

1

Tehnike obrade pogrešaka

2017/18.06

➤ Zaštita programa od neispravnog unosa podataka

- Defenzivna vožnja automobila temelji se na načelu da vozač nikad ne može biti siguran što će učiniti drugi vozači, pa unaprijed nastoji izbjeći nezgodu za slučaj pogreške drugih vozača
- U defenzivnom programiranju ideja vodilja je da će potprogram s neispravnim podacima “opstati” i onda kada su pogreškom pozivajuće procedure predani neispravni argumenti
- Pristup “smeće unutra, smeće van” (“garbage in, garbage out”) treba zamijeniti sa: “smeće unutra, ništa van”, “smeće unutra, poruka o pogrešci van” ili “smeću zabranjen ulaz”

➤ Osnovna pravila kojih se treba držati:

- Provjeriti ispravnost svih vrijednosti podataka iz vanjskih izvora (datoteka, korisnik, mreža, ...)
- Provjeriti ispravnost svih vrijednosti ulaznih parametara
- Odlučiti kako postupiti u slučaju neispravnih podataka

Jednostavan primjer defenzivnog programiranja

3

- ➡ Nedefenzivno programiranje rekurzivne funkcije za računanje faktoriijela

```
int faktorijel( int N )
{
    if ( N == 0 ) return 1;
    else return N * faktorijel( N-1 ) ;
}
```

- ➡ Za negativne vrijednosti nastupa (teoretski) beskonačna rekurzija.
- ➡ Defenzivno programiranje rekurzivne funkcije faktorijel

```
int faktorijel( int N )
{
    if ( N <= 0 ) return 1;
    else return N * faktorijel( N-1 ) ;
}
```

- ➡ Matematički neprecizno, ali sprječava beskonačnu rekurziju u slučaju preljeva

Tehnike obrade pogrešaka

4

➤ Tehnike obrade pogrešaka

- vratiti neutralnu vrijednost (0, "", NULL)
- zamijeniti neispravnu vrijednost sljedećom, moguće ispravnom, npr. while (GPSfix != OK) sleep(1/100s) ...
- vratiti vrijednost vraćenu pri prethodnom pozivu
- zamijeniti neispravnu vrijednost najbližom ispravnom, npr. min(max(kut, 0),360)
- zapisati poruku o pogrešci u datoteku, kombinirano s ostalim tehnikama
- vratiti kôd pogreške kao rezultat, baciti iznimku ili postaviti globalnu statusnu varijablu:
- pozvati "globalnu" metodu za obradu pogreške, npr. perror()
- bezuvjetni završetak programa, npr. Application.Exit (CancelEventArgs)

➤ Robusnost i ispravnost (programa)

- robusnost - u slučaju pogreške omogućen je daljnji rad programa, iako to ponekad znači vratiti neispravan rezultat
- ispravnost - nikad ne vratiti neispravan rezultat, iako to značilo ne vratiti ništa

Iznimke (Exceptions)

5

- Iznimka predstavlja problem ili promjenu stanja koja prekida normalan tijek izvođenja naredbi programa
- U programskom jeziku C#, iznimka je objekt instanciran iz razreda koji nasljeđuje *System.Exception*
- *System.Exception* – osnovni razred za iznimke
 - *StackTrace* – sadrži popis poziva postupaka koji su doveli do pogreške
 - *Message* – sadrži opis pogreške
 - *Source* – sadrži ime aplikacije koja je odgovorna za pogrešku
 - *TargetSite*¹ – sadrži naziv postupka koji je proizveo pogrešku (¹ ne postoji u .NET Coreu)

Neke ugrađene iznimke

6

► Neke sistemske iznimke

ArrayTypeMismatchException	tip vrijednosti koji se pohranjuje u polje je različit od tipa polja i implicitna konverzija se ne može obaviti
DivideByZeroException	pokušaj dijeljenja s 0
IndexOutOfRangeException	indeks polja je izvan deklarirane veličine polja
InvalidCastException	nedozvoljena konverzija tipa
OutOfMemoryException	nedostatak memorije za alociranje objekta
OverflowException	preljev pri izračunavanju aritmetičkog izraza
NullReferenceException	referenci nije pridružen objekt
StackOverflowException	stog je prepunjen

Obrada iznimki

7

- Obrada iznimki sprječava nepredviđeni prekid izvođenja programa
- Iznimka se obrađuju tzv. rukovateljem iznimki (exception handler)
 - Obrada pogreške sastoji se razdvajanju kôda u blokove *try*, *catch* i *finally*

```
try {  
    //dio kôda koji može dovesti do iznimke  
}  
catch (ExceptionType1 exOb) {  
    //kôd koji se obavlja u slučaju iznimke tipa ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    //kôd koji se obavlja u slučaju iznimke tipa ExceptionType2  
}  
...// ostali catch blokovi  
finally {  
    //kôd koji se obavlja nakon izvođenja try, odnosno catch bloka
```

Postupak obrade iznimki

8

➤ Varijante obrade

- Za jedan try blok može postojati jedan ili više catch blokova koji obrađuju različite vrste pogrešaka.
- Kada dođe do pogreške u try bloku, a postoji više catch blokova, obavlja se onaj catch blok koji obrađuje nastali tip iznimke. Ostali catch blokovi neće se obaviti.
- Ako postoji više catch blokova, posljednji se navodi blok koji obrađuje općenite iznimke (tipa *Exception*)

➤ Primjer: Kodiranje \ TryCatch

```
try {  
    int x = new Random().Next(2); //0 ili 1  
    int y = 10 / x;  
    int[] a = { 1, 2, 3 };  
    Console.WriteLine(a[3].ToString());  
}  
catch(DivideByZeroException e) { ... }  
catch(IndexOutOfRangeException e) { ... }  
catch(Exception e) { ... }  
finally { ... }
```


Generiranje (bacanje) iznimke naredbom throw

9

➤ Koristi se naredba throw

- `throw exceptionObject;`

- `npr. throw new Exception("Nova iznimka");`

- `throw;`

- proslijeđivanje iznimke na vanjski blok (rethrow)

➤ Primjer: Kodiranje\Rethrow

- proslijeđivanje uhvaćene iznimke

```
try {  
    MetodaKojaBacaIznimku();  
}  
catch (Exception e) {  
    Console.WriteLine($"Rethrow: {e.Message}\n{e.StackTrace}");  
    throw;  
}
```

Kreiranje vlastitih iznimki

10

➡ Definiranjem razreda izvedenog iz razreda Exception

➡ Izvorna ideja je bila da su sistemske iznimke izvedene iz SystemException, a aplikacijske iz ApplicationException, ali se odustalo od ApplicationException te se vlastite iznimke izvode direktno iz Exception

➡ Primjer:  Kodiranje\CustomException

```
class Iznimka : Exception
{
    private string val;
    public Iznimka() {}
    public Iznimka(string str) : base(str) {
        val = str;
    }
    public override string Message {
        get {
            return "Nije neparan " + val;
        }
    }
}
```

Preporuke za korištenje iznimki

11

- Izbjegavati prazne blokove za hvatanje iznimki (“catch { }”)
- Koristiti specifične iznimke (ne samo osnovne iznimke Exception) znajući što bacaju vlastite knjižnice
- Zapisivati trag bačenih iznimki (log)
- U poruci iznimke uključiti sve informacije o kontekstu nastanka iznimke
- **Razne tipove pogrešaka obrađivati na konzistentan način kroz čitav kod**
- **Razmotriti izradu centraliziranog sustava za dojavu iznimki u kodu**
- Prosljeđivanje iznimki raditi samo kada želimo specijalizirati iznimku
- Koristiti iznimke za obavijest drugim dijelovima programa o pogreškama koje se ne smiju zanemariti
- Bacati iznimke samo u stanjima koja su stvarno iznimna
- Ne bacati iznimke za pogreške koje se mogu obraditi lokalno
- Izbjegavati bacanje iznimki u konstruktorima i finalizatorima, osim ako ih na istom mjestu i ne hvatamo


Unutarnje iznimke

12

➤ `Exception.InnerException` – dobavlja instancu razreda *Exception* koja je izazvala aktualnu iznimku

➤ Primjer:  Kodiranje\InnerException

```
class Primjer {  
    public void F()  
    ...  
    throw new Exception  
        ("Iznimka u Primjer.F() :", e);  
    ...  
class Program  
{  
    static void Main(string[] args){  
        try{ new Primjer().F(); }  
        catch (Exception e)  
        {  
            Console.WriteLine("Iznimka u Main: "  
                + "{0}\nInnerException: {1}",  
                e.Message, e.InnerException.Message);  
        }  
    }  
}
```

- Naredbe kojim se program testira tako da određeni izraz mora biti istinit, a inače se izvršavanje programa zaustavlja
 - koriste se za uklanjanje pogrešaka (debugging) i dokumentiranje ispravnog rada programa
 - koriste se u fazi kodiranja, naročito u razvoju velikih, kompliciranih programa te u razvoju programa od kojih se zahtijeva visoka pouzdanost
- pišemo ih na mjestima gdje se pogreške ne očekuju (tj. ne smiju se pojaviti)
- Naredba `Debug.Assert` s logički izrazom za koji se pretpostavlja (tvrdi) da je istinit
 - Poruka koja se ispisuje ako izraz nije istinit je oblika *Assertion Failed*
 - Automatski se uklanjaju iz *Release* verzije
 - Primjer:  Kodiranje \ Barikade \ Pitagora.cs

```
double Korijen(double broj) {  
    Debug.Assert(broj >= 0, "Broj mora biti nenegativan");  
    double korijen = Math.Sqrt(broj);  
    return korijen;  
}
```

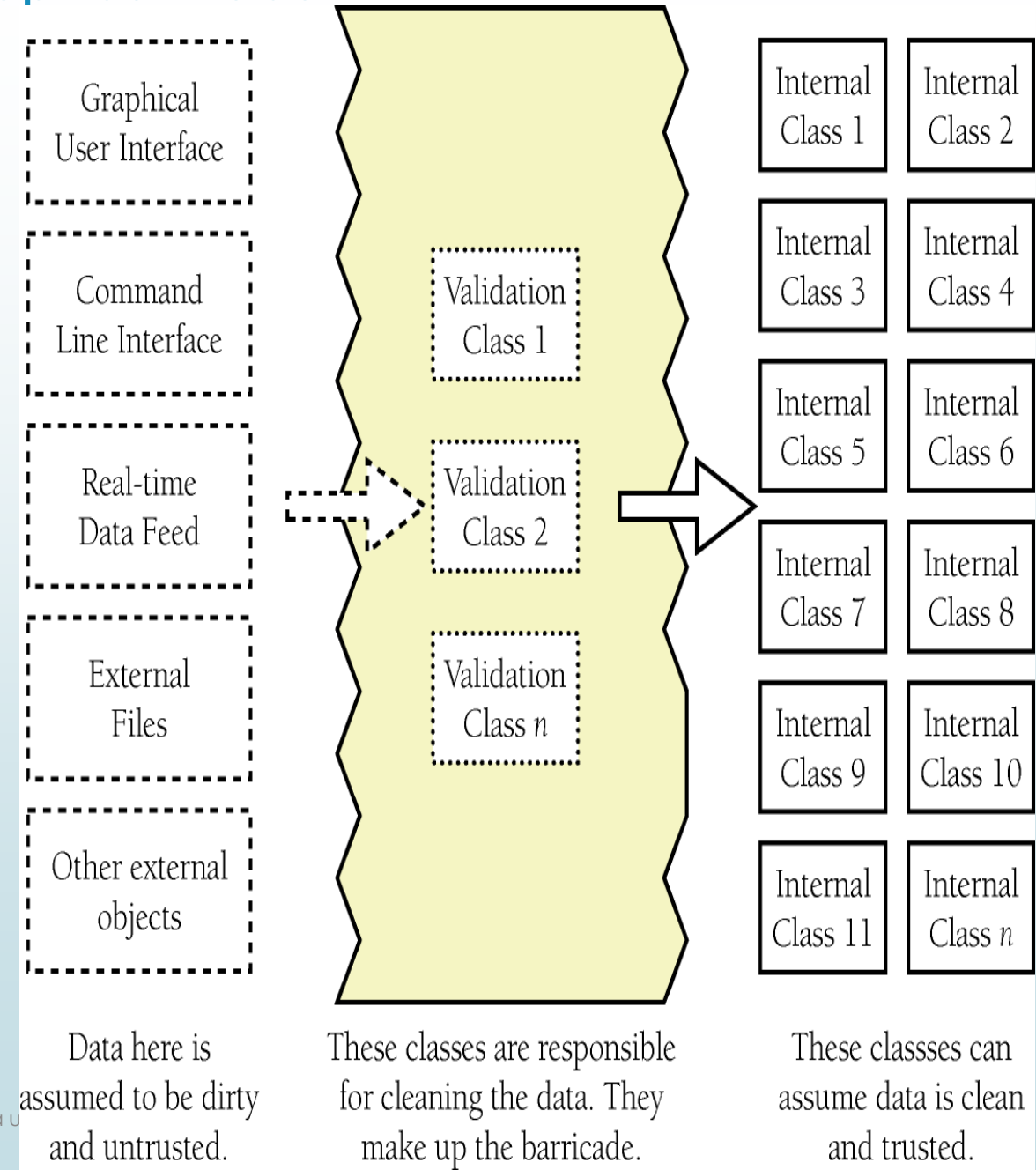
Preporuke za korištenje tvrdnji

14

- Obradu pogreške (iznimke) pisati tamo gdje očekujemo pogreške
- Koristiti tvrdnje tamo gdje nikad ne očekujemo pogreške
- Za jako robustan kod koristiti tvrdnje + kod za obradu pogreške
- Izbjegavati poziv metoda u izrazima tvrdnji
 - npr. `Debug.Assert (Obavi(), "Neobavljeno");`
- Koristiti tvrdnje
 - za dokumentiranje i verificiranje uvjeta koji moraju vrijediti prije pozivanja metode ili instanciranja razreda ("preconditions"), te
 - uvjeta koji moraju vrijediti poslije djelovanja metode ili rada s razredom ("postconditions").

Koncept barikada


- Konstrukcija sučelja kao granica prema “sigurnim” dijelovima koda
- Definiranje dijelova softvera koji će rukovati “prljavim” (nesigurnim) podacima i drugih koji rukuju samo s “čistim” podacima
- Validacijski razredi koji su odgovorni za provjeru ispravnosti podataka sačinjavaju barikadu prema internim razredima koji rukuju s podacima za koje se pretpostavlja da su provjereni i ispravni



```
class Pitagora
{
    private double a;
    public double A { //javno svojstvo (sluzi kao barikada)
        get {
            return a;
        }
        set {
            //provjeravamo je li unutar dozvoljenih vrijednosti
            //ako nije, pridjeljujemo joj neku drugu prikladnu vrijednost
            if (value <= 0) {
                Console.WriteLine($"{value} mora biti veci od 0, postavljam na 1");
                a = 1;
            }
            else {
                a = value;
            }
        }
    }
}
```


Preporuke za korištenje barikada

17

- Barikade naglašavaju razliku između tvrdnji i obrade iznimaka
 - Metode s vanjske strane barikade trebaju koristiti kôd za obradu pogreške
 - Unutarnje metode mogu koristiti tvrdnje jer se ovdje pogreške ne očekuju!
 - Pogledati cijeli primjer  Kodiranje \ Barikade
- Na razini razreda
 - javne metode rukuju s „priljavim” podacima i „čiste” ih
 - privatne metode rukuju samo s “čistim” podacima.
 - pojava „priljavog” podatka u privatnoj metodi nije iznimka koja se očekuje, već neispravnost tvrdnje koja ukazuje na pogrešku u kôdiranju
- Pretvarati podatke u ispravan tip odmah pri unosu


Otkrivanje pogrešaka

18

- Uobičajena zabluda programera je da se ograničenja koja se odnose na konačnu verziju softvera odnose i na razvojnu verziju
 - Treba biti spreman žrtvovati brzinu i resurse tokom razvoja u zamjenu za olakšani razvoj
- Ofenzivno programiranje – učiniti pogreške u fazi razvoja toliko očitim i bolnim da ih je nemoguće zanemariti
 - osigurati da `assert` naredbe uzrokuju prekid izvođenja pri pogrešci
 - popuniti bilo koju alociranu memoriju prije upotrebe radi detektiranja eventualnih problema s njenom alokacijom
 - popuniti alocirane datoteke ili tokove podataka prije upotrebe radi detektiranja eventualnih grešaka u formatu datoteka ili podataka
 - osigurati da svaka `case` naredba koja propagira do `default` slučaja uzrokuje pogrešku koju nije moguće zanemariti
 - napuniti objekt “smećem” (junk data) neposredno prije njegovog brisanja

Otkrivanje pogrešaka (2)


19

- Planirati uklanjanje dijelova programa koji služe kao pomoć u otkrivanju pogrešaka u konačnoj verziji softvera
 - koristiti alate za upravljanje verzijama
 - koristiti ugrađene predprocesore za uključivanje/isključivanje dijelova koda u pojedinoj verziji
 - korištenje vlastitog (samostalno napisanog) predprocesora
 - zamjena metoda za otkrivanje pogrešaka u konačnoj verziji “praznim” metodama koje samo vraćaju kontrolu pozivatelju
- Primjer:  Kodiranje\Conditional

```
#define RAZVOJ //definiramo simbol
...
#if (RAZVOJ)
    // kod za debugiranje
    Console.WriteLine("Poruka UNUTAR koda za debugiranje!");
#endif
...
Console.WriteLine("Poruka IZVAN koda za debugiranje!");
```

Otkrivanje pogrešaka (3)

20

- Umjesto `#if` i `#endif` koristiti *Conditional* (iz *System.Diagnostics*)
- Kod za testiranje odvojiti u posebni postupak i iznad postupka navesti atribut *Conditional*
 - U slučaju da simbol nije definiran, u kompiliranoj verziji nije uključen poziv označenog postupka
 - Simbol se može definirati u kodu, ali i kao parametar prilikom kompiliranja
 - Properties → Build → Conditional compilation symbols
- Primjer:  Kodiranje\Conditional

```
#define RAZVOJ //definiramo simbol
...
Test();
...
[Conditional("RAZVOJ")]
static void Test()
{
    Console.WriteLine("Poruka iz postupka Test");
}
```


Količina defenzivnog koda u završnoj verziji

21

- Ostaviti kôd koji radi provjere na opasne pogreške
- Ukloniti kôd koji provjerava pogreške s trivijalnim posljedicama
 - Ukloniti pretprocesorskim naredbama, a ne fizički
- Ukloniti kôd koji može uzrokovati pad programa
 - U konačnoj verziji treba omogućiti korisnicima da sačuvaju svoj rad prije nego se program sruši.
- Ostaviti kôd koji u slučaju pogreške omogućava “elegantno” rušenje programa
- Ostaviti kôd koji zapisuje pogreške koje se događaju pri izvođenju
 - Zapisivati poruke o pogreškama u datoteku.
- Treba biti siguran da su sve poruke o pogreškama koje softver dojavljuje “prijateljske”
 - Obavijestiti korisnika o “unutarnjoj pogrešci” i navesti e-mail ili broj telefona tako da korisnik ima mogućnost prijaviti pogrešku

Životni vijek objekta

22


- Primjer:  Kodiranje \ Using
- Instancirani objekt postoji dok ga sakupljač smeća (GarbageCollector) ne uništi
 - GC će ga obrisati ako na njega ne pokazuje niti jedna referenca*
- Što ako ne možemo čekati GC?
 - Implementirati sučelje *IDisposable* (postupak *Dispose*)

```
public class Razred : IDisposable {  
    public void Dispose() {  
        //zatvaranje datoteke, konekcije  
        // i sličnih "dragocjenih" resursa  
        ...  
    }  
}
```

- Ako neki razred implementira *IDisposable*, preporuka je da se za objekte tog razreda *Dispose* uvijek pozove nakon što objekt više ne bude potreban.

IDisposable, using blok i iznimke

23

- Dispose se može pozvati eksplicitno
- Što ako se dogodi iznimka prije poziva postupka Dispose?
 - Koristiti tzv. *using blokove*
 - Za objekt stvoren unutar *using bloka*, Dispose se automatski poziva nakon napuštanja bloka (bez obzira na razlog izlaska iz bloka)
 - Primjer:  Kodiranje \ Using

```
Razred r1 = new Razred("A1");  
using (Razred r2 = new Razred("B2")) {  
    Razred r3 = new Razred("C3");  
    throw new ApplicationException("Poruka");  
}  
r1.Dispose();
```

- Using blok se može koristiti samo za one razrede koji implementiraju IDisposable