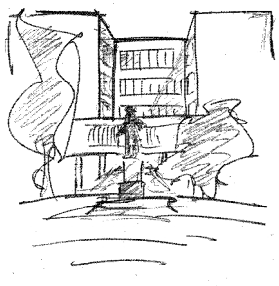
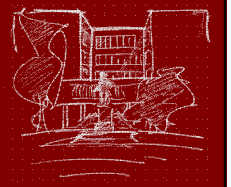
Развој софтвера 12б



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



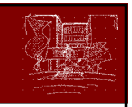
Тема 16

Архитектуре засноване на догађајима

Класичне архитектуре

- Класичан приступ развоју софтвера подразумева имплементирање алгоритама који обављају одређени посао
  - применом метода од врха наниже одређене целине проблема издвајају у потпрограме
  - и даље сви аспекти проблема остају целина
- Пример:
  - Програм који у петљи проверава да ли је корисник притиснуо неки тастер и затим поступа у складу са акцијом корисника

Архитектура заснована на догађајима



- Експлицитна комуникација међу објектима се замењује механизмом емитовања догађаја и дистрибуирања заинтересованим објектима
- Пример:
  - Програм обавештава ОС да је заинтересован да реагује на притисак тастера од стране корисника
  - ОС препознаје догађај и шаље програму поруку
  - Програм (одговарајући објекат) реагује на поруку

Архитектуре засноване на догађајима

Мотивација

- Архитектуре засноване на догађајима спуштају ниво међусобне зависности објеката и тако смањују укупну сложеност система
  - уобичајен је ниво спрегнутости путем параметара или чак путем порука
  - уобичајена је динамичка спрега
    - конфигурисање односа међу објектима у фази извршавања програма
    - конфигурисање односа међу објектима при прављењу објеката
  - смањује се укупан интензитет спрегнутости

Универзитет у Београду - Математички факултет

Архитектуре засноване на догађајима

Увећавање локалне сложености

- Цена смањивања укупне сложености применом архитектура заснованих на догађајима може бити увећавање локалне сложености
  - компоненте које сарађују у систему заснованом на догађајима су једноставније из угла сложености кода
  - њихове операције су дефинисане апстрактније па могу бити теже за разумевање, ако се посматрају локално, без разматрања читавог система

Универзитет у Београду - Математички факултет

Архитектуре засноване на догађајима

Основни појмови

- *Догађај* је препознатљив услов који иницира обавештење
- *Обавештење* је догађајем иницран сигнал који се шаље примаоцу

Универзитет у Београду - Математички факултет

Архитектуре засноване на догађајима

Слојеви тока догађаја

- *Генератор догађаја*
  - објекат који препознаје да је наступио услов који представља неки дефинисан догађај
- *Машина за обраду догађаја*
  - место на коме се препознаје настали догађај и одабире и покреће одговарајућа акција (или низ акција)
- *Канал догађаја*
  - механизам путем кога се догађај прослеђује од генератора до машине за обраду догађаја
- *Низ догађајима управљаних активности*
  - место испољавања догађаја

Универзитет у Београду - Математички факултет

Пример

- Илустроваћемо концепт примером, који ће из класичне форме програма постепено да еволуира до форме која користи програмирање са догађајима

Једноставан програм који ради неки посао и извештава о току посла

```
#include <iostream>
using namespace std;

int main(int, char *)
{
    int zbir = 0;
    for( int i=1; i<=100; i++ ){
        zbir += i;
        cout << ".";
        if( i%10 == 0 )
            cout << "[i=" << i << "]" << endl;
    }
    cout << "Kraj!" << endl;
    cout << "SUM(1,100) = " << zbir << endl;

    return 0;
}
```


Резултат...

```
..........[i=10]
..........[i=20]
..........[i=30]
..........[i=40]
..........[i=50]
..........[i=60]
..........[i=70]
..........[i=80]
..........[i=90]
..........[i=100]
Kraj!
SUM(1,100) = 5050
```


Пример...

- (пример је тривијалан, али ће послужити)
- (да би био јаснији смисао, потребно је замислити да је програм сложенији)
- У претходном примеру главни програм
  - ради посао
  - извештава о току посла
  - извештава о резултату посла
- Прво могуће унапређење је да се извршавање посла одвоји од главног програма

Једноставан програм који ради неки посао и извештава о току посла

```
#include <iostream>
using namespace std;

int uradiPosao( int n1, int n2 )
{
    int zbir = 0;
    for( int i=n1; i<=n2; i++ ){
        zbir += i;
        cout << ".";
        if( i%10 == 0 )
            cout << "[i=" << i << "]" << endl;
    }
    cout << "Kraj!" << endl;
    return zbir;
}

int main(int, char *)
{
    int zbir = uradiPosao(1,100);
    cout << "SUM(1,100) = " << zbir << endl;

    return 0;
}
```


Пример...

- У случају сложених проблема, често је потребно направити класе задатака
  - при прављењу објекта се одређују параметри
  - посебним методом се покреће извршавање
  - посебним методом се приступа резултату

Једноставан програм који ради неки посао и извештава о току посла

```
#include <iostream>
using namespace std;

class Zadatak {
public:
    Zadatak( int n1, int n2 )
        : _N1(n1), _N2(n2), _Rezultat(0)
    {}

    void uradiPosao()
    {
        int zbir = 0;
        for( int i=_N1; i<=_N2; i++ ){
            zbir += i;
            cout << ".";
            if( i%10 == 0 )
                cout << "[i=" << i
                     << "]" << endl;
        }
        cout << "Kraj!" << endl;
        _Rezultat = zbir;
    }

    int Rezultat() const
        { return _Rezultat; }

private:
    int _N1, _N2, _Rezultat;
};
```

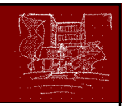
```
int main(int, char *)
{
    Zadatak z(1,100);
    z.uradiPosao();
    cout << "SUM(1,100) = "
         << z.Rezultat() << endl;
    return 0;
}
```


Пример...

- Сложени методи се разлажу на више једноставнијих метода...
- Помоћни послови се издвајају у посебне методе...

Једноставан програм који ради неки посао и извештава о току посла

```
class Zadatak {
    ...

    void uradiPosao()
    {
        int zbir = 0;
        for( int i=_N1; i<=_N2; i++ ){
            zbir += i;
            izvestajKorak();
            if( i%10 == 0 )
                izvestajKorak(i);
        }
        izvestajKraj();
        _Rezultat = zbir;
    }

    void izvestajKorak()
        { cout << "."; }

    void izvestajKorak( int i )
        { cout << "[i=" << i << "]" << endl; }

    void izvestajKraj()
        { cout << "Kraj!" << endl; }

    ...
};
```

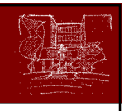



Пример...

- Анализа кохезије класе *Zadatak* показује да су методи за извештавање посебна целина
  - њих други методи употребљавају
  - из њих се не употребљавају други методи
- Штавише, апстраховањем тих метода у посебну класу омогућило би се и потенцијално мењање начина извештавања...

Једноставан програм који ради неки посао и извештава о току посла

```
class Izvestaj {
public:
    void izvestajKorak()
        { cout << "."; }

    void izvestajKorak( int i )
        { cout << "[i=" << i << "]" << endl; }

    void izvestajKraj()
        { cout << "Kraj!" << endl; }
};

class Zadatak {
...
    void uradiPosao()
    {
        Izvestaj izv;
        int zbir = 0;
        for( int i=_N1; i<=_N2; i++ ){
            zbir += i;
            izv.izvestajKorak();
            if( i%10 == 0 )
                izv.izvestajKorak(i);
        }
        izv.izvestajKraj();
        _Rezultat = zbir;
    }
...
};
```

Пример...

- Следећи ниво апстракције је да се статичка спрега имплементације класе *Zadatak* ослаби у спрегу са параметром, која допушта каснију виртуализацију
- Конкретизација спреге се постиже динамички, при позивању метода, а не статички, при његовом писању

Једноставан програм који ради неки посао и извештава о току посла

```
class Zadatak {
...
    void uradiPosao( Izvestaj& izv )
    {
        int zbir = 0;
        for( int i=_N1; i<=_N2; i++ ){
            zbir += i;
            izv.izvestajKorak();
            if( i%10 == 0 )
                izv.izvestajKorak(i);
        }
        izv.izvestajKraj();
        _Rezultat = zbir;
    }
...
};


int main(int, char *)
{
    Zadatak z(1,100);
    Izvestaj izv;
    z.uradiPosao(izv);
    cout << "SUM(1,100) = " << z.Rezultat() << endl;

    return 0;
}
```


Пример...

- Постојећа спрега између класа је на нивоу типова
- Она би могла да се сведе на спрегу спецификација, ако би се извештавање решавало преко шаблона
  - Методи за извештавање могу да буду статички
- Штавише, тако би извештавање могло једноставно и да се искључит, ако није потребно
  - Коришћењем класе за извештавање која не ради ништа
- Остављамо такву трансформацију за вежбу

Пример...

- Уместо тога покушаћемо да даље апстрахујемо однос међу класама применом механизма догађаја:
  - Извршавање задатка генерише догађаје
  - “Неки” објекат за извештавање је дужан да прихвати догађаје

Qt

- Користимо окружење *Qt* за управљање догађајима
- Претпоставке:
  - класа која генерише или прихвата догађаје мора да наследи класу *QObject*
  - на самом почетку декларације класе наводи се макро *Q_OBJECT*

Qt – сигнали

- Методи који производе догађаје називају се *сигнали*
  - декларишу се у оквиру секције *signals*
    - као методи типа *void*
    - не имплементирају се
  - сигнали се *емитују* у облику:
    emit singnal(...)
  - кључна реч *emit* је макро
    - за стандардни C++ има празну дефиницију
    - зато може и да се изостави
    - препоручује се да се наводи
      - *ради компатибилности кода*
      - *ради документовања и лакшег проналажења емитовања*

Qt – слотови

- Методи који обрађују догађаје називају се *слотови*
  - декларишу се у оквиру секције *private slots* или *public slots*
  - имплементирају се као сасвим обичне функције
  - морају бити истог типа као сигнали за чију су обраду намењени

Пример...

- Привремено *прекинемо* спрегу између класа:
  - у класи *Zadatak* декларишемо сигнале
  - у класи *Izvestaj* од постојећих метода направимо слотове

Једноставан програм који ради неки посао и извештава о току посла...

```
// датотека Zadatak.h

#ifndef ZADATAK_H
#define ZADATAK_H

#include <QObject>
#include <iostream>
using namespace std;

class Zadatak : public QObject
{
Q_OBJECT
public:
    Zadatak( int n1, int n2 )
        : _N1(n1), _N2(n2), _Rezultat(0)
    {}

    void uradiPosao();

    int Rezultat() const
        { return _Rezultat; }

signals:
    void izvestajKorak();
    void izvestajKorak( int i );
    void izvestajKraj();

private:
    int _N1, _N2, _Rezultat;
};
#endif // ZADATAK_H
```

```
// датотека Zadatak.cpp

#include "Zadatak.h"

void Zadatak::uradiPosao()
{
    int zbir = 0;
    for( int i=_N1; i<=_N2; i++ ){
        zbir += i;
        emit izvestajKorak();
        if( i%10 == 0 )
            emit izvestajKorak(i);
    }
    emit izvestajKraj();
    _Rezultat = zbir;
}
```

Једноставан програм који ради неки посао и извештава о току посла...

```
// датотека Izvestaj.h

#ifndef IZVESTAJ_H
#define IZVESTAJ_H

#include <QObject>

#include <iostream>
using namespace std;

class Izvestaj : public QObject
{
Q_OBJECT
public slots:
    void izvestajKorak()
        { cout << "."; }

    void izvestajKorak( int i )
        { cout << "[i=" << i << "]"
               << endl; }

    void izvestajKraj()
        { cout << "Kraj!" << endl; }
};

#endif // IZVESTAJ_H
```


Једноставан програм који ради неки посао и извештава о току посла...

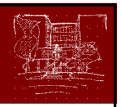
```
// датотека main.cpp

#include <iostream>
using namespace std;

#include "Zadatak.h"

int main(int argc, char **argv)
{
    Zadatak z(1,100);
    z.uradiPosao();
    cout << "SUM(1,100) = "
         << z.Rezultat() << endl;

    return 0;
}
```

- Овако написан програм ради исправно
  - (иако то може да изгледа чудно)
- Само нема извештавања о току извршавања

Qt – повезивање

- Повезивање се остварује помоћу статичког метода *QObject::connect*
  - први аргумент је показивач на објекат који емитује сигнал
  - други аргумент је сигнал који се емитује
    - наводи се помоћу макроа *SIGNAL*
  - трећи аргумент је показивач на објекат који хвата сигнал
  - четврти аргумент је слот којим се сигнал хвата
    - наводи се помоћу макроа *SLOT*
  - слот и одговарајући сигнал морају бити истог типа
    - није важно име
- Зависно од конфигурације, након емитовања сигнала може да се сачека да се заврши његова обрада или да се одмах настави са радом
  - подразумевано понашање је да се сачека са обрадом

Пример...



- У тело функције *main* додајемо конфигурисање реаговања на догађаје

Једноставан програм који ради неки посао и извештава о току посла...

```
#include <iostream>
using namespace std;

#include "Zadatak.h"
#include "Izvestaj.h"

int main(int argc, char **argv)
{
    Zadatak z(1,100);
    Izvestaj izv;

    QObject::connect( &z, SIGNAL(izvestajKorak()),
                      &izv, SLOT(izvestajKorak()));

    QObject::connect( &z, SIGNAL(izvestajKorak(int)),
                      &izv, SLOT(izvestajKorak(int)));

    QObject::connect( &z, SIGNAL(izvestajKraj()),
                      &izv, SLOT(izvestajKraj()));

    z.uradiPosao();
    cout << "SUM(1,100) = " << z.Rezultat() << endl;

    return 0;
}
```


Резултат

- На представљеном примеру се види да је спрега двеју класа спуштена на веома низак ниво
- Класа *Zadatak* емитује сигнале
  - не мора да зна *ништа* о објектима који ће обрађивати сигнале
- Класа *Izvestaj* обрађује сигнале одређеног типа
  - не мора знати скоро ништа о објектима који емитују сигнале
    - не зна којој класи припадају
    - не зна називе сигнала
    - зна само типове сигнала на које реагује
- Спрега се успоставља динамички, на месту које је за то најпогодније – у овом случају у главном програму
  - само је при успостављању спреге потребно имати све информације о сигналима и слотовима

Последице

- Оваква архитектура омогућава да се за многе сложене проблеме направи решење које обезбеђује најнижи могући (познати?) интензитет спреге
- Поред тога, постоје додатни квалитети:
  - ако неки сигнали престану да буду значајни, може да престане њихово праћење, а да компонента која их емитује при томе не мора да се мења
  - ако је неке сигнале потребно обрадити вишеструко, довољно је додати нове слотове који ће их прихватати, а да се компоненте које емитују сигнале, као ни друге компоненте које већ обрађују те сигнале, не морају мењати

Последице на примеру

- Можемо изменити класу *Izvestaj* тако да као параметар добија ток у коме је потребно остваривати извештавање
- Затим можемо различите делове извештаја слати у различите токове
- Исте делове извештаја можемо слати у више токова
- При томе не мењамо ни класу која емитује сигнале ни класу која их прихвата, већ само *динамички* мењамо *конфигурацију* система
  - нпр. у зависности од аргумената командне линије
- Имплементацију остављамо за вежбу

Динамичко повезивање

- Спрега преко сигнала и слотова је *динамичка спрега*
  - објекти који се повезују нису спрегнути у коду којим су дефинисани
  - спрега се успоставља у посебном *конфигурационом* делу програмског кода
- Ипак, успостављање спреге (тј. повезивање слотова и сигнала) је имплементирано на статички начин:
  - Везе се успостављају једнократно
  - Везе између сигнала и слотова се успостављају безусловно

Динамичко повезивање (2)

- Успостављање спреге путем повезивања сигнала и слотова може да буде имплементирано и на динамички начин:
  - Везе се могу постављати у зависности од контекста
  - Током извршавања програма се везе могу раскидати и поново успостављати
    - метод: ***QObject::disconnect***

Динамичко повезивање (3)

- Наредни корак би било динамичко програмско *прављење* слотова и сигнала и затим њихово повезивање
- Библиотека *Qt* нема непосредну подршку за динамичко прављење слотова и сигнала, али постоји више примера додатних класа које то омогућавају
  - пример: *http://doc.qt.digia.com/qq/qq16-dynamicqobject.html*
- Могуће примене су у имплементацији скрипт језика, динамичких окружења за прављење или мењање корисничких интерфејса и сл.

Литература за тему

- *Ted Faison,* ***Event-Based Programming****, Springer, 2006.*
- ***Qt Reference Documentation****, http://qt-project.org/doc/*