

Razvoj primijenjene programske potpore

4. Defenzivno programiranje, konfiguracijske datoteke, praćenje traga rada

Defenzivno programiranje

- Zaštita programa od neispravnih podataka, događaja koji se „nikad nisu smjeli dogoditi” i ostalih programerskih pogrešaka
- 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”, odnosno neće izazvati štetu i onda kad su pogrešni podaci posljedica nekog drugog potprograma
 - Pogreške se prvenstveno sprječavaju iterativnim dizajnom, pisanjem pseudo-koda prije kodiranja, pisanjem testova (prije koda), inspekcijom koda, ...
 - Defenzivno programiranje je dodatak na gore navedene tehnike

Smeća na ulazu - smeće na izlazu

- „Garbage IN, Garbage Out” (GIGO) – 1957.?
 - <https://www.newspapers.com/clip/50687334/the-times/>
 - računalo radi ono što mu kažete, iako je to možda krivo
 - frazu popularizirao George Fuechsel (IBM)
 - <https://www.atlasobscura.com/articles/is-this-the-first-time-anyone-printed-garbage-in-garbage-out>

- Charles Babbage

On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to comprehend the kind of confusion of ideas that could provoke such a question.

(1864.)

Smeća na ulazu - ? na izlazu

- Dobar program nikad ne smije proizvesti smeće, neovisno o ulaznim podacima. Pristup „smeće unutra, smeće van” treba zamijeniti sa:
 - „smeće unutra, ništa van”,
 - „smeće unutra, poruka o pogrešci van”
 - „smeću zabranjen ulaz”
 - „počisti smeće” (*turn garbage input into clean input*)
- 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

Tehnike obrade pogrešaka (1)

- vratiti neutralnu vrijednost
 - 0, "", NULL
 - Krivi kod boje? Obojati bijelo?
- zamijeniti neispravnu vrijednost sljedećom, moguće ispravnom
 - npr. while (GPSfix != OK) sleep(1/100s) ...
 - Neispravna adresa dostave? Poslati na adresu za dostavu računa?
- vratiti vrijednost vraćenu pri prethodnom pozivu
- zamijeniti neispravnu vrijednost najbližom ispravnom
 - npr. min(max(kut, 0), 360)
 - Krivo napisana adresa. Uzeti najbližnju? – ne mora nužno biti dobro
- zapisati poruku o pogrešci u datoteku, kombinirano s ostalim tehnikama

Tehnike obrade pogrešaka (2)

- vratiti kôd pogreške
 - pogrešku obrađuje neki drugi dio koda – dojava drugom dijelu
 - mogućnosti ovise o programskom jeziku
 - globalna varijabla
 - „zlouporaba” povratne vrijednosti funkcije
 - iznimke
- pozvati „globalnu” (centraliziranu) metodu za obradu pogreške
- odmah prikazati poruku pogreške
- bezuvjetni završetak programa

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
- Suprotstavljeni pristupi. Što odabrati?
 - Odgovor ovisi o tipu aplikacije i odluci prilikom uspostavljanja arhitekture (dizajna) rješenja
 - Hoće li neispravan rezultat proizvesti npr. zanemarivi artefakt na ekranu ili će npr. krivo izračunati dozu kojom treba ozračiti pacijenta na rendgenu?
 - Što se događa kada tramvaj izgubi vezu ili GPS signal i ne zna na kojoj se stanici približava?

Iznimke

- 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` – sadrži naziv postupka koji je proizveo pogrešku
 - `InnerException` – unutarnja (omotana) iznimka
- `NullReferenceException`, `IndexOutOfRangeException`, `DivideByZeroException`, `InvalidCastException`, ...

Obrada iznimki

- Obrada iznimki sprječava nepredviđeni prekid izvođenja programa
 - Iznimka se obrađuju tzv. rukovateljem iznimki (exception handler)
 - Obrada pogreške sastoji se try, 1 ili više catch blokova i finally bloka
 - catch ili finally se mogu ispustiti (ali samo jedan od njih)

```
try {  
    //dio kôda koji može dovesti do iznimke  
}  
catch (ExcepType1 ex0b){  
    //kôd koji se obavlja u slučaju iznimke tipa ExcepType1  
}  
catch (ExcepType2 ex0b) {  
    //kôd koji se obavlja u slučaju iznimke tipa ExcepType2  
}  
...// ostali catch blokovi  
finally {  
    //kôd koji se uvijek obavlja
```

Preporuke za korištenje iznimki (1)

- Koristiti iznimke za obavijest drugim dijelovima programa o pogreškama koje se ne smiju zanemariti
- Bacati iznimke samo u stanjima koja su stvarno iznimna
 - Ulazne podatke provjeriti što je moguće prije
 - Ne bacati iznimke za pogreške koje se mogu obraditi lokalno
- Izbjegavati bacanje iznimki u konstruktorima i finalizatorima (destruktorima), osim ako ih na istom mjestu i ne hvatamo
 - Problematično npr. u C++-u, iako nije neuobičajeno imati iznimke u konstruktorima u jezicima sa skupljačem smeća (npr. C# ili Java)
- Izbjegavati prazne blokove za hvatanje iznimki - `catch { }`
 - Iznimke ne ignorirati – ako ništa bolje ne možemo, onda barem evidentirati


Preporuke za korišćenje iznimki (2)

- Razne tipove pogrešaka obrađivati na konzistentan način kroz čitav kod
 - Razmotriti izradu centraliziranog sustava za dojavu iznimki u kodu
 - Zapisivati trag bačenih iznimki (log)
- Koristiti specifične iznimke (ne samo osnovne iznimke *Exception*) znajući što bacaju vlastite knjižnice
 - U poruci iznimke uključiti sve informacije o kontekstu nastanka iznimke
 - Odrediti ispravan nivo apstrakcije iznimke (na razini sučelja, a ne implementacije)
 - Originalna iznimka se može omotati

Preporuke za korištenje iznimki (3)

- Hvatati specifične iznimke umjesto Exception
 - možda „uhvatimo” nešto što nismo trebali
 - više catch blokova ako zahtjeva različito ponašanje
- Očistiti resurse u *finally* bloku ili koristiti podržane konstrukcije koje će to automatski obaviti
 - Ne bacati iznimke u *finally* bloku
 - try-with-resources u Javi
 - *using* u C#-u - vidi sljedeći slajd
- Razmotriti alternative (rukovanju) iznimkama

Životni vijek objekta – čišćenje resursa

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


```
public class C : IDisposable {  
    public void Dispose() {  
        //zatvaranje datoteke, veze prema bazi  
        // 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

- *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)

```
C a1 = new C("A1");
using (C b2 = new C("B2")) {
    C c3 = new C("C3");
    throw new Exception("Poruka");
}
a1.Dispose();
```

- *using blok* se može koristiti samo za one razrede koji implementiraju *IDisposable*
 - Koncept sličan try-with-resources u Javi. Pogledati IL kod s ILDASM
Primjer:  DefensiveProgramming \ Using


Tvrdnje

- Naredba kojom se program testira tako da određeni izraz mora biti istinit, a inače se izvršavanje programa zaustavlja
 - Obično se sastoji od izraza koji treba provjeriti i poruke koja se prikazuje ako izraz nije istinit
- Tvrdnje se koriste 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
 - Uklanjanju se iz produkcijske verzije
- **Pišemo ih na mjestima gdje se pogreške ne očekuju (tj. ne smiju se pojaviti)**
 - Ako je tvrdnja istinita, onda znači da program radi kako je zamišljeno

Tvrdnjama provjeravamo i dokumentiramo pretpostavke

- Provjeravamo „nemoguće” uvjete
 - Jesu li parametri za koje se smatra da su ispravni zaista ispravni?
 - Npr. računamo li opseg trokuta za nešto što nije trokut
 - npr. stranice duljine 2, 3, 5
 - Je li primljena referenca zaista valjana (ne *null*) kao što je trebala biti po dokumentaciji
- Detektiraju pogreške nastale negdje drugdje u kodu, kao posljedicu krivih pretpostavki i interpretacija ili naknadnih modifikacija koda
- Tvrdnje ujedno mogu služiti i kao dokumentacija

Tvrdnje u C#-u

- Prostor imena System.Diagnostics, naredba Debug.Assert
 - Logički izraz za koji se pretpostavlja (tvrdi) da je istinit i poruka koja se ispisuje ako izraz nije istinit
 - U .NET Core-u neistinit uvjet uzrokuje trenutno završavanje programa (bez izvođenja eventualnog *finally* bloka)
 - Automatski se uklanjaju iz Release verzije
- Primjer:  DefensiveProgramming \ Barricades \ Course.cs

```
public double AverageGrade() {  
    int sum = 0;  
    foreach (var pair in grades) {  
        int grade = pair.Value;  
        Debug.Assert(grade >= 1 && grade <= 5,  
            $"Invalid grade in dictionary {pair.Key} =  
                {pair.Value}. This should never happens!");  
        sum += grade;  
    }...
```

Preporuke za korištenje tvrdnji

- Obradu pogreške (iznimke) pisati tamo gdje očekujemo pogreške
 - Iznimke su informacije o pogrešnom uvjetu ili neočekivanom događaju
 - Mogu se uhvatiti
- Koristiti tvrdnje tamo gdje nikad ne očekujemo pogreške
 - Predstavljaju fatalne pogreške, ne mogu se uhvatiti i ukazuju na logičku pogrešku u programskom kodu
- Za jako robustan kod koristiti tvrdnje + kod za obradu pogreške čak i ako je pogreška posljedica neispravne pretpostavke
- 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”).
- Izbjegavati poziv metoda u izrazima tvrdnji
 - npr. `Debug.Assert (Obavi(), "Neobavljeno");`
 - u produkciji (s isključenim tvrdnjama) *Obavi* se neće nikada izvršiti

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



Data here is assumed to be dirty and untrusted.

These classes are responsible for cleaning the data. They make up the barricade.

These classes can assume data is clean and trusted.

Koncept barikada

- Primjer:  DefensiveProgramming \ Barricades \ Course.cs

```
public class Course {  
    private Dictionary<string, int> grades =  
        new Dictionary<string, int>();  
    public int this[string name] {  
        get {  
            //what if there is no student's name in the dictionary?  
            bool exists = grades.TryGetValue(name, out int grade);  
            return exists ? grade : -1;  
        }  
        set {  
            if (value < 1 || value > 5) {  
                throw new ArgumentOutOfRangeException(  
                    $"Invalid grade {value}. It should be from 1 to 5");  
            }  
            grades[name] = value;  
        }  
    }  
    ...  
}
```


Preporuke za korišćenje barikada

- 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!
- Na razini razreda
 - javne metode rukuju s „prljavim” podacima i „čiste” ih
 - Ne misle se nužno na *public* metode, već one koje služe za unos podataka
 - privatne metode rukuju samo s “čistim” podacima.
 - pojava „prljavog” 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? Baciti iznimku?

Otkrivanje pogrešaka

- Uobičajena zablude 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)

- 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:  DefensiveProgramming\Conditional

```
#define DEMO //definiramo simbol
...

#if (DEMO)
    Console.WriteLine("Print something..."); //kod za debugiranje
#endif
```

Otkrivanje pogrešaka (3)

- 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:  DefensiveProgramming\Conditional

```
...  
CheckSomething(); //ne izvodi se ako DEMO nije definiran  
...  
[Conditional("DEMO")]  
static void CheckSomething() {  
    ...  
}
```


Količina defenzivnog koda u završnoj verziji

- Previše defenzivnog koda može usporiti izvršavanje i povećati složenosti.
 - 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

Checklist: Defensive Programming u Steve McConnell, Code Complete, 2nd Edition (str. 211 i 212)

CHECKLIST: Defensive Programming

General

- ☐ Does the routine protect itself from bad input data?
- ☐ Have you used assertions to document assumptions, including preconditions and postconditions?
- ☐ Have assertions been used only to document conditions that should never occur?
- ☐ Does the architecture or high-level design specify a specific set of error-handling techniques?
- ☐ Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- ☐ Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- ☐ Have debugging aids been used in the code?
- ☐ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ☐ Is the amount of defensive programming code appropriate—neither too much nor too little?
- ☐ Have you used offensive-programming techniques to make errors difficult to overlook during development?

Exceptions

- ☐ Has your project defined a standardized approach to exception handling?
- ☐ Have you considered alternatives to using an exception?
- ☐ Is the error handled locally rather than throwing a nonlocal exception, if possible?
- ☐ Does the code avoid throwing exceptions in constructors and destructors?
- ☐ Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- ☐ Does each exception include all relevant exception background information?
- ☐ Is the code free of empty *catch* blocks? (Or if an empty *catch* block truly is appropriate, is it documented?)


Security Issues

- ☐ Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, HTML injection, integer overflows, and other malicious inputs?
- ☐ Are all error-return codes checked?
- ☐ Are all exceptions caught?
- ☐ Do error messages avoid providing information that would help an attacker break into the system?

Postavke (konfiguracijski parametri) aplikacije


- Postavke (konfiguracijske parametre) aplikacije za koje očekujemo da će se mijenjati definirati izvan programa
 - kao varijable okruženja
 - unutar konfiguracijskih datoteka
- U protivnom promjena postavki zahtijeva ponovnu izgradnju programa (nova verzija?) i predstavlja potencijalni sigurnosni problem (npr. ispis u tragu stoga kod iznimke)

Variable okruženja (1)

- Variable okruženje definirane operacijskim sustavom, okruženjem u oblaku ili datotekom *launchSettings.json*
 - Nastane ručno ili kroz Visual Studio
 - desni klik na projekt Debug → Enviroment Variables
- Primjer:  ... / Secrets / Properties / launchSettings.json


```
{
  "profiles": {
    "Secrets": {
      "commandName": "Project",
      "environmentVariables": {
        "CustomEnvValue": "Demo"
      }
    }
  }
}
```

Variable okruženja (2)

- Vrijednost varijable okruženja moguće dobiti s
Environment.EnvironmentVariable
- Primjer dohvata svih varijabli okruženja
 DefensiveProgramming / Secrets / Program.cs


```
var enumerator = Environment
    .GetEnvironmentVariables()
    .GetEnumerator();
while(enumerator.MoveNext()) {
    Console.WriteLine(
        $"{enumerator.Key} = {enumerator.Value}");
}
```

Konfiguracijske datoteke (1)

- U .NET Core-u uobičajeno appsettings.json
 - može i nešto drugo (ne mora biti u JSON formatu)
 - U web-aplikacijama će automatski biti uključena u konfiguracijske objekte, u konzolnoj aplikaciji je treba eksplicitno uključiti
- Primjer:  DefensiveProgramming / Secrets / appsettings.json
 - Desni klik → Copy To Output Directory : Copy if newer

```
{
  "Name": "Bob",
  "Demo": {
    "Key0": 123,
    "Key1": "something that should not be publicly available"
  }
}
```

Konfiguracijske datoteke (2)

- Konfiguracijske datoteke se mogu kombinirati (i s varijablama okruženja) u zajednički objekt tipa *IConfiguration* pri čemu je bitan redoslijed uključivanja.
 - svaki novi izvor nadjačava identične parametre
 - Po potrebi koristiti parametar *optional*
- Primjer:  DefensiveProgramming / Secrets / Program.cs

```
IConfiguration configuration = new ConfigurationBuilder()  
    .AddJsonFile("appsettings.json")  
    .AddJsonFile("missing_one.json", optional: true)  
    .AddEnvironmentVariables()  
    ...  
    .Build();  
Console.WriteLine($"Name = " + configuration["Name"]);  
Console.WriteLine($"Key0 = " + configuration["Demo:Key0"]);
```

Osjetljivi podaci u konfiguracijskim datotekama

- Što ako prilikom razvoja treba javno dijeliti izvorni kod s osjetljivim podacima u konfiguracijskim datotekama?
- Mehanizam „korisničkih tajni” (engl. user secrets)
 - U projektnim datotekama stoji ključ (<userSecretsId>), a vrijednost spremljena u korisnikovom profilu
 - Windows:
%APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json
 - Mac & Linux:
~/ .microsoft/usersecrets/<userSecretsId>/secrets.json
 - **Samo za razvoj! Datoteke nisu kriptirane**
 - Prilikom kontinuirane isporuke osmisli mehanizam zamjene šifre s pravom šifrom na produkciji, jer tajna datoteka možda ne postoji na serveru
- Alternativa: Odvojiti osjetljive podatke u posebne datoteke koje neće biti dio repozitorija

Postavljanje datoteke s „tajnim” vrijednostima

- Izmijeniti projektnu datoteku tako da sadrži *UserSecretsId* i dodati odgovarajuće pakete (*Microsoft.Extensions.Configuration.UserSecrets*)

- Primjer:  DefensiveProgramming / Secrets / Secrets.csproj

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0</TargetFramework>
  <UserSecretsId>RPPP-Secrets</UserSecretsId>
</PropertyGroup>
```

- U naredbenom retku u mapi projekta pokrenuti
`dotnet user-secrets set "Demo:Key1" "some secret val"`
 - Stvara .../RPPP-Secrets/secrets.json s odgovarajućim ključ
- Moguće i kroz Visual Studio opcijom *Manage User Secrets*
 - Ako *UserSecretsId* nije postavljen, Visual Studio automatski generira jedinstvenu vrijednost

Uključivanje tajnih vrijednosti

- Primjer:  DefensiveProgramming / Secrets / Program.cs

```
IConfiguration configuration = new ConfigurationBuilder()  
    .AddJsonFile("appsettings.json")  
    .AddEnvironmentVariables()  
    .AddUserSecrets("RPPP-Secrets")  
    .Build();
```

Praćenje traga rada aplikacije

- Motivacija
 - Evidentirati pogreške tijekom rada (neovisno o dojavama korisnika)
 - Detektirati uska grla aplikacije
 - Prikupljanje ostalih informacija
 - npr. parametri pretrage, prijave korisnika
 - ...
- Gdje evidentirati?
 - Baza podataka, Datoteka, E-mail poruke, SMS, Event Log na Windowsima, ...
- Kako evidentirati?
 - Na uniforman (centraliziran) način neovisan o odredištu traga

Razine važnosti zapisa (1)

- Trace
 - Informacija namijenjena programeru u cilju lakšeg rješavanja problema
 - npr. ispis svih parametara nekog postupka
 - Bilježi male korake izvođenja programa
 - Nije preporučljivo koristiti u produkciji
 - smije sadržavati osjetljive podatke
 - veća količina zapisa u odnosu na ostale razine
- Debug
 - Informativna poruka u cilju otklanjanja pogrešaka
 - ispis određene vrijednosti kratkoročne koristi
 - Slično kao Trace, ali rjeđe (i manje detaljno)
 - nije nužno namijenjena samo programeru
 - Najčešće automatski isključeno iz produkcijske verzije

Razine važnosti zapisa (2)

- Information
 - Zapis trajnijeg karaktera koji služi za praćenje toka rada aplikacije
 - npr. informacija o posjetu određenoj stranici ili evidencija postavljenih kriterija pretrage
- Warning
 - Za pogreške koje ne utječu na daljnji rad aplikacije, ali predstavljaju potencijalno opasne situacije te zahtijevaju naknadnu pažnju
 - npr. konfiguracijska datoteka ne sadrži traženu vrijednost, pa se koristi predodređena

Razine važnosti zapisa (3)

- Error
 - Služi za evidentiranje pogrešaka i iznimki koje se ne mogu obraditi
 - Predstavljaju kritičnu pogrešku za određeni postupak, ali ne i za cijelu aplikaciju
 - npr. pogreška prilikom dodavanja novog podatka u bazu
- Critical (Fatal)
 - Situacije koje uzrokuju prekid rada cijele aplikacije
 - npr. nedostatak prostora na disku, neispravne postavke za spajanje na bazu podataka, ...
- Trace < Debug < Information < Warning < Error < Critical
- Ponekad se kao nivo navodi i *None*
 - Trace < Debug < Information < Warning < Error < Critical < None

Praćenje traga u .NET Coreu

- Microsoft.Extensions.Logging.Abstractions
 - Podrška za različite alate i odredišta zapisivanja tragova
 - Nekoliko ugrađenih implementacija
 - Npr. konzolni ispis (Microsoft.Extensions.Logging.Console)
 - Moguće dodati vlastitu implementaciju
- Sučelje *ILogger* kao apstrakcija nad različitim implementacijama
 - Može poslužiti za konzistentan i centralizirani način obrade iznimke


Sučelje ILogger

- `IsEnabled(LogLevel logLevel)`
 - provjerava evidentira li konkretna implementacija zapise navedene razine
- `void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception exception, Func<TState, Exception, string> formatter)`
 - evidentiranje zapisa određene razine i vrste događaja
 - Zapis (state) ne mora nužno biti string, već može biti bilo kojeg tipa
 - Vrsta događaja opisana strukturom `EventId`: Id i naziv
 - Uz zapis (state) može biti vezana određena iznimka (exception)
 - *formatter*: funkcija koja kreira string na osnovi zapisa i iznimke
- `IDisposable BeginScope<TState>(TState state)`
 - Služi za grupiranje više zapisa u jedan zajednički zapis
 - samo ako konkretna implementacija podržava

Pokrate za postupak Log

- Statički razred *Microsoft.Extensions.Logging.LoggerExtensions* s proširenjima za sučelje *ILogger*
- Nekoliko preopterećenih postupaka:
 - *LogTrace*
 - *LogDebug*
 - *LogInformation*
 - *LogError*
 - *LogCritical*

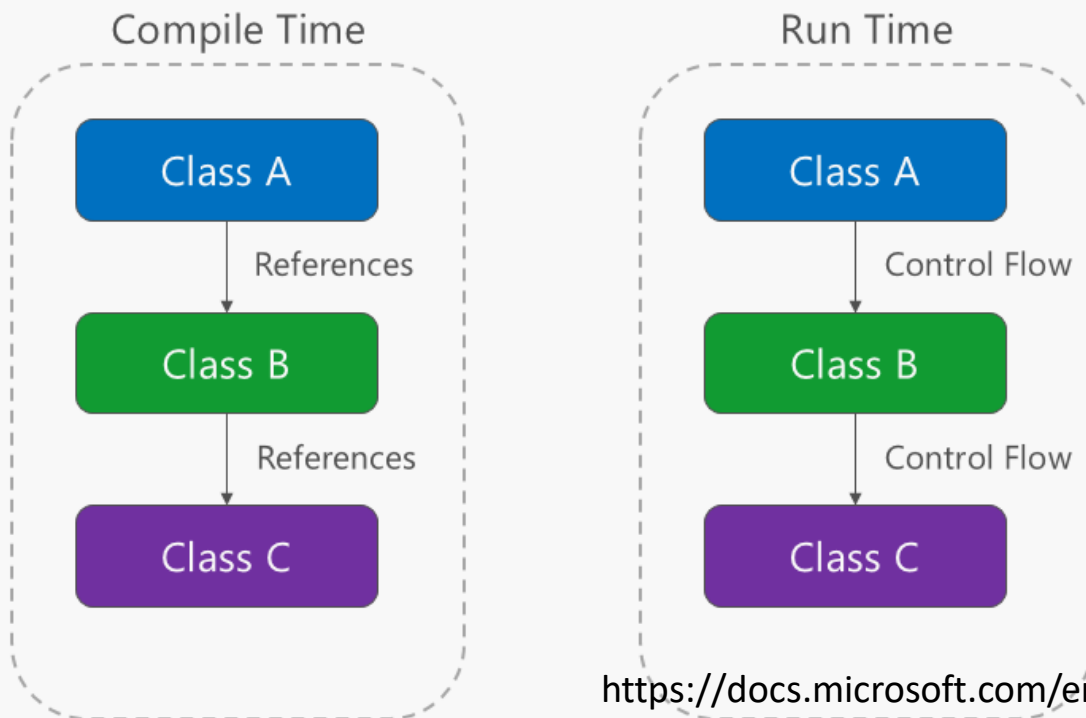
Primjer za praćenje traga i obradu iznimki

- Primjer:  DefensiveProgramming \ Logging
 - Sučelje *IDataLoader* definirano s jednom metodom za učitavanje parova (osoba, datum rođenja) iz određene datoteke
 - Implementacija sučelja (razred *DataLoader*) prilikom čitanja datoteke želi obraditi pogreške na konzistentan i centraliziran način što npr. *ILogger* omogućava.
 - Kako stvoriti objekt tipa *DataLoader* uz što manje kopčanja (ovisnosti) i omogućiti naknadne promjene
 - Konkretna odredišta se postavljaju prilikom inicijalizacije programa
 - Koristi se tehnika *Dependency Injection*
 - Konkretno u ovom slučaju *ConstructorInjection*
 - *DataLoader* kao argument konstruktora prima referencu tipa *ILogger*

Uobičajeni način definiranja ovisnosti

- Ovisnost komponenti odgovara redoslijedu korištenja

Direct Dependency Graph



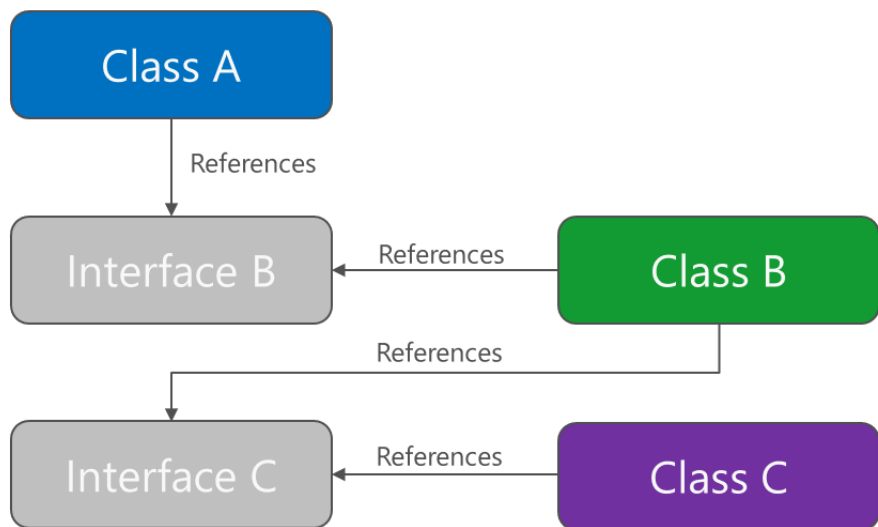
<https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion>

Dependency Inversion

- Ovisnost o sučelju, a ne o konkretnoj implementaciji
 - Bude povezano (umetnuto) naknadno (*Dependency injection*)

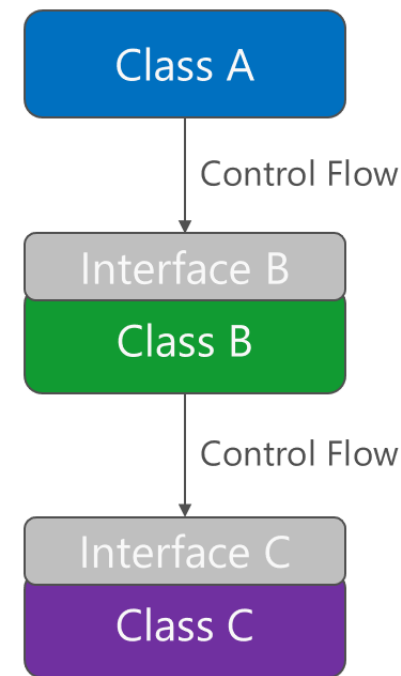
Inverted Dependency Graph

Compile Time




<https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion>

Run Time



Postavljanje lanca ovisnosti


- Za pojedino sučelje ili razred definira se implementacija koja se koristiti kad stvaranje objekta ovisi o tom sučelju (razredu)
 - Primjer  DefensiveProgramming \ Logging \ Program.cs

```
IServiceCollection services = new ServiceCollection();  
var provider = services.AddLogging(...) //definira za ILogger  
                        .AddTransient<IDataLoader, DataLoader>()  
                        .BuildServiceProvider();
```

- Add[Transient|Singleton|Scope] određuju način stvaranja objekta
 - Transient = svaki put stvori novi primjerak
- Novi objekti se, umjesto s new stvaraju koristeći *GetRequiredService*
 - Uzrokuje kreiranje potrebnih objekata u lancu ovisnosti


```
var dataLoader = serviceProvider.GetRequiredService<IDataLoader>();
```

Postavljanje minimalne razine praćenja traga (1)

- Minimalna razina praćenja traga određuje hoće li neka poruka uopće biti prosljeđena povezanim *loggerima*
 - Trace < Debug < Information < Warning < Error < Critical < None
 - Pojedini *logger* može imati i zasebnu dodatnu konfiguraciju (za poruke koje uopće stignu do njega)
- Razina se može postaviti u konfiguracijskoj datoteci
 - Primjer  DefensiveProgramming \ Logging \ appsettings.json


```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Trace"  
    }  
  }  
}
```

Postavljanje minimalne razine praćenja traga (2)

- Prilikom podešavanja lanca ovisnosti postavke minimalne razine traga pročitane iz konfiguracijske datoteke
 - Primjer  DefensiveProgramming \ Logging \ Program.cs


```
var configuration = new ConfigurationBuilder()  
    .SetBasePath(Directory.GetCurrentDirectory())  
    .AddJsonFile("appsettings.json")  
    .Build();  
  
IServiceCollection services = new ServiceCollection();  
var provider = services.AddLogging(configure => {  
    configure.AddConfiguration(  
        configuration.GetSection("Logging"));  
    ...  
})  
    .BuildServiceProvider();
```

NLog

- NLog kao jedan od alata za praćenje traga kompatibilan s Microsoft.Extensions.Logging
 - <http://nlog-project.org>
- Omogućava više vrsta (različitih) spremišta i formata ovisno o razini zapisa
- Uključuje se prilikom konfiguriranja praćenja traga
 - Primjer:  DefensiveProgramming \ Logging \ Program.cs

```
IServiceCollection services = new ServiceCollection();  
var provider = services.AddLogging(configure => {  
    configure.AddConfiguration(  
        configuration.GetSection("Logging"));  
    configure.AddConsole();  
    configure.AddNLog(new  
        NLogProviderOptions { RemoveLoggerFactoryFilter = false });  
}).BuildServiceProvider();
```


Konfiguracijska datoteka za NLog

- Moguće definirati više različitih odredišta, a zatim pravila koja određuju gdje će sve pojedini zapis biti evidentiran
 - Primjer:  DefensiveProgramming \ Logging \ nlog.config

```
<targets>
  <target xsi:type="File" name="allfile"
    fileName="logs\nlog-all-`${shortdate}.log"
    layout="`${longdate}|${event-
properties:item=EventId_Id:whenEmpty=0}|${logger}|${uppercase:${leve
l}}|${message} ${exception}" />
  <target xsi:type="File" name="ownFile"... />
  <target xsi:type="Null" name="blackhole" />
</targets>
<rules>
  <logger name="*" minlevel="Trace" writeTo="allfile" />
  <logger name="Microsoft.*" minlevel="Trace" writeTo="blackhole"
    final="true" />
  <logger name="*" minlevel="Trace" writeTo="ownFile" />
</rules>
```