Razvoj softvera

Saša Malkov

Odlomak iz knjige u pripremi...

8.4 ...Primer refaktorisanja

Pravi smisao refaktorisanja se najbolje vidi na nešto složenijim primerima. Pokušaćemo da sagledamo primenu nekih od osnovnih tehnika refaktorisanja na jednom zanimljivom primeru programa. Posmatrani problem nije posebno složen i relativno je jednostavan za razumevanje, ali ipak dopušta da se naprave neke greške u dizajnu, koje kasnije možemo da popravljamo refaktorisanjem.

Radi se o programu za sprovođenje testiranja korisnika. Program sprovodi testiranje postavljanjem pitanja sa ponuđenim odgovorima. Korisnik bira jedan od ponuđenih odgovora upisivanjem odgovarajućeg slova. Tačan odgovor nosi 5 poena, pogrešan odgovor nosi -3 poena, a odgovor „Ne znam“ nosi -1 poen. Jedan primer izvršavanja programa bi mogao da bude:

```
*** Pitanje br. 1 ***

Sta je to 'std::map'?
----------------------------------------
a - Putokaz do gusarskog blaga.
b - Asocijativni niz u standardnoj biblioteci P.J. C++.
c - Mocvarni teren opasan po zivot.
----------------------------------------
Upisite slovo koje stoji ispred tacnog odgovora
ili znak @ ako ne znate: b


*** Pitanje br. 2 ***

Koliko tacnih odgovora ima ovo pitanje?
----------------------------------------
```

```
a - Tri.
b - Dva.
c - Jedan.
-----------------------------------------
Upisite slovo koje stoji ispred tacnog odgovora
ili znak @ ako ne znate: b


Rezultat testiranja:
-----------------------------------------
1.  ( 5 ) Tacan odgovor.
2.  (-3 ) Netacan odgovor.
-----------------------------------------
Rezultat: 2 poena
```

Podsetićamo čitaoce da je u praksi neophodno da se refaktorisanje radi uz doslednu upotrebu testova jedinica koda. U ovom primeru se ne koriste testovi jedinica koda, pre svega zbog ograničenog prostora, ali i radi usmeravanja pažnje čitalaca na same probleme i tehnike, a ne na prateće testove.

U planu je da se program proširuje dodavanjem novih vrsta pitanja i mogućnosti zapisivanja izveštaja o testiranju u datoteci. Program nije dovoljno dobro dizajniran, pa dodavanje novih mogućnosti nije jednostavno. Zbog toga ćemo nizom refaktorisanja pokušati da popravimo dizajn programa. Nakon nekoliko refaktorisanja uslediće dodavanje nekih novih mogućnosti, a zatim ćemo nastaviti sa još nekoliko refaktorisanja.

Program pre refaktorisanja

Na početku, program čine dve klase: `Test` i `Pitanje`. Klasa `Pitanje` predstavlja jedno pitanje i ponuđene odgovore. Ima samo konstruktor i pristupne metode `TekstPitanja`, `TacanOdgovor` i `PonudjeniOdgovori`. Klasa `Test` predstavlja model jednog testa i može da obuhvati veći broj pitanja. Ima metode:

- podrazumevani konstruktor – pravi prazan test (bez pitanja);
- `Dodaj` – dodaje jedno pitanje testu;
- `PostaviPitanja` – izvodi testiranje i ispisuje izveštaj o rezultatima testiranja.

Sledi početni programski kod:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;
```

```
//---------------------------------------------------------------
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja,
             const string& tacanOdgovor,
             const string& odgovor1,
             const string& odgovor2 = "",
             const string& odgovor3 = "",
             const string& odgovor4 = "",
             const string& odgovor5 = ""
        )
        : TekstPitanja_( tekstPitanja ),
          TacanOdgovor_( tacanOdgovor )
    {
        PonudjeniOdgovori_.push_back( odgovor1 );
        if( odgovor2 != "" )
            PonudjeniOdgovori_.push_back( odgovor2 );
        if( odgovor3 != "" )
            PonudjeniOdgovori_.push_back( odgovor3 );
        if( odgovor4 != "" )
            PonudjeniOdgovori_.push_back( odgovor4 );
        if( odgovor5 != "" )
            PonudjeniOdgovori_.push_back( odgovor5 );
    }

    const string& TekstPitanja() const
        { return TekstPitanja_; }
    const string& TacanOdgovor() const
        { return TacanOdgovor_; }
    const vector<string>& PonudjeniOdgovori() const
        { return PonudjeniOdgovori_; }

private:
    string TekstPitanja_;
    string TacanOdgovor_;
    vector<string> PonudjeniOdgovori_;
};

//---------------------------------------------------------------
class Test
{
public:
    void Dodaj( const Pitanje& p )
        { Pitanja_.push_back(p); }

    void PostaviPitanja()
    {
        string odgovor;
        vector<string> rezultati;
        int zbirPoena = 0;
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            Pitanje& pitanje = Pitanja_[i];
            cout << endl
                 << "*** Pitanje br. " << (i+1) << " ***"
```

```
                << endl << endl
                << pitanje.TekstPitanja() << endl
                << "----------------------------------------"
                << endl;
            for( unsigned j=0;
                 j<pitanje.PonudjeniOdgovori().size();
                 j++
               )
                cout << pitanje.PonudjeniOdgovori()[j] << endl;
            cout << "----------------------------------------"
                 << endl
                 << "Upisite slovo koje stoji "
                    "ispred tacnog odgovora " << endl
                 << "ili znak @ ako ne znate: ";
            cin >> odgovor;
            if( odgovor == pitanje.TacanOdgovor() ) {
                rezultati.push_back("( 5 ) Tacan odgovor.");
                zbirPoena += 5;
            } else if( odgovor == "@" ) {
                rezultati.push_back("(-1 ) Nije odgovoreno.");
                zbirPoena -= 1;
            } else {
                rezultati.push_back("(-3 ) Netacan odgovor.");
                zbirPoena -= 3;
            }
            cout << endl;
        }

        cout << endl << "Rezultat testiranja:" << endl
             << "----------------------------------------"
             << endl;
        for( unsigned i=0; i<Pitanja_.size(); i++ )
        cout << (i+1) << ".  "
             << rezultati[i] << endl;
        cout << "----------------------------------------"
             << endl
             << "Rezultat: " << zbirPoena << " poena" << endl;
    }

private:
    vector<Pitanje> Pitanja_;
};

//-------------------------------------------------------------
int main( int argc, char** argv )
{
    Test test;
    test.Dodaj( Pitanje(
        "Sta je to 'std::map'?",
        "b",
        "a - Putokaz do gusarskog blaga.",
        "b - Asocijativni niz u standardnoj biblioteci P.J. C++.",
        "c - Mocvarni teren opasan po zivot."
    ));
```

```
    test.Dodaj( Pitanje(
        "Koliko tacnih odgovora ima ovo pitanje?",
        "c",
        "a - Tri.",
        "b - Dva.",
        "c - Jedan."
    ));

    test.PostaviPitanja();
}
```

Korak 1 – Izdvajanje metoda PostaviPitanje

Predstavljen program radi u skladu sa opisom, ali već i površno pregledanje koda je dovoljno da se uoči da je praktično sva složenost programa smeštena u jednom metodu: `Test::PostaviPitanja`. Ovaj metod je obuhvatio previše različitih stvari, od postavljanja pitanja, proveravanja i bodovanja odgovora, pa sve do ispisivanja izveštaja o obavljenom testiranju. Zbog toga bi prvih nekoliko koraka u refaktorisanju trebalo da budu posvećeni izdvajanju delova koda iz ovog metoda u nove metode.

Prvi kandidat može da bude deo koda za postavljanje pojedinačnog pitanja:

```
    void PostaviPitanja()
    {
        string odgovor;
        ...
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            Pitanje& pitanje = Pitanja_[i];
            cout << endl
                 << "*** Pitanje br. " << (i+1) << " ***"
                 << endl << endl
                 << pitanje.TekstPitanja() << endl
                 << "----------------------------------------"
                 << endl;
            for( unsigned j=0;
                 j<pitanje.PonudjeniOdgovori().size();
                 j++
               )
                cout << pitanje.PonudjeniOdgovori()[j] << endl;
            cout << "----------------------------------------"
                 << endl
                 << "Upisite slovo koje stoji "
                    "ispred tacnog odgovora " << endl
                 << "ili znak @ ako ne znate: ";
            cin >> odgovor;
            ...
        }
        ...
    }
```

Označeni deo koda izdvajamo u novi privatni metod `Test::PostaviPitanje`. Dodajemo potrebne argumente i lokalnu promenljivu `odgovor`:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl
         << pitanje.TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
        )
        cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Zatim popravljamo metod `Test::PostaviPitanja` tako da koristi novi metod. Takođe, premeštamo definiciju lokalne promenljive odgovor sve do mesta na kome se inicijalizuje:

```
void PostaviPitanja()
{
    ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        ...
    }
    ...
}
```

Korak 2 – Izdvajanje metoda ProveriOdgovor

Nastavljamo sa popravljanjem pretrpanog metoda `Test::PostaviPitanja`. Sada ćemo iz metoda `PostaviPitanja` da izdvojimo deo koji se odnosi na proveravanje ispravnosti odgovora:

```
void PostaviPitanja()
{
    ...
```

```
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        if( odgovor == pitanje.TacanOdgovor() ) {
            rezultati.push_back("( 5 ) Tacan odgovor.");
            zbirPoena += 5;
        } else if( odgovor == "@" ) {
            rezultati.push_back("(-1 ) Nije odgovoreno.");
            zbirPoena -= 1;
        } else {
            rezultati.push_back("(-3 ) Netacan odgovor.");
            zbirPoena -= 3;
        }
        cout << endl;
    }
    ...
}
```

Pravimo nov privatan metod `Test::ProveriOdgovor`:

```
int ProveriOdgovor( const Pitanje& pitanje,
                    const string& odgovor,
                    vector<string>& rezultati ) const
{
    if( odgovor == pitanje.TacanOdgovor() ){
        rezultati.push_back("( 5 ) Tacan odgovor.");
        return 5;
    }else if( odgovor == "@" ){
        rezultati.push_back("(-1 ) Nije odgovoreno.");
        return -1;
    }else{
        rezultati.push_back("(-3 ) Netacan odgovor.");
        return -3;
    }
}
```

Popravljamo metod `Test::PostaviPitanja` tako da koristi nov metod za proveravanje ispravnosti odgovora:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<_Pitanja.size(); i++ ){
        Pitanje& pitanje = _Pitanja[i];
        string odgovor = PostaviPitanje( i, pitanje );
        zbirPoena +=
            ProveriOdgovor( pitanje, odgovor, rezultati );
        cout << endl;
    }
    ...
}
```

Korak 3 – Izdvajanje i premeštanje metoda PostaviPitanje

Primetimo da metod `Test::PostaviPitanje` ima dve celine koje nemaju isti domen primene. Prvi deo, koji ispisuje redni broj pitanja, predstavlja pripremu za postavljanje pitanja i zavisi od konteksta u kome se izvršava, ali ne i od pitanja koje se postavlja. Njemu je mesto u klasi `Test`. Međutim, drugi deo, koji zaista postavlja pitanje, pre pripada klasi `Pitanje` nego klasi `Test`.

Najpre ćemo postojeći metod podeliti na dva:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***";
         << endl << endl;
    return PostaviPitanje( pitanje );
}

string PostaviPitanje( const Pitanje& pitanje ) const
{
    cout << pitanje.TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
        )
        cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Zatim novi metod premeštamo u klasu `Pitanje`:. Više nije neophodan eksplicitan argument `pitanje`, zato što će ga zameniti implicitan argument `this`:

```
string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
```

```
            << "ili znak @ ako ne znate: ";
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }
```

Preostaje nam još da popravimo metod `Test::PostaviPitanje`:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl;
    return pitanje.PostaviPitanje();
}
```

Korak 4 – Premeštanje metoda ProveriOdgovor

U prethodnom koraku smo premestili metod `PostaviPitanje` (t.j. deo prvobitne verzije tog metoda) iz klase `Test` u klasu `Pitanje`. I metod `ProveriOdgovor` zaslužuje isti tretman, zato što se odnosi isključivo na pojedinačno pitanje, a ne na test kao celinu.

Primetimo da je pre premeštanja ovog metoda potrebno da blago izmenimo njegovu semantiku – u klasi `Test` ovaj metod dodaje rezultat u niz rezultata, ali nije dobro da zahtevamo da metod klase `Pitanje` *zna* za niz rezultata. Zbog toga ćemo da promenimo drugi argument metoda tako da umesto niza rezultata to bude samo niska prenesena po referenci.

Najpre popravljamo metod `Test::ProveriOdgovor`:

```
int ProveriOdgovor( const Pitanje& pitanje,
                    const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == pitanje.TacanOdgovor() ){
        rezultat = "( 5 ) Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "(-1 ) Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "(-3 ) Netacan odgovor.";
        return -3;
    }
}
```

Zatim popravljamo i metod `Test::PostaviPitanja`:

```
void PostaviPitanja()
{
```

```
    ...
        string rezultat;
        zbirPoena +=
            ProveriOdgovor( pitanje, odgovor, rezultat );
        rezultati.push_back(rezultat);
    ...
}
```

Sada metod `ProveriOdgovor` možemo da premestimo u klasu `Pitanje`:

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "( 5 ) Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "(-1 ) Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "(-3 ) Netacan odgovor.";
        return -3;
    }
}
```

Ostaje nam još da popravimo metod `Test::PostaviPitanja`:

```
void PostaviPitanja()
{
    ...
        zbirPoena +=
            pitanje.ProveriOdgovor( odgovor, rezultat );
    ...
}
```

Korak 5 – Izdvajanje metoda Izvestaj iz PostaviPitanja

Prethodnim koracima smo malo pojednostavili metod `Test::PostaviPitanja`, ali on i dalje radi nekoliko različitih stvari. Da bismo ga dodatno rasteretili, potrebno je da se ispisivanje izveštaja izdvoji u poseban metod `Test::Izvestaj`. Umesto jednog metoda:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<_Pitanja.size(); i++ ){
        ...
    }

    cout << endl << "Rezultat testiranja:" << endl
```

```
            << "----------------------------------------"
            << endl;
        for( unsigned i=0; i<_Pitanja.size(); i++ )
        cout << (i+1) << ".  "
            << rezultati[i] << endl;
        cout << "----------------------------------------"
            << endl
            << "Rezultat: " << zbirPoena << " poena" << endl;
    }
```

sada ćemo imati dva odvojena metoda:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<_Pitanja.size(); i++ ){
        ...
    }
    Izvestaj( zbirPoena, rezultati );
}

void Izvestaj( int zbirPoena,
               const vector<string>& rezultati ) const
{
    cout << endl << "Rezultat testiranja:" << endl
         << "----------------------------------------"
         << endl;
    for( unsigned i=0; i<_Pitanja.size(); i++ )
    cout << (i+1) << ".  "
             << rezultati[i] << endl;
    cout << "----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Korak 6 – Pamćenje odgovora umesto rezultata

Dobijeni kod je nešto bolji nego što je bio, ali još uvek ima mnogo slabosti. Jedan od problema predstavlja prenošenje podataka o zbiru poena i rezultatima između delova za postavljanje pitanja i ispisivanje izveštaja. Ustvari, problem nije samo u prenošenju podataka već i mestu njihovog pravljenja. Pravo mesto za analizu odgovora nije deo u kome se postavljaju pitanja, već je to pre deo u kome se ispisuje izveštaj. Da bismo mogli da odvojimo ta dva aspekta ponašanja, najpre je neophodno da se pri postavljanju pitanja zapamte svi odgovori.

Da bismo mogli da zapamtimo odgovore, klasi `Test` dodajemo privatni podatak `_Odgovori`:

```
vector<string>   _Odgovori;
```

Zatim menjamo metod `Test::PostaviPitanja` tako da pamti odgovore:

```
void PostaviPitanja()
{
    _Odgovori.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<_Pitanja.size(); i++ ){
        Pitanje& pitanje = _Pitanja[i];
        odgovor = PostaviPitanje( i, pitanje );
        _Odgovori.push_back( odgovor );
        string rezultat;
        zbirPoena +=
            pitanje.ProveriOdgovor( odgovor, rezultat );
        rezultati.push_back(rezultat);
        cout << endl;
    }

    Izvestaj( zbirPoena, rezultati );
}
```

Zatim izbacujemo pomoćnu promenljivu `pitanje`:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string odgovor = PostaviPitanje( i, Pitanja_[i] );
        Odgovori_.push_back( odgovor );
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( odgovor, rezultat );
        rezultati.push_back( rezultat );
        cout << endl;
    }
    Izvestaj( zbirPoena, rezultati );
}
```

Na sličan način izbacujemo i pomoćnu promenljivu `odgovor`:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ));
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
```

```
            rezultati.push_back( rezultat );
            cout << endl;
        }
        Izvestaj( zbirPoena, rezultati );
    }
```

Pa podelimo petlju na dve petlje. U prvoj samo prikupljamo odgovore, a u drugoj obrađujemo rezultate:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
        cout << endl;
    }

    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
        rezultati.push_back( rezultat );
    }

    Izvestaj( zbirPoena, rezultati );
}
```

Sada možemo da premestimo računanje i analizu rezultata iz dela za postavljanje pitanja u deo za ispisivanja izveštaja. Više nema potrebe da se metodu `Izvestaj` prenose podaci o poenima i rezultatima na pojedinačna pitanja:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
        cout << endl;
    }
    Izvestaj();
}

void Izvestaj() const
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
        rezultati.push_back( rezultat );
```

```
        }
        ...
    }
```

U narednom koraku možemo u metodu `Izveštaj` da integrišemo dve petlje u jednu, pri čemu prestaje potreba za nizom `rezultati`:

```
void Izvestaj() const
{
    cout << endl << "Rezultat testiranja:" << endl
         << "-----------------------------------------"
         << endl;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
        cout << (i+1) << ".  "
             << rezultat << endl;
    }
    cout << "-----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Primetimo da smo primenili niz malih transformacija i da je nakon svake od njih program mogao da se prevede i izvršava bez ikakvih promena ponašanja. To su bila mala refaktorisanja na delu.

Korak 7 – Više manjih popravki

Naš metod se zove `Test::PostaviPitanja`, a ustvari on izvršava (pozivanjem drugog metoda) i izračunavanje broja poena i ispisivanje izveštaja o testiranju. Bilo bi mnogo ispravnije da napravimo novi metod `Test::Testiranje`, koji bi pozivao metode `PostaviPitanja` i `Izvestaj`, a da iz metoda `Test::PostaviPitanja` obrišemo pozivanje metoda `Izvestaj`. Istu promenu bismo mogli da objasnimo i na drugi način: najpre menjamo naziv metoda `PostaviPitanja` u `Testiranje`, a zatim iz njega izdvajamo metod `PostaviPitanja`. U svakom slučaju, nakon ove dve izmene dobijamo:

```
void Testiranje()
{
    PostaviPitanja();
    Izvestaj();
}

void PostaviPitanja()
{
    Odgovori_.clear();
```

```
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
            cout << endl;
        }
    }
```

Popravljamo i glavnu funkciju programa tako da koristi metod `Test::Testiranje`:

```
int main( int argc, char** argv )
{
    ...
    test.Testiranje();
}
```

U nekoliko navrata smo već izdvajali kod iz složenog metoda i pravili nove metode, da bismo tako dobili jednostavnije i jasnije metode. Sada imamo pred sobom upravo suprotan slučaj – metod `Test::PostaviPitanje` ne radi dovoljno toga što bi moglo da opravdava njegovo postojanje. Zbog toga ćemo obrisati taj metod i njegovo telo prebaciti u metod `Test::PostaviPitanja`, na jedino mesto u našem programu na kome se taj metod koristi:

```
    void PostaviPitanja()
    {
        Odgovori_.clear();
        for( unsigned i=0; i<Pitanja_.size(); i++ ){
            cout << endl
                 << "*** Pitanje br. " << (i+1) << " ***"
                 << endl << endl;
            Odgovori_.push_back( Pitanja_[i].PostaviPitanje() );
            cout << endl;
        }
    }
```

Korak 8 – Još nekoliko manjih izmena

Pri ocenjivanju pojedinačnih odgovora broj poena se navodi dva puta, najpre kao ceo broj a zatim i u okviru tekstualnog opisa rezultata. Tu redundantnost ne bi trebalo da ostavimo u kodu. Zbog toga je potrebno da se izmeni sadržaj poruka tako da one više ne obuhvataju broj poena. Da bi ponašanje softvera ostalo isto, potrebno je promeniti i način ispisivanja poruka.

Najpre menjamo način izračunavanja rezultata[17]:

[17] Na ovom mestu bi trebalo razmisliti i o eventualnoj upotrebi jednostavne strukture za prenošenje broja poena i tekstualnog opisa ocene pojedinačnih odgovora.

```
    int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const
    {
        if( odgovor == TacanOdgovor() ){
            rezultat = "Tacan odgovor.";
            return 5;
        }else if( odgovor == "@" ){
            rezultat = "Nije odgovoreno.";
            return -1;
        }else{
            rezultat = "Netacan odgovor.";
            return -3;
        }
    }
```

Zatim menjamo i način ispisivanja rezultata u metodu `Test::Izvestaj` tako da najpre ispišemo broj poena (u istom formatu kao ranije) pa tek onda tekstualnu poruku[18]:

```
#include <iomanip>
...

    void Izvestaj() const
    {
        ...
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            string rezultat;
            int brojPoena =
                Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
            zbirPoena += brojPoena;
            cout << (i+1) << ".  ("
                 << setw(2) << brojPoena
                 << " ) " << rezultat << endl;
        }
        cout << "----------------------------------------"
             << endl
             << "Rezultat: " << zbirPoena << " poena" << endl;
    }
...
```

U uvodu smo naveli da imamo u planu da omogućimo da se izveštaji zapisuju u datotekama. Najjednostavniji način uopštavanja ispisivanja izveštaje je da kao

[18] U primeru smo upotrebili objekat `setw(2)`, koji predstavlja tzv. *manipulator izlaznog toka*. Manipulatori služe za upravljanje radom izlaznog toka. Implementiraju se tako što posebna instanca operatora prosleđivanja na osnovu dobijenog manipulatora menja način rada toka. Manipulator `setw` postavlja minimalnu širinu koju će zauzimati naredni podatak koji se pošalje u tok, što je u konkretnom slučaju ceo broj `brojPoena`. . Da bismo mogli da upotrebljavamo manipulatore moramo da uključimo standardnu biblioteku `iomanip`.

dodatni argument odgovarajućih metoda navedemo izlazni tok. Pri tome možemo da navedemo i da je podrazumevan izlazni tok upravo konzolni izlaz, pa nećemo morati da menjamo glavnu funkciju programa:

```
void Testiranje( ostream& ostr = cout )
{
    PostaviPitanja();
    Izvestaj( ostr );
}

void Izvestaj( ostream& ostr ) const
{
    ostr << endl << "Rezultat testiranja:" << endl
         << "-----------------------------------------"
         << endl;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        int brojPoena =
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat );
        zbirPoena += brojPoena;
        ostr << (i+1) << ".  ("
             << setw(2) << brojPoena
             << " ) " << rezultat << endl;
    }
    ostr << "-----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Korak 9 – Nova vrsta pitanja

Sada je vreme da dodamo novu vrstu pitanja. To nije refaktorisanje, već dodavanje novog ponašanja. Pri tome ćemo namerno da novi kod dodamo na prilično loš način, kako bismo posle toga nastavili sa refaktorisanjem. U praksi je mnogo bolje da se pre dodavanja novog koda naprave potrebna refaktorisanja, tako da da planirano dodavanje novog koda ne naruši kvalitet programa.

Nova vrsta pitanja liči na prethodnu – i dalje imamo jedno pitanje i više ponuđenih odgovora. Razlika je u tome što se od korisnika sada ne očekuje da navede tačno jedan od ponuđenih odgovora, već da ih navede sve u odgovarajućem tačnom redosledu. Potpuno tačan odgovor nosi onoliko poena koliko ima ponuđenih odgovora, ili 5 poena ako je broj ponuđenih odgovora manji od 5. Netačan odgovor i dalje nosi -3 poena, a izostavljen odgovor -1 poen. Novina je da odgovor može biti i delimično tačan – ako su bar dva ponuđena odgovora na tačnim mestima, onda odgovor nosi onoliko poena koliko ima odgovora na tačnim mestima.

Najpre ćemo u glavnoj funkciji programa naš test dopuniti novim pitanjem:

```
...
test.Dodaj( Pitanje(
    "Poredjati tipove od najmanjeg do najveceg "
        "po broju bajtova.",
    "badc",
    "a - short",
    "b - char",
    "c - long double",
    "d - long"
));
...
```

Zatim u metodu `Pitanje::PostaviPitanje` menjamo tekst uputstva koje se ispisuje za novu vrstu pitanja:

```
string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl;
    if( TacanOdgovor().size() == 1 )
        cout << "Upisite slovo koje stoji "
                "ispred tacnog odgovora " << endl
             << "ili znak @ ako ne znate: ";
    else
        cout << "Navedite tacan redosled odgovora "
                "upisivanjem slova" << endl
             << "navedenih ispred odgovora (npr: acbd)"
             << endl
             << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Na kraju moramo da promenimo i način ocenjivanja pitanja u metodu `Pitanje::ProveriOdgovor`. Za izračunavanje broja poena u slučaju tačnog odgovora koristimo šablon `max`[19]. Kao i ranije, netačan odgovor nosi -3, a izostavljanje odgovora -1 poen. Međutim, sada moramo da procenimo i vrednost delimično tačnih odgovora:

[19] Iako šabloni funkcija često mogu da se upotrebljavaju bez eksplicitnog navođenja tipa, ovde moramo da navedemo tip zato što konstanta 5 i metod `size()` imaju različite celobrojne tipove. Alternativa je da konstantu zapišemo kao „`5u`“.

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "Tacan odgovor.";
        return max<unsigned>(5,TacanOdgovor().size());
    }else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    }else if( TacanOdgovor().size() > 1 ){
        int poena = 0;
        for( unsigned i=0;
             i<TacanOdgovor().size() && i<odgovor.size();
             i++
           )
           if( TacanOdgovor()[i] == odgovor[i] )
               poena++;
        if( poena>1 ){
            rezultat = "Delimicno tacno.";
            return poena;
        }
    }
    rezultat = "Netacan odgovor.";
    return -3;
}
```

Primetimo da u slučaju nekih prevodilaca (tj. verzija biblioteka) može biti potrebno da eksplicitno dodamo biblioteku `algorithm`, u kojoj je definisan šablon `max`:

```
#include <algorithm>
```

Korak 10 – Priprema za uvođenje hijerarhije klasa pitanja

U prethodnom koraku smo dodali novo ponašanje, ali to nije urađeno na dobar način. Metod `Pitanje::ProveriOdgovor` je postao prilično neuredan, zato što se u jednoj istoj funkciji poeni računaju na različite načine, a u zavisnosti od vrste pitanja. Takođe, i metod `Pitanje::PostaviPitanje` se ponaša različito za različita pitanja.

Uobičajeno sredstvo za razdvajanje ponašanja koje zavisi od neke vrste objekta jeste pravljenje hijerarhije klasa. U našem slučaju možemo da prepoznamo da postoje dve različite vrste pitanja i da napravimo odgovarajuće klase za njih. Štaviše, možemo i da se pripremimo za dodavanje novih vrsta pitanja pravljenjem dovoljno uopštene bazne klase hijerarhije.

Ipak, jedan od prvih problema koje moramo da rešimo je priprema našeg koda za prelazak na rad sa hijerarhijom klasa. Rad sa hijerarhijama klasa i dinamičko razrešavanje metoda prema konkretnim klasama objekata u programskom jeziku C++ zahtevaju upotrebu objekata putem referenci ili pokazivača. Do sada smo testu dodavali automatske objekte pitanja, a sada ćemo morati da ih pravimo dinamički i

da ih dodajemo putem pokazivača. Ta priprema mora da prethodi pravljenju hijerarhije klasa.

Najpre menjamo tip kolekcije pitanja:

```
vector<const Pitanje*>   Pitanja_;
```

Pa metod `Test::Dodaj`:

```
void Dodaj( const Pitanje* p )
{
    Pitanja_.push_back(p);
}
```

Neophodan nam je i destruktor:

```
~Test()
{
    for( unsigned i=0; i<Pitanja_.size(); i++ )
        delete Pitanja_[i];
}
```

Da se ne bismo bavili kopiranjem testova naglasićemo da ne želimo da se implementiraju podrazumevani konstruktor kopije i operator dodeljivanja[20]. Punu ispravnu implementaciju ovih metoda ostavljamo za vežbu. Napominjemo da je potrebno da se posveti posebna pažnja ispravnom kopiranju testova i pojedinačnih pitanja.

```
private:
    Test( const Test& ) = delete;
    Test& operator=( const Test& ) = delete;
```

Iako smo samo zabranili podrazumevani konstruktor kopije, zapravo smo eksplicitno deklarisali *nekakav* konstruktor, što je dovoljno da više ne postoji podrazumevani konstruktor klase `Test`. Zato moramo da ga napišemo:

---

[20] Alternativno, može da se primeni klasičan pristup (koji je ujedno bio i jedini do verzije C++11) da se metodi deklarišu kao privatni i da se ne implementiraju. Na taj način se sprečava njihova eksplicitna ili implicitna upotreba. Eventualna upotreba ovih metoda van klase `Test` proizvela bi grešku pri prevođenju zbog pokušaja upotrebe privatnog metoda. Sa druge strane, u slučaju njihove eventualne upotrebe u okviru same klase `Test` će biti prijavljena greška pri povezivanju, zato što smo metode samo deklarisali a ne i definisali, tj. implementirali.

Od verzije C++11 postoji i posebna sintaksa za „izbacivanje“ podrazumevanih ili nasleđenih metoda, kao i sprečavanje nekih podrazumevanih konverzija – dovoljno je da se na kraju deklaracije metoda navede „`= delete`“.

```
    Test() {}
```

Menjamo i ostale metode klase `Test` u kojima se koriste pitanja:

```
    void PostaviPitanja()
    {
        ...
            Odgovori_.push_back(Pitanja_[i]->PostaviPitanje());
        ...
    }

    void Izvestaj( ostream& ostr ) const
    {
        ...
            int brojPoena = Pitanja_[i]->ProveriOdgovor(
                                        Odgovori_[i], rezultat );
        ...
    }
```

Na kraju, menjamo i način pravljenja testa u glavnoj funkciji programa:

```
int main( int argc, char** argv )
{
    Test test;
    test.Dodaj( new Pitanje(
        ...
    ));
    test.Dodaj( new Pitanje(
        ...
    ));
    test.Dodaj( new Pitanje(
        ...
    ));

    test.Testiranje();
}
```

Korak 11 – Spuštanje klase AbcdRedosled

U prethodnom koraku smo se pripremili za uvođenje hijerarhije klasa pitanja. Nova vrsta pitanja predstavlja uopštenje ranije postojećih pitanja sa više ponuđenih odgovora. Zbog toga najpre izdvajamo novu klasu `AbcdRedosled`, koja će da nasledi postojeću klasu `Pitanje`:

```
class AbcdRedosled : public Pitanje
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
```

```
                    const string& odgovor3 = "",
                    const string& odgovor4 = "",
                    const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}
};
```

Menjamo i glavnu funkciju:

```
int main( int argc, char** argv )
{
    ...
    test.Dodaj( new AbcdRedosled(
        "Poredjati tipove od najmanjeg do najveceg "
            "po broju bajtova.",
        "badc",
        "a - short",
        "b - char",
        "c - long double",
        "d - long"
    ));
    ...
}
```

U ovom trenutku imamo ispravan kod, ali se naša nova klasa još uvek ni po čemu ne razlikuje od stare, a nismo ni popravili kod. Drugim rečima, sve do sada je bila priprema, a sada ćemo razdvojiti ponašanje ovih klasa.

Najpre u klasi `Pitanje` deklarišemo kao virtualne one metode koji (bi trebalo da) se ponašaju različito za različite vrste pitanja i dodajemo virtualni destruktor:

```
    virtual ~Pitanje()
    {}

    virtual string PostaviPitanje() const
    {...}

    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const
    {...}
```

Zatim ih iskopiramo u klasu `AbcdRedosled`[21]:

```
    string PostaviPitanje() const override
    {...}
```

[21] U izvedenoj klasi ne mora da stoji „`virtual`“, mada nije problem ni ako se navede.

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const override
{...}
```

Sada iz metoda `Pitanje::PostaviPitanje` i `Pitanje::ProveriOdgovor` izbacimo viškove i ostavimo samo ono što nam je neophodno u klasi `Pitanje` – tj. vraćamo ih u stanje koje je prethodilo dodavanju nove vrste pitanja:

```
virtual string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}

virtual int ProveriOdgovor( const string& odgovor,
                            string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "Netacan odgovor.";
        return -3;
    }
}
```

Slično uradimo i u klasi `AbcdRedosled` – u njoj nema razloga da stoji ništa osim opisa ponašanja koje je specifično za novu vrstu pitanja:

```
string PostaviPitanje() const override
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
```

```
            << "Navedite tacan redosled odgovora "
               "upisivanjem slova" << endl
            << "navedenih ispred odgovora (npr: acbd)"
            << endl
            << "ili znak @ ako ne znate: ";
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }

    int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const override
    {
        if( odgovor == TacanOdgovor() ){
            rezultat = "Tacan odgovor.";
            return max<unsigned>(5,TacanOdgovor().size());
        }else if( odgovor == "@" ){
            rezultat = "Nije odgovoreno.";
            return -1;
        }else{
            int poena = 0;
            for( unsigned i=0;
                 i<TacanOdgovor().size() && i<odgovor.size();
                 i++
               )
               if( TacanOdgovor()[i] == odgovor[i] )
                   poena++;
            if( poena>1 ){
                rezultat = "Delimicno tacno.";
                return poena;
            }else{
                rezultat = "Netacan odgovor.";
                return -3;
            }
        }
    }
```

Korak 12 – Izdvajanje uopštene bazne klase Pitanje

Klasa `Pitanje` je napravljena prema prvobitno jedinoj vrsti pitanja. Očigledno je da ona sada predstavlja vrlo specifičan vid pitanja, a ne uopšteno pitanje, kako bi naziv klase mogao da sugeriše. Na primer, ako bismo dodavali novu vrstu pitanja na koja se očekuje upisivanje tačnog odgovora (na primer, na pitanje „Izračunaj 2+3“ mora da se upiše tačan odgovor „5“), onda nam za to ne bi bili potrebni ponuđeni odgovori. Zbog toga ćemo sada da uvedemo novu baznu klasu `Pitanje`.

Promenu ćemo da napravimo u dva manja koraka: najpre ćemo da postojećoj klasi `Pitanje` promenimo naziv u `AbcdPitalica`, a zatim i da izdvojimo uopštenu osnovu ove klase u novu klasu `Pitanje`. Pri tome se klasa `Test` praktično ne menja, ali pri pravljenju testa u glavnoj funkciji sada pitanja moraju da se prave kao objekti klase `AbcdPitalica`.

Promena imena klase se izvodi relativno jednostavno. Menjamo ime u definiciji klase, u definicijji izvedene klase `AbcdRedosled` i u glavnoj funkciji programa:

```
class AbcdPitalica
{
public:
    AbcdPitalica( ... )
    ...
};

class AbcdRedosled : public AbcdPitalica
{
public:
    AbcdRedosled( ... )
        : AbcdPitalica( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}
    ...
};
...

int main( int argc, char** argv )
{
    ...
    test.Dodaj( new AbcdPitalica( ... ));
    test.Dodaj( new AbcdPitalica( ... ));
    test.Dodaj( new AbcdPitalica( ... ));
    ...
}
```

Zatim pravimo novu klasu `Pitanje`. Moramo odmah da joj dodamo deklaraciju interfejsa, da bi klasa `Test` mogla da je koristi. Sve ostalo ćemo dodavati kasnije, pa ova klasa na početku može da izgleda ovako:

```
class Pitanje
{
public:
    virtual ~Pitanje() {}
    virtual string PostaviPitanje() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;
};
```

Potrebno je da izmenimo i klasu `AbcdPitalica` tako da nasleđuje klasu `Pitanje`:

```
class AbcdPitalica : public Pitanje
...
```

Korak 13 – Uređivanje odgovornosti klasa hijerarhije

Nakon što smo završili inicijalno pravljenje novih klasa, sada možemo da pristupimo postepenom uređivanju njihovih odgovornosti. Potrebno je da uočimo šta je odgovornost koje klase u hijerarhiji i da odgovrajuće ponašanje podignemo ili spustimo kroz hijerarhiju.

Svako pitanje bi trebalo da ima tekst pitanja i tačan odgovor, pa je odgovarajuće elemente potrebno da premestimo iz klase `AbcdPitalica` u klasu `Pitanje`. Dodajemo i odgovarajući konstruktor i u skladu sa time menjamo konstruktor klase `AbcdPitalica`:

```
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja, const string& tacanOdgovor )
        : _TekstPitanja(tekstPitanja),
          _TacanOdgovor(tacanOdgovor)
        {}

    virtual ~Pitanje() {}
    virtual string PostaviPitanje() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;

    const string& TekstPitanja() const
        { return _TekstPitanja; }
    const string& TacanOdgovor() const
        { return _TacanOdgovor; }

private:
    string _TekstPitanja;
    string _TacanOdgovor;
};

class AbcdPitalica : public Pitanje
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
                  const string& odgovor4 = "",
                  const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor )
    ...
};
```

Postavljanje pitanja je u do obe implementirane konkretne klase veoma slično. Pažljivijim pregledom možemo da uočimo da bi postavljanje pitanja trebalo da

podelimo na tri dela: ispisivanje pitanja, ispisivanje uputstva i čitanje odgovora. Za sada vidimo da može biti razlike u ispisivanju pitanja, već imamo razliku u uputstvu, ali ne vidimo da je potrebna razlika pri čitanju odgovora. U skladu sa time, u klasi `AbcdPitalica` izdvajamo metode `IspisiPitanje` i `Uputstvo`:

```
string PostaviPitanje() const
{
    IspisiPitanje();
    cout << Uputstvo();
    string odgovor;
    cin >> odgovor;
    return odgovor;
}

virtual void IspisiPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl;
}

virtual const char* Uputstvo() const
{
    return  "Upisite slovo koje stoji "
            "ispred tacnog odgovora\n "
            "ili znak @ ako ne znate: ";
}
```

Ako u klasi `AbcdRedosled` implementiramo odgovarajuću verziju metoda `Uputstvo`, onda iz nje možemo da obrišemo posebnu verziju metoda `PostaviPitanje`:

```
const char* Uputstvo() const override
{
    return  "Navedite tacan redosled odgovora "
            "upisivanjem slova\n"
            "koja stoje ispred odgovora (npr: acbd)\n"
            "ili znak @ ako ne znate: ";
}
```

Napisana implementacija metoda `PostaviPitanje` je sada dovoljno uopštena da može da bude u baznoj klasi hijerarhije, pa ovaj metod podižemo uz hijerarhiju. Štaviše, nakon premeštanja on više ne mora da bude virtualan. Umesto toga, u klasi `Pitanje` deklarišemo nove virtualne metode `IspisiPitanje` i `Uputstvo`:

```
class Pitanje
{
public:
    ...
    string PostaviPitanje() const
    {
        IspisiPitanje();
        cout << Uputstvo();
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }

    virtual void IspisiPitanje() const = 0;
    virtual const char* Uputstvo() const = 0;
    ...
};
```

Sledeći korak uopštavanja može da bude premeštanje najopštijeg dela ponašanja iz metoda `AbcdPitalica::IspisiPitanje` u `Pitanje::IspisiPitanje`:

```
class Pitanje
{
public:
    ...
    virtual void IspisiPitanje() const
    {
        cout << TekstPitanja() << endl
             << "----------------------------------------"
             << endl;
    }
    ...
};

class AbcdPitalica : public Pitanje
{
public:
    ...
    void IspisiPitanje() const override
    {
        Pitanje::IspisiPitanje();
        for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
            cout << PonudjeniOdgovori()[j] << endl;
        cout << "----------------------------------------"
             << endl;
    }
    ...
};
```

Rezultat

Sledeći korak bi možda moglo da bude brisanje nekih suvišnih metoda: tekstu pitanja sada pristupamo samo iz klase `Pitanje`, pa možemo da obrišemo odgovarajući pristupni metod i njegovu upotrebu zamenimo eksplicitnim pristupanjem. Slično je i sa nizom ponuđenih odgovora, koji se sada koristi samo u klasi u kojoj je i definisan.

Zatim bi moglo da se pristupi daljem strukturiranju metoda `ProveriOdgovor`. Na primer, metod bi mogao da se uopšti uvođenjem novih virtualnih metoda:

```
    int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const
    {
        if( OdgovorJeTacan(odgovor) )
            return OceniTacanOdgovor(rezultat);
        else if( odgovor == "@" ){
            rezultat = "Nije odgovoreno.";
            return -1;
        else
            return OceniNetacanOdgovor( odgovor, rezultat );
    }
```

U nekim daljim koracima bismo mogli da razdvojimo apstrakciju pitanja od apstrakcije postavljanja pitanja, tako da se kroz testiranje ne menja test nego neki drugi objekat koji opisuje rezultate testiranja.

Ipak, ovde ćemo da zastanemo. Uvek ima i dalje i bolje, ali nekad mora i da se stane. Predstavljeno je dovoljno refaktorisanja da čitalac može da stekne utisak o suštini procesa.

Nakon svih napravljenih izmena program izgleda ovako:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

//-----------------------------------------------------------------
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja, const string& tacanOdgovor )
        : TekstPitanja_( tekstPitanja ),
          TacanOdgovor_( tacanOdgovor )
        {}

    virtual ~Pitanje() {}
```

```
    string PostaviPitanje() const
    {
        IspisiPitanje();
        cout << Uputstvo();
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }

    virtual void IspisiPitanje() const
    {
        cout << TekstPitanja() << endl
             << "----------------------------------------"
             << endl;
    }

    virtual const char* Uputstvo() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;

    const string& TekstPitanja() const
        { return TekstPitanja_; }
    const string& TacanOdgovor() const
        { return TacanOdgovor_; }

private:
    string TekstPitanja_;
    string TacanOdgovor_;
};

//---------------------------------------------------------------
class AbcdPitalica : public Pitanje
{
public:
    AbcdPitalica( const string& tekstPitanja,
              const string& tacanOdgovor,
              const string& odgovor1,
              const string& odgovor2 = "",
              const string& odgovor3 = "",
              const string& odgovor4 = "",
              const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor )
    {
        PonudjeniOdgovori_.push_back( odgovor1 );
        if( odgovor2 != "" )
            PonudjeniOdgovori_.push_back( odgovor2 );
        if( odgovor3 != "" )
            PonudjeniOdgovori_.push_back( odgovor3 );
        if( odgovor4 != "" )
            PonudjeniOdgovori_.push_back( odgovor4 );
        if( odgovor5 != "" )
            PonudjeniOdgovori_.push_back( odgovor5 );
    }
```

```
    const vector<string>& PonudjeniOdgovori() const
        { return PonudjeniOdgovori_; }

    void IspisiPitanje() const override
    {
        Pitanje::IspisiPitanje();
        for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
            cout << PonudjeniOdgovori()[j] << endl;
        cout << "----------------------------------------"
             << endl;
    }

    const char* Uputstvo() const override
    {
        return "Upisite slovo koje stoji "
               "ispred tacnog odgovora \n"
               "ili znak @ ako ne znate: ";
    }

    virtual int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const
    {
        if( odgovor == TacanOdgovor() ) {
            rezultat = "Tacan odgovor.";
            return 5;
        } else if( odgovor == "@" ) {
            rezultat = "Nije odgovoreno.";
            return 1;
        } else {
            rezultat = "Netacan odgovor.";
            return -3;
        }
    }

private:
    vector<string> PonudjeniOdgovori_;
};

//---------------------------------------------------------------
class AbcdRedosled : public AbcdPitalica
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
                  const string& odgovor4 = "",
                  const string& odgovor5 = ""
        )
        : AbcdPitalica( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}
```

```
    const char* Uputstvo() const override
    {
        return  "Navedite tacan redosled odgovora "
                "upisivanjem slova\n"
                "koja stoje ispred odgovora (npr: acbd)\n"
                "ili znak @ ako ne znate: ";
    }

    int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const override
    {
        if( odgovor == TacanOdgovor() ) {
            rezultat = "Tacan odgovor.";
            return max<unsigned>( 5, TacanOdgovor().size() );
        } else if( odgovor == "@" ) {
            rezultat = "Nije odgovoreno.";
            return 1;
        } else {
            int poena = 0;
            for( unsigned i=0;
                 i<TacanOdgovor().size() && i<odgovor.size();
                 i++
                )
                if( TacanOdgovor()[i] == odgovor[i] )
                    poena++;
            if( poena>1 ){
                rezultat = "Delimicno tacno.";
                return poena;
            } else {
                rezultat = "Netacan odgovor.";
                return -3;
            }
        }
    }
};

//---------------------------------------------------------------
class Test
{
public:
    Test()
    {}

    ~Test()
    {
        for( unsigned i=0; i<Pitanja_.size(); i++ )
            delete Pitanja_[i];
    }

    void Dodaj( const Pitanje* p )
        { Pitanja_.push_back(p); }

    void Testiranje( ostream& ostr = cout )
    {
```

```
        PostaviPitanja();
        Izvestaj( ostr );
    }

    void PostaviPitanja()
    {
        Odgovori_.clear();
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            cout << endl
                << "*** Pitanje br. " << (i+1) << " ***"
                << endl << endl;
            Odgovori_.push_back( Pitanja_[i]->PostaviPitanje() );
            cout << endl;
        }
    }

    void Izvestaj( ostream& ostr ) const
    {
        ostr << endl << "Rezultat testiranja:" << endl
             << "-----------------------------------------"
             << endl;
        int zbirPoena = 0;
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            string rezultat;
            int brojPoena = Pitanja_[i]->ProveriOdgovor(
                                        Odgovori_[i], rezultat );
            zbirPoena += brojPoena;
            ostr << (i+1) << ".  ("
                 << setw(2) << brojPoena
                 << " ) " << rezultat << endl;
        }
        ostr << "-----------------------------------------"
             << endl
             << "Rezultat: " << zbirPoena << " poena" << endl;
    }

private:
    Test( const Test& ) = delete;
    Test& operator=( const Test& ) = delete;

    vector<const Pitanje*> Pitanja_;
    vector<string> Odgovori_;
};

//----------------------------------------------------------------
int main( int argc, char** argv )
{
    Test test;
    test.Dodaj( new AbcdPitalica(
        "Sta je to 'std::map'?",
        "b",
        "a - Putokaz do gusarskog blaga.",
        "b - Asocijativni niz u standardnoj biblioteci P.J. C++.",
        "c - Mocvarni teren opasan po zivot."
    ));
```

```
    test.Dodaj( new AbcdPitalica(
        "Koliko tacnih odgovora ima ovo pitanje?",
        "c",
        "a - Tri.",
        "b - Dva.",
        "c - Jedan."
    ));
    test.Dodaj( new AbcdRedosled(
        "Poredjati tipove od najmanjeg do najveceg "
            "po broju bajtova.",
        "badc",
        "a - short",
        "b - char",
        "c - long double",
        "d - long"
    ));

    test.Testiranje();
}
```

Već na prvi pogled programski kod je značajno veći, ali i bolje oblikovan. Često ćemo u praksi imati slučajeve da refaktorisanje povećava obim koda, ali to nikako ne bi smelo da predstavlja prepreku refaktorisanju. Naprotiv, ako je kod duži, ali je čitljiviji, jasniji i bolje strukturiran, onda će se on mnogo lakše održavati, pa dodatna veličina ne predstavlja nikakvu smetnju.