobrom istorijom“.

```
using System;

namespace Zad4
{

 class MainApp
 {
  static void Main()
  {
   // otvaranje racuna
   Account account = new Account("Pera Peric");

   //  Finansijske transakcije tekuceg korisnika racuna
   account.Deposit(500.0);
   account.Deposit(300.0);
   account.Deposit(550.0);
   account.PayInterest();
   account.Withdraw(2000.00);
   account.Withdraw(1100.00);
   Console.Read();
  }
 }

 // "State"
 abstract class State
 {
  protected Account account;
  protected double balance;

  protected double interest;
  protected double lowerLimit;
  protected double upperLimit;

  // Svojstva
  public Account Account
  {
   get{ return account; }
   set{ account = value; }
  }

  public double Balance
  {
   get{ return balance; }
   set{ balance = value; }
  }

  public abstract void Deposit(double amount);
  public abstract void Withdraw(double amount);
  public abstract void PayInterest();
 }

 // "ConcreteState", opis računa koji je u minusu
 class RedState : State
 {    double serviceFee;
```

```
  // Konstruktor
  public RedState(State state)
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  private void Initialize()
  {
    // na primer, podaci uzeti iz datoteke RedStateOpis.txt
    interest = 0.0;
    lowerLimit = -100.0;
    upperLimit = 0.0;
    serviceFee = 15.00;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    amount = amount - serviceFee;
    Console.WriteLine("Nema sredstava za isplatu!");
  }

  public override void PayInterest()
  {
    // bez placanja interesa
  }

  private void StateChangeCheck()
  {
    if (balance > upperLimit)
    {
      account.State = new SilverState(this);
    }
  }
}

// "ConcreteState" , opis tek otvorenog racuna
class SilverState : State
{
  // Overload-ovani konstruktor
  public SilverState(State state) : this( state.Balance, state.Account)    {      }

  public SilverState(double balance, Account account)
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  private void Initialize()
```

```
  {
    // na primer, podaci iz datoteke SilverStateOpis.txt
    interest = 0.0;
    lowerLimit = 0.0;
    upperLimit = 1000.0;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    balance -= amount;
    StateChangeCheck();
  }

  public override void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  private void StateChangeCheck()
  {
    if (balance < lowerLimit)
    {
      account.State = new RedState(this);
    }
    else if (balance > upperLimit)
    {
      account.State = new GoldState(this);
    }
  }
}

// "ConcreteState", opis racuna sa dobrom istorijom
class GoldState : State
{
  // Overload-ovani konstruktori
  public GoldState(State state)       : this(state.Balance,state.Account)    {      }

  public GoldState(double balance, Account account)
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  private void Initialize()
  {
    // na primer, podaci iz datoteke GoldStateOpis.txt
    interest = 0.05;
    lowerLimit = 1000.0;
    upperLimit = 10000000.0;
```

```
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    balance -= amount;
    StateChangeCheck();
  }

  public override void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  private void StateChangeCheck()
  {
    if (balance < 0.0)
    {
      account.State = new RedState(this);
    }
    else if (balance < lowerLimit)
    {
      account.State = new SilverState(this);
    }
  }
}

// "Context"

class Account
{
  private State state;
  private string owner;

  // Konstruktor
  public Account(string owner)
  {
    // novi racun je  'Silver' racun
    this.owner = owner;
    state = new SilverState(0.0, this);
  }

  // Svojstva
  public double Balance
  {
    get{ return state.Balance; }
  }

  public State State
  {
    get{ return state; }
```

```
    set{ state = value; }
  }

  public void Deposit(double amount)
  {
    state.Deposit(amount);
    Console.WriteLine("Deponovano {0:C} --- ", amount);
    Console.WriteLine(" Stanje = {0:C}", this.Balance);
    Console.WriteLine(" Status = {0}\n" ,
      this.State.GetType().Name);
    Console.WriteLine("");
  }

  public void Withdraw(double amount)
  {
    state.Withdraw(amount);
    Console.WriteLine("Isplaceno {0:C} --- ", amount);
    Console.WriteLine(" Stanje = {0:C}", this.Balance);
    Console.WriteLine(" Status = {0}\n" ,
      this.State.GetType().Name);
  }

  public void PayInterest()
  {
    state.PayInterest();
    Console.WriteLine("Placanje interesa --- ");
    Console.WriteLine(" Stanje = {0:C}", this.Balance);
    Console.WriteLine(" Status = {0}\n" ,
      this.State.GetType().Name);
  }
 }
}
```

Izlaz

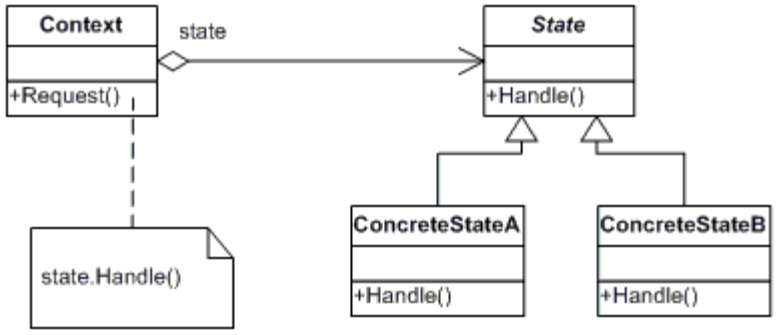
```
Deponovano $500.00 ---
 Stanje = $500.00
 Status = SilverState


Deponovano $300.00 ---
 Stanje = $800.00
 Status = SilverState


Deponovano $550.00 ---
 Stanje = $1,350.00
 Status = GoldState


Placanje interesa ---
 Stanje = $1,417.50
 Status = GoldState

Isplaceno $2,000.00 ---
 Stanje = ($582.50)
 Status = RedState
```

```
Nema sredstava za isplatu!
Isplaceno $1,100.00 ---
 Stanje = ($582.50)
 Status = RedState
```

State, objektni uzor ponašanja

Definicija: Dozvoljava da objekat promeni ponašanje kada se njegovo interno stanje promeni.

UML Dijagram klasa



Elementi

Klase i/ili objekti koje učestvuju u ovom obrascu su:

- **Context (Account)**
  - definiše interfejs koji je potreban klijentu
  - vodi računa o instanci klase ConcreteState koja definiše trenutno stanje
- **State (State)**
  - definiše interfejs za enkapsulaciju ponašanja koje je asocirano sa određenim stanjem klase Context.
- **Concrete State (RedState, SilverState, GoldState)**

svaka podklasa implementira ponašanje koje je asocirano sa odgovarajućim stanjima klase Context