

|   |    |
|---|----|
| 1. Standardy vývoje .....                   | 2  |
| 1.1 ASP.NET Coding Standards .....          | 2  |
| 1.1.1 ASP.NET Blazor Coding Standards ..... | 3  |
| 1.2 C# Coding Standards .....               | 4  |
| 1.3 Code Analysis & Analyzers .....         | 12 |
| 1.4 Console Application .....               | 15 |
| 1.5 CSS Coding Standards .....              | 16 |
| 1.6 GIT Workflow Standard .....             | 16 |
| 1.7 Source Code Management (SCM) .....      | 22 |
| 1.8 SQL Coding Standards .....              | 22 |
| 1.9 Vizualní standardy .....                | 25 |

# Standardy vývoje

## Navigace

- [Podpora v ReSharperu](#)
- [Názvové konvence](#)
  - [Konzistentní pojmenování](#)
  - [Množiny dat množným číslem](#)
  - [Akce se slovesem](#)
- [Normalizace hodnot](#)
  - [Procenta](#)

## Samostatné stránky

- [ASP.NET Coding Standards](#)
  - [ASP.NET Blazor Coding Standards](#)
  - [ASP.NET MVC Coding Standards](#)
  - [ASP.NET WebForms Coding Standards](#)
- [C# Coding Standards](#)
- [Code Analysis & Analyzers](#)
- [Console Application](#)
- [CSS Coding Standards](#)
- [GIT Workflow Standard](#)
- [\[Obsolete\] StyleCop](#)
- [Source Code Management \(SCM\)](#)
- [SQL Coding Standards](#)
- [Vizuální standardy](#)

## Podpora v ReSharperu

Link [ReSharper](#) - naše nastavení

## Názvové konvence

### Konzistentní pojmenování

Ve všech vrstvách systému používáme konzistentní pojmenování. Není přípustné, aby se v databázi jmenoval objekt Auto, v Services byl Car a v prezentační vrstvě Vehicle-List.aspx.

### Množiny dat množným číslem

Množiny dat pojmenováváme vždy množným číslem, popř. se suffixem vyjadřujícím hromadnost, - Cars, CarCollection, InvoiceList, CarSet, ...

Výjimkou jsou databázové tabulky, pro které používáme jednotné číslo (diskuzi se nebráníme, ale aktuální volba je takováto).

### Akce se slovesem

Je-li něco akce (funkce, metoda, storka, ...) měla by hlavní složka názvu začínat slovesem v rozkazovacím tvaru.

## Normalizace hodnot

### Procenta

Procenta ukládáme a v kódu zpracováváme zásadně jako desetinné číslo (100% jako 1.0), stovkou násobíme až při prezentaci do UI. Ostatně i .NET format-string {0:p} funguje stejně.

## ASP.NET Coding Standards

- [ASP.NET Blazor Coding Standards](#)
- [ASP.NET MVC Coding Standards](#)
- [ASP.NET WebForms Coding Standards](#)

## ASP.NET Blazor Coding Standards

- [Naming Conventions](#)
  - [Standard component parameters](#)
  - [Event callbacks](#)
  - [Event callback handlers](#)
  - [Event-propagation service](#)
- [Coding Standards](#)
  - [Parameter, Inject](#)
  - [Component-class members order](#)

### Naming Conventions

#### Standard component parameters

- `CssClass` - derivatives e.g. `FooterCssClass`
- `Enabled` - not `IsEnabled`, nor `Disabled`
- `Visible` - not `IsVisible`, `Hidden`, or anything similar

#### Event callbacks

- action callbacks should be prefixed with `On`, e.g. `OnClick`, `OnClosed`, `OnUpdating`, `OnSelected`
- we prefer past perfect form for post-action events - e.g. `OnSelected`, `OnClosed`, `OnShown`
  - there are some well-known established exceptions - `OnClick`, `OnSubmit`, etc.
  - user present continuous form for pre-action events - e.g. `OnUpdating`, `OnSorting`, etc.
- `[Parameter]-changed` callbacks follow the `{ParameterName}Changed` [rule for binding](#)
- special callbacks usually do not follow the `On-`prefix pattern, e.g. `DataProvider`, etc.

#### Event callback handlers

- regular callback handlers should follow the `HandleCallbackName` pattern, e.g. `HandleItemSelected`, `HandleValidSubmit`, `HandleValueChanged`
- handlers of special callbacks can have their specific names, e.g. `DataProvider="LoadInvoices"`, `DataProvider="GetSuggestions"`

#### Event-propagation service

- `Dispatcher` suffix, e.g. `ClientCardChangedDispatcher`
- (Web.Client service responsible for signaling an event from it's origin to subscribers.)

### Coding Standards

#### Parameter, Inject

- attribute should be on the same line as the property definition (especially for single-line auto-implemented properties)
- `[Inject]` properties should be `protected`

```
[Inject] protected IMessenger Messenger { get; set; }
[Parameter] public RenderFragment ChildContent { get; set; }
```

#### Component-class members order

1. `[Parameter]`
2. `[Inject]`
3. private fields
4. constructor
5. methods - To improve readability, we want to respect component life-cycle, i.e. follow the order of execution when possible.
  - a. `OnInitialized`
  - b. `OnParametersSet`
  - c. `BuildRenderTree`

- d. OnAfterRender
- e. Dispose
- 6. static methods

## C# Coding Standards

### Navigace

- [Základní zásady](#)
  - [Pravidlo nula](#)
  - [Framework Design Guidelines](#)
  - [Automatická kontrola některých pravidel \(StyleCop, Code Analysis\)](#)
  - [Přípustnost odchylek od pravidel](#)
- [Naming Guidelines](#)
  - [Framework Design Guidelines](#)
  - [Obecné požadavky na pojmenování](#)
  - [Angličtina/čeština v pojmenovávání](#)
  - [Konzistentní pojmenování](#)
  - [Testy a testovací metody](#)
  - [Business Layer](#)
- [Design Standards](#)
  - [Dependency Injection](#)
- [Coding Standards](#)
  - [Formátování kódu](#)
    - [Odsazujeme pomocí tabelátorů \(indent\)](#)
    - [Pořadí členů ve třídě](#)
    - [Použití #region \[Obsolete\("Regiony již nepoužívat, resp. pouze explicitní."\)\]](#)
    - [Závorkování bloků příkazů {}](#)
    - [Závorkování výrazů, zejména logických podmínek](#)
    - [Method Chaining - object.DoSomething\(\).DoSomethingElse\(\).DoAnotherThing\(\)](#)
  - [Komentáře](#)
    - [XML dokumentace \(+ Contracts\)](#)
    - [Výjimka: Nechceme komentáře, které nepřidávají žádnou informační hodnotu](#)
    - [Zakomentované bloky kódu jsou zpravidla nežádoucí](#)
  - [Enums](#)
    - [flagsValue.HasFlag\(flag\)](#)
  - [Entity Framework](#)
    - [Preferujeme strong-typed konstrukce před literals](#)
    - [Fluent API vs. Data Annotations](#)
  - ["Legacy" BusinessLayer\[Generator\] ;-\)](#)
    - [IEnumerable<T> jako návratový typ](#)
  - [Nedoporučované/zakázané konstrukce](#)
    - [Ternární operátor](#)
    - [Konstrukce ala C++](#)
    - [Zkracování StringBuilderu změnou délky](#)
  - [Unit-testing](#)

### Základní zásady

Nesnažíme se o originalitu za každou cenu, ale zejména o srozumitelnost, předvídatelnost a konzistentní podobu. Jde nám o dobře udržovatelný kód, se kterým se dobře pracuje, je snadné se k němu vrátit i po delší době, nebo jej kdykoliv předat druhému.

#### Pravidlo nula

1. Otevři-li libovolný náš zdrojový kód, měl by z hlediska standardů kódování a dodržování konvencí vypadat tak, jak bych ho sám napsal.
2. Pokud mám pocit, že bych něco dělal jinak, navrhnu změnu pravidel (např. osobně, prostřednictvím Yammeru, komentáře zde na Wiki, atp.).
3. Pravidla nejsou nedotknutelná. Pokud mám dobrý důvod je porušit, mohu to udělat, musím být však schopen si to obhájit a měl bych to nejlépe v kódu vysvětlit komentářem.

### Framework Design Guidelines

Dodržujeme zejména všeobecně uznávané guidelines obsažené v MSDN:

- [MSDN: Framework Design Guidelines](#) (dříve "Design Guidelines for Developing Class Libraries")
  - [Naming Guidelines](#)
    - [Capitalization Conventions](#)
    - [General Naming Conventions](#)

- Names of Assemblies and DLLs
- Names of Namespaces
- Names of Classes, Structs, and Interfaces
- Names of Type Members
- Naming Parameters
- Naming Resources
- Type Design Guidelines
  - Choosing Between Class and Struct
  - Abstract Class Design
  - Static Class Design
  - Interface Design
  - Struct Design
  - Enum Design
  - Nested Types
- Member Design Guidelines
  - Member Overloading
  - Property Design
  - Constructor Design
  - Event Design
  - Field Design
  - Extension Methods
  - Operator Overloads
  - Parameter Design
- Designing for Extensibility
  - Unsealed Classes
  - Protected Members
  - Events and Callbacks
  - Virtual Members
  - Abstractions (Abstract Types and Interfaces)
  - Base Classes for Implementing Abstractions
  - Sealing
- Design Guidelines for Exceptions
  - Exception Throwing
  - Using Standard Exception Types
  - Exceptions and Performance
- Usage Guidelines
  - Arrays
  - Attributes
  - Collections
  - Serialization
  - System.Xml Usage
  - Equality Operators
- Common Design Patterns

## Automatická kontrola některých pravidel (StyleCop, Code Analysis)

Mnohá pravidla jsou na projektech kontrolována automaticky pomocí StyleCop a CodeAnalysis (FxCop).

Pro StyleCop používáme OPT-OUT přístup - vycházeli jsme ze základní sady pravidel StyleCopu a vypnuli ta, která se nám nehodila. Pro Code Analysis používáme OPT-IN přístup.

## Přípustnost odchylek od pravidel

V praxi nastanou situace, kdy zdravý rozum nebo lokální důvody velí odchylku od našich pravidel. Takové odchylky jsou samozřejmě přípustné. Kdo se pro ně rozhodne, musí být však schopen takovou odchylku argumentačně uhájit a navíc by prakticky vždy měly být důvody pro takovou odchylku dokumentovány (obvykle formou komentáře v kódu),

## Naming Guidelines

### Framework Design Guidelines

Základem všech názvových konvencí je tomu věnovaná sekce ve [Framework Design Guidelines](#):

#### Naming Guidelines

- Capitalization Conventions
  - UPDATE 31.8.2017 - privátní fieldy pojmenováváme camelCase (bez podtržítka, vč. backing-fields)
- General Naming Conventions
- Names of Assemblies and DLLs
- Names of Namespaces

- [Names of Classes, Structs, and Interfaces](#)
- [Names of Type Members](#)
- [Naming Parameters](#)
- [Naming Resources](#)

## Obecné požadavky na pojmenování

- pojmenování každého elementu musí být srozumitelné, vystihující jeho funkci, ideálně značně samovysvětlující
- pojmenování musí být odhadnutelné, maximálně využíváme ustálených spojení a obecných zvyklostí
- pojmenování musí být konzistentní
  - pokud se stejná funkčnost používá opakovaně, měla by se pokaždé jmenovat stejně, i na různých projektech, např. `Role`
  - pro stejnou věc by se nemelo používat více označení - pokud se business-objekt jmenuje `Faktura`, potom je špatně pojmenovat metodu `GetInvoice()`
- pojmenování nesmí být v konfliktu/zaměnitelné s konvencemi a existujícími třídami .NET Frameworku i našich Framework Extensions
  - např. `Contract`, ani `System` není vhodný název business-trždy
  - ani `IPhone` není ideální
- připomínám, že názvy metod musí začínat imperativní (rozkazovací) formou slovesa, např.
  - `PosliXy()` nikoliv `PoslatXy()`
  - `SendXy()` nikoliv `SendingXy()`
- pokud možno, vyhýbáme se pomnožným slovům v pojmenování tříd, abychom si neznemožnili jasné odlišení kolekcí množným číslem
  - např. `Zarizeni` není vhodný název, protože `Zakaznik.Zarizeni` nevystihuje, jde-li o jedno zařízení, nebo o kolekci
- vyhýbáme se zkratkám, pokud se nejedná o naprosto zřejmé významy
  - správně `RenderHtml()`, `QueryStringID` (pozor výjimka: na rozdíl od obecných konvencí dáváme přednost `ID` před `Id`)
  - špatně `PosliNotifikaciNoveOP()`, správně `PosliNotifikaciNoveObchodniPrilezitosti()`
  - nebojíme se dlouhých názvů, pokud je to pro dosažení srozumitelnosti potřebné a nepřinese to nepřehlednost při použití

## Angličtina/čeština v pojmenovávání

- pojmenování musí být v českém jazyce bez diakritiky, nebo v anglickém jazyce, nebo v přiměřené kombinaci (s respektováním ustálených zvyklostí, např. `GetFaktury()`)
  - dáváme přednost srozumitelnosti a snadné čitelnosti
  - určitě je lepší správně česky, než blbě anglicky
  - nebojíme se pozdějšího refaktoringu
- kde je to zúčastněným vývojářům srozumitelné, dáváme přednost angličtině
  - nemá však smysl jít za každou cenu do angličtiny, pokud jde o line-of-business aplikaci, kde členové týmu jsou rádi, že chápou české výrazy
- projekty s mezinárodním prvkem (např. nadnárodní zákazník) anglicky
  - stejnětak pokud zákazník používá anglické výrazy
- v ideálním případě se výrazy používané v aplikaci shodují s terminologií zákazníka (i co se jazyka týče)

## Konzistentní pojmenování

- označení jednoho prvku musí být v celém systému konzistentní
- nepřípustné jsou kolize typu

```
public Faktura GetInvoice(...);
DateInfo.CountryID odkazující do tabulky Stat
```

- pokud chceme změnit názvosloví, musíme udělat refactoring, jinak musíme respektovat i nevhodné pojmenování

## Testy a testovací metody

- **Projekt** s testy vždy končí suffixem `Tests` (název projekty i assembly)
  - Preferujeme test-project per project a suffix oddělujeme v takovém případě tečkou, tj. např. k `Havit.Goran.BusinessLayer` máme `Havit.Goran.BusinessLayer.Tests`.
  - Pokud je projekt jediný, jmenuje se obvykle přímo `Tests`, např. `Havit.HealthGuard.Tests` (Podle všeho však paralelní spouštění testů ve Visual Studiu dává každé větvi samostatnou assembly, jediným projektem tak např. bráníme paralelizaci.)
- **Namespaces** v testovacím projektu obvykle odpovídají namespaces testovaného, např. `Havit.Goran.BusinessLayer.Tests.Finance`.

- Vzhledem k toolingu (Generate Unit Test ve VS2015) je povoleno umísťování testů do namespace, které se shoduje s namespace testovaného (avšak samozřejmě v samostatném projektu).
- Pokud je jeden testovací projekt pro více testovaných projektů, je první úrovní v testovacím projektu rozdělení dle názvů testovaných projektů, např. `Havit.HealthGuard.Tests.Web.Controllers`.
- **Třídy** v testovaném projektu obvykle názvem odpovídají názvům testovaných tříd + suffix `Tests`, např. `Havit.Goran.BusinessLayerTests.Timesheety.TimesheetItemTests`.
  - Pokud test pokrývá více tříd, název odpovídá testované oblasti, mnohdy přímo namespace, např. `Havit.Goran.Tests.Services.WorklogTimesheetsExtraction.WorklogTimesheetsExtractionTests`.
  - Pokud jsou testy rozsáhlejší, lze je rozdělit do více tříd-souborů suffixem takto: `TimesheetItemTests_AreaXy`, `TimesheetItemTests_BlaBla`
- Testovací **metody** pojmenováváme podle schématu `<TestovanáOblast>_[TestovanáMetoda_]<TestovanáFunkčnost>`
  - testovanou oblastí je v případě unit testu je obvykle třída testované metody
  - má-li testovaná služba jedinou metodu nebo hlavní entry-point, není nutné ho v názvu testu uvádět (prostřední část)
  - např. `SoftDelete_IsDeletedShouldBeSetAfterRemove`, `EventReceiverService_SubmitEvent_ShouldAddEventToRepository`, `BaseRepository_AddAndGetByID`
  - neplatí zde běžné konvence (sloveso na začátku názvu metody, atp.)
  - důležité je přehledné a srozumitelné zobrazení v testovacím toolingu (Test Explorer atp.)
  - netlačil bych za každou do názvu testovací metody ani slovo `Test`, leda by se tam hodilo pro lepší čitelnost

## Business Layer

Názvové konvence a specifické coding-standards pro Business Layer jsou na [samostatné stránce](#).

## Design Standards

### Dependency Injection

Kde tomu nebrání zdravý rozum, preferujeme [Dependency Injection](#), a to i na projektech, které dosud podle zásad DI uspořádány nebyly.

Jako výchozí volbu používáme DI Container [Castle Windsor](#).

O soužití s Business Layerem viz též [Unit testy - testování Business Layeru](#), inspirací může být projekt 080.XWT.

Ke studiu doporučuji knihu [Mark Seeman: Dependency Injection](#) in .NET, kterou máme do firmy zakoupenou. Elektronická verze (byla v ceně) je zde: `\\TOPOL.havit.local\\Projekty\\000.HAV - HAVIT Provoz\\Training\\Dependency_Injection.pdf` (+ ePub).

## Coding Standards

### Formátování kódu

### Odsazujeme pomocí tabulátorů (indent)

~~Odsazování provádíme zásadně tabulátory – jeden tabulátor pro jednu úroveň odsazení. Odsazování tabulátory (*Keep Tabs*) zajišťuje sdílené týmové nastavení Visual Studia.~~

~~Do Visual Studia doporučujeme *Fix Mixed Tabs* extension jako součást [Productivity Power Tools](#).~~

### Pořadí členů ve třídě

Ve třídě umísťujeme členy v logickém pořadí:

1. [konstanty a statické property](#)
2. [instanční property - private fieldy, které nesou hodnotu property jsou bezprostředně za property](#)
3. [private fieldy - nepříslušející přímo property](#)
4. [instanční constructory](#)
5. [instanční metody – v pořadí dle životního cyklu třídy, pokud lze aplikovat](#)
6. [instanční destructor](#)
7. [statický constructor](#)
8. [statické metody](#)

Pořadí memberů se pro každý explicitně implementovaný interface uplatňuje samostatně.

Class members explicitně implementující jeden interface musí být pohromadě, nelze promíchat pořadí s běžnými metodami nebo ostatními explicitně implementovanými interface (tj. musí být nejdříve members vlastní třídy, potom members pro `ISerializable`, potom pro `IXxx`, atd.).

[Dodržování tohoto pravidla částečně kontrolují naše custom StyleCop rules:](#)

- Kontroluje se pořadí class memberů: Vlastosti, konstruktory, metody. Nekontrolují se fieldy a metody začínající na *On*.
- Kontrolují se explicitní implementace interface, aby se nepromíchávali s "běžnými" members nebo navzájem.

## Použití #region [Obsolete("Regiony již nepoužívat, resp. pouze explicitní.")]

#regiony již používejme jen pro situace, kdy chceme explicitně seskupit nějakou haldu kódu, která k sobě kontextově patří. Např. mnoho přetížení jedné metody, nebo mnoho handlerů událostí jednoho prvku, atp.

Původní náš specifický způsob využití regionů pro strukturování kódu, již je přežitý, pere se s aktuálními nástroji pro generování/údržbu kódu a místo něj lze použít nástroje pro přehledné zobrazení struktury otevřeného souboru, např.

- [Resharper - File Structure](#)
- [CodeMaid - Spade](#)
- [VS10x CodeMAP](#)

Je dovoleno odstraňovat z našich stávajících projektů old-style #regiony (nikoliv ty explicitní), pokud na konkrétním projektu nerozhodne dev-lead jinak.

~~Každý člen třídy musí být součástí alespoň jednoho regionu, typicky regionu obalujícího právě jeden member s názvem tohoto memberu. Regiony musí poskytovat přehledné členění kódu. Název regionu se statickými členy musí končit „(static)“, pokud není celá třída static. Pro obalení metody regionem s názvem této metody je možné použít command *Surround with named region* z [HAWIT Visual Studio Extensions](#).~~

Doporučené regiony:

- ~~Private fields~~
- ~~Constructors~~
- ~~Destructor~~
- ~~Static constructor~~

~~Regiony nemusí být v plném rozsahu použity v třídách WebForms/WinForms, pokud to není na újmou přehlednosti (v případě WebForms tolerujeme bez regionů code-behinds třídy, které jsou celé zobrazitelné na běžném monitoru bez scrollování).~~

~~Regiony není vhodné používat uvnitř metod. Žádná metoda by neměla být tak dlouhá, aby potřebovala region.~~

~~Dodržování tohoto pravidla je na projektech, kde je uplatňováno, kontrolováno pomocí našeho custom [\[Obsolete\] StyleCop rule](#).~~

## Závorkování bloků příkazů {}

Veškeré vnořené příkazy uzavíráme do bloků {}, byť by se jednalo o jediný příkaz. To se týká zejména statementů if, for, foreach, atp. Závorky umísťujeme zásadně na nový řádek, jak závorku otevírací, tak uzavírací (výchozí nastavení automatického formátování Visual Studio).

```
// správn
if (cond)
{
    DoSomething();
}

// špatn
if (cond) DoSomething();
if (cond)
    DoSomething();
```

## Závorkování výrazů, zejména logických podmínek

Ve výrazech závorkami nešetříme a závorkujeme i tam, kde jinak platí precedence operátorů. Zejména v podmínkách uzavíráme každý primitivní výraz do samostatné závorky, aby byla podmínka snadno čitelná.

```
// správn
if ((x == y) || ((a >= b) && !isReady))
{
```



```

    return ((c == d) || ((a * b) + 6 > 100));
}

// špatn
if (x == y || a >= b && (!isReady))
{
    return c == d || a * b + 6 > 100;
}

```

## Method Chaining - object.DoSomething().DoSomethingElse().DoAnotherThing()

- Method-chaining používáme, pouze pokud takové použití bylo zamýšleno autory API (Fluent APIs, LINQ, Moq, ...)
- Method-chainingu se jinak vyhýbáme, pokud se nejedná o primitivní operace, jejichž řetězení zpřehlední kód (např. `text.Replace(...).ToLower().Trim()`)
- Method-chaining nikdy nepoužíváme v kombinaci s constructorem (např. `new MyClass().DoSomething()`), pokud to není nedílná součást nějakého obskurního Fluent API

## Komentáře

### XML dokumentace (+ Contracts)

C# XML komentáře jsou povinné na všech entitách, které jsou určeny k použití někým dalším než jenom jejich autorem (publikovaný interface).

Zejména (avšak nikoliv výhradně) jsou povinné na

- na všech public a protected members BusinessLayer (tříd, properties, metod, událostí, atp.)
- na všech public members WebBase a Services
- na všech public members controls z projektu Web (např. `SubjektPicker`, atp.)
- na všech members HAVIT Framework Extensions (nutnost dokumentace bez výjimky). Komentáře jsou obvykle větou, na jejímž konci píšeme tečku.

Na stejných prvcích jsou povinné i contracts

- minimálně kontrola vstupních argumentů metod, pokud to má smysl (`Contract.Requires`, nebo `if-then-throw`)
- pokud má projekt Code Contracts, potom použít ty, jinak `if-then-throw`

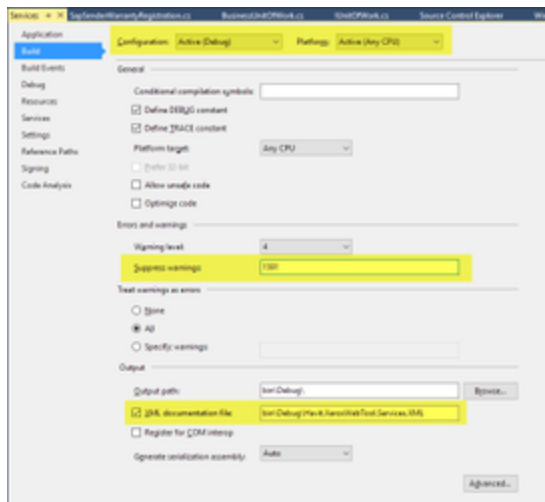
### Výjimka: Nechceme komentáře, které nepřidávají žádnou informační hodnotu



#### Výjimka z pravidla

Nechceme komentáře ala generovaný GhostDoc, které nepřinášejí k signatuře metody žádnou další informační hodnotu. Každý je dostatečně inteligentní a odpovědný, aby se rozhodl, zdali je signatura metody natolik vševysvětlující, že si může dovolit komentář vynechat.

Pokud je na projektu zapnuté generování XML souboru s komentáři a chceme přesto povolit tuto výjimku "odpovědného nekomentování", lze warning *"Missing XML comment for publicly visible type or member"* vypnout v konfiguraci projektu, na záložce Build, volba *Suppress warnings*, hodnota 1591. Je potřeba to udělat pro všechny relevantní konfigurace.



## Zakomentované bloky kódu jsou zpravidla nežádoucí

- zdrojové kódy nejsou archiv minulých myšlenek, od toho slouží SCM
- zakomentované části můžeme v kódu ponechat pouze tehdy, pokud je zjevné, že budou brzo znovu potřeba, nebo pokud to významně přispěje k orientaci/pochopení aktivního kódu
  - vždy v takových případech musí být opatřeny zřetelným vysvětlením, např.:

```
DoSomething();

// kontrola vstupních podmínek na žádost zákazníka doasn
// deaktivována (F1235)
// ValidateCosi(...);

DoSomething();
```

## Enums

### flagsValue.HasFlag(flag)

V .NET 4+ použijeme kvůli přehlednosti zásadně metodu HasFlag pro zjišťování flagů:

```
// správn
if (row.RowState.HasFlag(DataControlRowState.Insert))
{
    ...
}

// obsolete
if ((row.RowState & DataControlRowState.Insert) == DataControlRowState.
Insert)
{
    ...
}
```

## Preferujeme strong-typed konstrukce před literals

Pokud lze stejnou věc zapsat pomocí strong-type konstrukce i literalu, preferujeme strong-type verzi (kontrola, refactoring, ...)

```
// správn
context.Employees.Include(e => e.LoginAccount).FirstOrDefault();
// špatn
context.Employees.Include("LoginAccount").FirstOrDefault();
```

## Fluent API vs. Data Annotations

Pro mapování entity na DB preferujeme Fluent API. [Data Annotations atributy](#) jsou přípustné pouze pro meta-data, která popisují entitu nezávisle na její perzistenci do DB.

Příklady Data Annotations atributů, které popisují POCO entitu/property nezávisle na způsobu uložení (ač mohou být jinak nevhodné):

- Required
- CreditCard
- EmailAddress
- MinLength, MaxLength, StringLength
- Phone
- Range
- Url

Data Annotations používáme výhradně ty z .NET Base Class Libraries (System.ComponentModel.DataAnnotations.dll), byť by jinak splňovaly pravidlo. Do projektu s DomainClasses nechceme kvůli Data Annotations referencovat knihovny Entity Frameworku (od .NET 4.5 a EF5 byly Data Annotations, které byly dříve v EF, přesunuty do BCL).

?? Diskutabilní je např. atribut SoftDelete, pokud má entita property IsDeleted.

### "Legacy" BusinessLayer[Generator] ;-)

Tato sekce se aplikuje na projekty, které používají náš původní BusinessLayer(Generator) a novým přístupem již mohou být některá pravidla překonána.

## IEnumerable<T> jako návratový typ

*IEnumerable<T>* jako návratový typ metody se používá jen tehdy, když metoda vrátí typ, který je určen k [deferred execution](#), tomu se navíc z důvodů horšího ladění a dohledávání chyb snažíme vyhnout. Pokud metoda vrátí skutečnou kolekci objektů, používáme jako typ návratové hodnoty obvykle *List<T>*.

### Nedoporučované/zakázané konstrukce

## Ternární operátor

Ternární operátor používáme opatrně. Konstrukce musí zůstat přehledná, snadno srozumitelná.

## Konstrukce ala C++

Vyhýbáme se konstrukcím, které zhoršují srozumitelnost kódu, například `++i` (`i++` je ve smyčce `for` samozřejmě povoleno, tyto konstrukce k sobě patří).

## Zkracování StringBuilderu změnou délky

```
sb.Length = sb.Length - 6
```

## Unit-testing

- používáme MSTest,
- používáme [Moq](#),
- test musí mít strukturu arrange-act-assert (AAA), nebo jinou všeobecně uznávanou (given-when-then, ...)
- viz též [konvence pojmenování](#)

## Code Analysis & Analyzers

Analyzery nám pomáhají ve vynucování coding-standards a provádí při buildech kontrolu určitých pravidel a požadků.


## Navigace

- [Centrální synchronizace .editorconfig a vynucování jeho voleb při buildech](#)
- [\[LEGACY\] Centrální pravidla a analyzery z \TOPOL](#)
  - [Instalace do Visual Studio](#)
    - [Visual Studio 2019](#)
  - [Pravidla](#)
  - [Kontrola pravidel](#)
  - [Potlačení pravidla v kódu](#)
  - [Možné problémy při kompilaci nového projektu na build serveru](#)
  - [StyleCop Rules - vypnutí pravidla](#)
    - SA1004: DocumentationLinesMustBeginWithSingleSpace (= NONE)
    - SA1008: An opening parenthesis within a C# statement is not spaced correctly. (= NONE)
    - SA1102: QueryClauseMustFollowPreviousClause (= NONE)
    - SA1114: ParameterListMustFollowDeclaration (= NONE)
    - SA1129: DoNotUseDefaultValueTypeConstructor (= NONE)
    - SA1131: UseReadableConditions (= NONE)
    - SA1410: RemoveDelegateParenthesisWhenPossible (= NONE)
    - SA1514: ElementDocumentationHeadersMustBePrecededByBlankLine (= NONE)
    - SA1623: PropertySummaryDocumentationMustMatchAccessors (= NONE)
    - SA1649: FileNameMustMatchTypeName (=NONE)

## Centrální synchronizace .editorconfig a vynucování jeho voleb při buildech

Novější projekty (např. BlazorStack) používají pro vynucování coding-standards týmový .editorconfig, který synchronizujeme mezi jednotlivými repository z jeho centrální podoby (TODO - zatím ručně, automatickou synchronizaci připravujeme). Snažíme se vystačit se stále bohatšími analyzery, které jsou vestavěny v .NET ekosystému, tj. bez StyleCop či SonarLint analyzérů.

.editorconfig předepisuje coding-standards, přičemž ty, které se projevují při vývoji jako WARNINGS, interpretuje build-server jako ERROR.

 Nový .editorconfig se teprve usazuje a je pravděpodobné, že najdete situace, které nevyhodnocuje správně. Prosím dejte nám ([@ Robert Haken](#) , [@ Jiří Kanda](#) ) vědět, abychom mohli vše doladit pro pohodlnou a produktivní práci.

Analyzery v této podobě již nejsou závislé na \TOPOL a každý projekt funguje sám o sobě i bez VPN. Celá věc se aktivuje přepínačem `<EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>` pro MSBuild, který obvykle umísťujeme pro celou solution do *Directory.Build.props*, bez tohoto přepínače se build-server pokouší aplikovat původní analyzery a pravidla z \TOPOL, viz níže.

## [LEGACY] Centrální pravidla a analyzery z \TOPOL

Původní pravidla a analyzery z \TOPOL se uplatňují při buildu všech projektů, které nemají volbu `<EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>`. Ta je u nových projektů nastavena v *Directory.Build.props*, výjimečně může být i v .csproj, pokud je potřeba project-level granularita (např. 002.HFW-HavitFramework).



### Visual Studio 2019

Nakopírovat soubor z `\\topol.havit.local\Library\VS2019\CodeAnalysis\HavitCodeAnalysis.targets` do složky `C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\MSBuild\Current\Microsoft.Common.Targets\ImportAfter`

Soubor načítá jiný target umístěný na topolu, což umožňuje aktualizovat jej na jednom místě. Ten zajišťuje:

- Zapne `CodeAnalysisTreatWarningsAsErrors`, pokud je zapnuto `TreatWarningsAsErrors`
- Nastaví `CodeAnalysisRuleSet` na `\\topol.havit.local\Library\VS2019\CodeAnalysis\HavitMain.ruleset`
- Zapojí analyzery
  - `Microsoft.AnalyzerPowerPack 1.1.0`
  - `StyleCop.Analyzers 1.0.2`
  - `System.Runtime.Analyzers 1.1.0`

Kdo by chtěl na svém stroji mít trvale zapnuté *Treat warnings as errors*, bez ohledu na nastavení projektu, pak nechť si k sobě zkopíruje ještě soubor `Havit.TreatWarningsAsErrors.targets` analogicky.

### Pravidla

Množina kontrolovaných pravidel je v `\\topol.havit.local\Library\VS2019\CodeAnalysis\HavitMain.ruleset` (lze otevřít ve Visual Studiu).

Mezi pravidly je zapnuté jedno (víceméně náhodně vybrané pravidlo) z `Managed Binary Analyzers`. Pokud není žádné pravidlo vybráno, zobrazuje se chyba [CA0064](#).

### Kontrola pravidel

Ke kontrole pravidel dochází v rámci kompilace (přímo pomocí `csc.exe`). Pokud se od posledního buildu nezměnili zdrojové kódy, build projektu se automaticky přeskakuje (zjednodušeně řečeno). Pak se nespustí `csc.exe` a proto se případné chyby nezobrazí.

### Potlačení pravidla v kódu

```
[SuppressMessage("Stylecop.Analyzers", "SAxxxx", Justification = "Dvod  
potlaení pravidla")]
```

### Možné problémy při kompilaci nového projektu na build serveru

Pokud se při kompilaci projektu objeví následující chyby (kompilace na build serveru):

```
CSC : error CS8034: Unable to load Analyzer assembly \\topol.havit.  
local\Library\VS2019\CodeAnalysis\StyleCop.Analyzers.dll : Could not  
load file or assembly 'file:///\\topol.havit.  
local\Library\VS2019\CodeAnalysis\StyleCop.Analyzers.dll' or one of its  
dependencies. Operation is not supported. (Exception from HRESULT:  
0x80131515) [D:\Agent2\_work\53\s\PinRevealTest\PinRevealTest.csproj]  
CSC : error CS8034: Unable to load Analyzer assembly \\topol.havit.  
local\Library\VS2019\CodeAnalysis\System.Runtime.Analyzers.dll : Could  
not load file or assembly 'file:///\\topol.havit.  
local\Library\VS2019\CodeAnalysis\System.Runtime.Analyzers.dll' or one  
of its dependencies. Operation is not supported. (Exception from  
HRESULT: 0x80131515) [D:\Agent2\_work\53\s\PinRevealTest\PinRevealTest.
```

```

csproj]
CSC : error CS8034: Unable to load Analyzer assembly \\topol.havit.
local\Library\VS2019\CodeAnalysis\System.Runtime.CSharp.Analyzers.dll :
Could not load file or assembly 'file:///\\topol.havit.
local\Library\VS2019\CodeAnalysis\System.Runtime.CSharp.Analyzers.dll'
or one of its dependencies. Operation is not supported. (Exception from
HRESULT: 0x80131515) [D:\Agent2\_work\53\s\PinRevealTest\PinRevealTest.
csproj]
CSC : error CS8034: Unable to load Analyzer assembly \\topol.havit.
local\Library\VS2019\CodeAnalysis\Microsoft.AnalyzerPowerPack.Common.
dll : Could not load file or assembly 'file:///\\topol.havit.
local\Library\VS2019\CodeAnalysis\Microsoft.AnalyzerPowerPack.Common.
dll' or one of its dependencies. Operation is not supported. (Exception
from HRESULT: 0x80131515) [D:
\Agent2\_work\53\s\PinRevealTest\PinRevealTest.csproj]
CSC : error CS8034: Unable to load Analyzer assembly \\topol.havit.
local\Library\VS2019\CodeAnalysis\Microsoft.AnalyzerPowerPack.CSharp.
dll : Could not load file or assembly 'file:///\\topol.havit.
local\Library\VS2019\CodeAnalysis\Microsoft.AnalyzerPowerPack.CSharp.
dll' or one of its dependencies. Operation is not supported. (Exception
from HRESULT: 0x80131515) [D:
\Agent2\_work\53\s\PinRevealTest\PinRevealTest.csproj]

```

Je třeba odebrat balíček "Microsoft.Net.Compilers" a vše, co ho používá (například také Microsoft.CodeDom.Providers.DotNetCompilerPlatform).

V případě následného problému při kompilaci ve visual studiu je třeba restartovat visual studio!

#### StyleCop Rules - vypnutá pravidla

#### SA1004: DocumentationLinesMustBeginWithSingleSpace (= NONE)

Vypnuto pro nežádoucí implementaci.

Viz např. Goran.

```

/// 


/// ProjektID int NOT NULL</item>
/// Projekt objekt</item>
/// PokryvaVsechnyFaze bit NOT NULL (bool)</item>
/// FazeID int Nullable</item>
/// Faze objekt</item>
/// TeamID int NOT NULL</item>
/// Team objekt</item>

```

#### SA1008: An opening parenthesis within a C# statement is not spaced correctly. (= NONE)

Vypnuto, umožníme tak použití Tuples v C# 7.0.

#### SA1102: QueryClauseMustFollowPreviousClause (= NONE)

Neumožní mít uprostřed query komentáře.

#### SA1114: ParameterListMustFollowDeclaration (= NONE)

Vypnuto, toto pravidlo nechceme.

```
container.Register(
```

```
// UnitOfWork  
Component.For<IUnitOfWork>().ImplementedBy<UnitOfWork>().LifestylePerWebRequest(),
```

### SA1129: DoNotUseDefaultValueTypeConstructor (= NONE)

Vypnuto, není chybou (lepší `x = new CancellationToken` než `x = default(CancellationToken)`).


### SA1131: UseReadableConditions (= NONE)

Ovšem některé podmínky to může znečitelnit (`if ((satisfactionRate < 0) || (100 < satisfactionRate))` - konstanta nesmí být dle pravidla nalevo).

### SA1410: RemoveDelegateParenthesisWhenPossible (= NONE)

Vypnuto, není chybou.

```
ExecuteWithCulture(uzivatel.GetCultureEffective(), delegate {  
    {  
        . . . . .  
        . . . . .  
        . . . . .  
    }  
})
```



### SA1514: ElementDocumentationHeadersMustBePrecededByBlankLine (= NONE)

Vypnuto, pokud používáme regiony, není dokumentačním komentářem volný řádek.

### SA1623: PropertySummaryDocumentationMustMatchAccessors (= NONE)

Vypnuto, nepotřebujeme, aby summary komentářů povinně obsahovaly "Gets or sets...".

### SA1649: FileNameMustMatchTypeName (=NONE)

Vypnuto, protože soubory Views v MVC mají pojmenování `Class.cshtml.cs`, které jsou s pravidlem nekompatibilní kvůli "cshtml". Chceme umožnit standardní pojmenování souborů a tříd MVC.

## Console Application

Při vytváření konzolové aplikace nezapomenout na:

- `ExceptionTracer`
- `IdentityMapScope`

K tomu je potřeba referencovat knihovny *Havit.Business.dll* a *Havit.dll*.

Metoda `Main` typicky vypadá nějak takto:

```
#region Main  
static void Main(string[] args)  
{  
    Havit.Diagnostics.ExceptionTracer.Default.SubscribeToUnhandledExceptions  
(false);  
  
    using (IdentityMapScope identityMapScope = new IdentityMapScope())  
    {  
        Importuj();  
    }  
}  
#endregion
```

## ExceptionTracer

Základní popis mechanismu viz KnowledgeBase <http://knowledge-base.havit.cz/2008/01/31/jednoduchy-exception-logging-pomoci-tracetracesource-mechanismu-net/>

Používáme v případě konzolových i WinForm aplikací a k odbru se přihlašujeme na začátku metody `Main()`.

Nachází se v knihovně `Havit.dll`, namespace `Havit.Diagnostics`.

## CSS Coding Standards

### Navigace

- [Základní standardy](#)
  - [In-line stylování se vyhýbáme](#)
  - [Pojmenování tříd](#)

### Základní standardy

#### In-line stylování se vyhýbáme

Pokud to je jenom trochu možné, vyhýbáme se in-line stylování, tj. CSS pravidlům umístěným u prvku do atributu `style` a obdobné techniky (třeba ve WebForms cokoliv jiného než `XyStyle-CssClass`, např. `XyStyle-HorizontalAlign`). Zásadně preferujeme extrakci stylů do CSS/LESS souborů a využití potenciálu této technologie.

#### Pojmenování tříd

CSS třídy pojmenováváme malými písmeny, víceslovné názvy oddělujeme pomlčkou, např. `button-large`.

## GIT Workflow Standard

### Zásady

- vyvíjíme přírůstkově, nepodporujeme cherry-picking
- držíme jednu hlavní vývojářskou linii, ideálně lineární historii v master
- každý commit by měl dostávat projekt z jednoho konzistentního stavu do nového konzistentního stavu (ne nutně nasaditelný, ale nebránící práci ostatních vývojářů)
- nikdo by neměl potřebovat řešit konflikty jiných dvou vývojářů (pokud nastane konflikt, jednou z konfliktních stran jsem vždy "já")

### Short Story

1. Jediná vývojářská mainline = master.
2. V případě potřeby short-lived feature branches *feature/feature-name*.
  - a. Často dotahujeme do feature branch změny z *master* (merge z master do feature).
  - b. Nakonec merge feature do master (bez squash, rebase možný) a smazání feature branch.
3. Nasazování tagujeme *release/<environment>/<release version>*. Při nasazování přes ADOS Releases vzniká automaticky.
4. Přípravu release a následné hotfixování provádíme v release branch *release/<release name>*, např. *release/iteration15*.
  - a. Branch zakládáme, až když je to potřeba, např. pokud potřebujeme už zahájit v *master* práce na dalším release, nebo pokud potřebujeme připravit hotfix něčeho již nasazeného.
  - b. Často (nejpozději s nasazováním) integrujeme změny z release branch do master. Pokud již výjimečně existuje release branch pro další release, musíme nejprve tam.

### Long Story

#### Předpoklady

1. Používáme vždy jediný centrální repositář (Azure DevOps Repos, ADOS). Každý vývojář pracuje nad svým lokálním klonem tohoto centrální repa (= jediný remote), nepoužíváme žádné okliky ani přímá propojení repositářů, vše jde hvězdicově přes ADOS.



2. Každý regulérní release je založený na zdrojových kódech předchozích release. Neudržíme více aktivních linií vývoje zdrojových kódů našich projektů, více živých verzí. (Pokud ano, jedná se o jiný projekt se separátními zdrojovými kódy, např. 003.GOR vs. 003.GOR-G3.)
3. Vývoj projektů je převážně přírůstkový.
  - a. Obsah release je buď dopředu plánovaný, nebo vzniká v režimu flow z aktuálního postupu vývoje.
  - b. Nesestavujeme release výběrem z již vyvinutých funkcí. Nežádoucí jsou požadavky na vyjmutí určité vyvinuté funkčnosti z release. I s takovými situacemi se však umíme výjimečně vypořádat.
4. Vyvíjíme pomocí relativně malých přírůstků (tasků, commitů), obvykle v rozsahu ne větším než několik hodin programátorské práce, které se snaží posouvat projekt z jednoho konzistentního stavu do nového konzistentního stavu.
  - a. Minimální - nikoliv postačující - podmínkou konzistence je plná buildovatelnost projektu (vč. procházejících unit-testů a code-analyzerů), v ideálním případě pak dokonce nasaditelnost projektu (přínejmenším jako něco, čemu se snažíme přiblížit, když plánujeme vývojářské práce).
  - b. Základní podmínkou konzistence je držet projekt ve stavu, který nebude bránit ve vývoji ostatním členům týmu a nebude projekt příliš vzdalovat od nasaditelné podoby, tj. snažíme se minimalizovat situace, kdy bychom projekt nějakým mezistavem rozložili. Pokud je chvilkový rozklad nevyhnutelný, domluvíme se na vhodném postupu s ostatními členy týmu (např. použijeme branch).
5. Jsme tým, kde spolu intenzivně komunikujeme. Jsme schopni efektivně sdílet informace o probíhajícím vývoji, domlouvat se na potenciálních kolizích a návaznostech, vnímat chod projektu i požadavky "zákazníka".
  - a. Proto nepotřebujeme technická omezení, která by striktně chránila zdrojové kódy před neodpovědným či škodlivým postupem (nutná např. u OSS).
  - b. Každý může udělat chybu, nebo se dostat na hranice svých aktuálních dovedností. I GIT má spoustu pastí, které vás mohou potrápit. Nebojte se zeptat, navzájem si poradit.
6. Nepoužíváme ``git push --force``, nikdy. Jediná dovolená destrukční operace na centrálním repozitáři je smazání feature branch, kterou jsme sami vytvořili.

## Požadavky

1. Chceme být schopni kdykoliv získat zdrojové kódy od aktuálně nasazené podoby projektu (release).
2. Chceme být schopni kdykoliv nasadit drobnou opravu nebo úpravu aktuálně běžící podoby projektu (hotfix), přestože již probíhá vývoj nových funkcí.
3. Chceme minimalizovat integrační úsilí (tj. pracnost slučování více vývojových větví).
  - a. Chceme co nejméně konfliktů, nicméně ne za cenu omezení paralelizace práce. Konflikty jsou nevyhnutelné, chceme se s nimi však vypořádat efektivně, zejména pak dobrou komunikací a koordinací práce vývojářů.
  - b. Pokud již konflikt nastane, chceme ho řešit co nejdříve, ideálně bezprostředně při jeho vzniku (při commitu+push druhého z konfliktních souborů), tj. dokud má vývojář v hlavě, co zrovna řešil a jak konfliktní situaci efektivně vypořádat.
  - c. Vyhýbáme se uspořádání, kde bychom měli řešit konflikty "s několikadenním odstupem", kdy vývojář již nemá v hlavě aktuální kontext a musí si vše znovu vybavit.
  - d. Nikdo by neměl potřebovat řešit konflikt mezi dvěma cizími změnami zdrojových kódů (vždy by měla být jedna z konfliktních změn moje).
4. Využíváme automatizace - Continuous Integration (CI) buildy, plně automatické nasazování do CI prostředí, ručně spouštíme automatizované nasazování na STAGE a PROD.

## Shrnutí

1. Používáme GIT workflow, které je variací **základního modelu OneFlow** (<https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow>), vývojovou mainline je pro nás *master*.
2. Naše GIT workflow je škálovatelné.
  - a. Má určitou základní výchozí podobu,
  - b. podle situace konkrétního týmu a projektu lze však některé nepovinné prvky vyjmout a flow zjednodušit (např. nepoužívat feature/release branches),
  - c. stejně tak máme připravené nadstavbové postupy pro komplexnější situace, které si to vyžadují.

## Popis základního flow

### Běžný vývoj

Vývojář začíná svoji práci na *master*.

## Základní jednoduchá podoba

Pokud se jedná o triviální task na několik hodin, který převádí projekt z jedné konzistentní podoby do nové konzistentní podoby, tj. zejména nezablokuje práci někomu jinému, provedu commit přímo na lokální *master* a hned i *push* do ADOS na *origin/master*.

Pokud se mezi tím *master* v ADOS změnil, provede *pull* s *rebase* a zkusí *push* znovu.

Pokud přes veškerou péči CI build selže (kompilace, code-analyzer, unit-testy), dostávám notifikaci (e-mail) a jsem povinen okamžitě provést korekci, nebo revert.

master



## Rozšířená podoba s feature branch

Zejména pokud:

- potřebuji mít rozpracováno více věcí najednou (a nevystačím si s chvilkovým *stash*),
- nebo vyvíjím funkčnost, která není triviální a vývoj mi potrvá několik dní,
- nebo potřebuji součinnost více vývojářů a práci si nemůžeme efektivně předávat přes konzistentní podobu projektu v *master*,
- nebo pracuji na projektu, kde bylo dohodnuto, že feature branches explicitně používáme a základní podoba není žádoucí,

pak vytváříme pro takový vývoj *feature branch* (*/feature/název-feature*) z *master*.

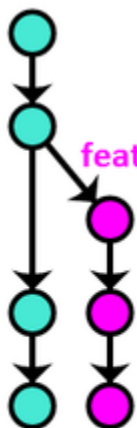
Co nejčastěji dělám dotahování změn z *master* (příčetně, podle očekávané aktivity ostatních, např. jednou denně, nebo dle komunikace v týmu, pokud v průběhu dne vím, že někdo jiný dal na *master* něco, s čím je potřeba se integrovat):

- dokud jsem *feature branch* nepushnul na server, mohu dělat opakovaně *rebase* nad změny z *master*,
- jinak jde o merge z *master* do *feature branch*.

Pokud je to potřeba a vývoj probíhá déle, mohu si na *feature branch* nastavit CI build (pozor však, že máme jediné CI prostředí, nikoliv per *feature*).

Nakonec udělám merge *feature branch* do *master* a *feature branch* smažu. Před samotným merge do *master* preferujeme provést *rebase* nad poslední verzí *master*, zejména pokud jsem byl jediným vývojářem celé *feature*.

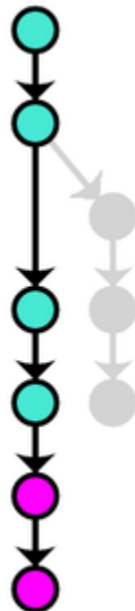
master



feature/my-feature



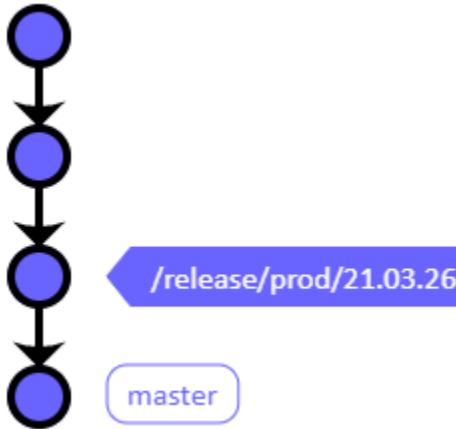
master



Releasujeme pomocí Azure DevOps Pipelines/Releases, každý release (mimo CI) automaticky přidá do repository ke commitu tag `/release /env/name`, např. `release/production/003.gor-2021-03-09.02r`. Můžeme tak kdykoliv snadno dohledat zdrojové kódy konkrétní nasazené podoby.

## Minimalistická jednoduchá podoba release

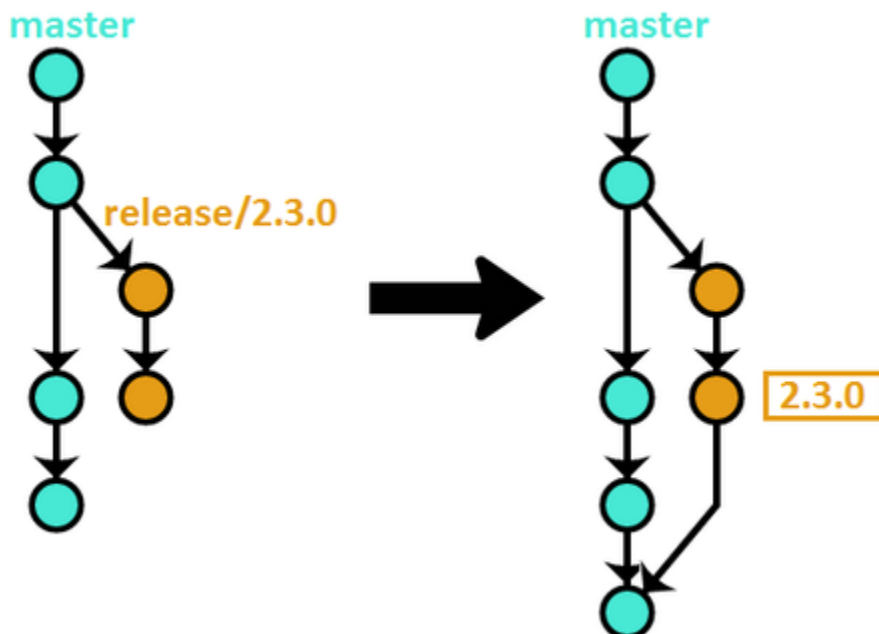
Na triviálních projektech se sekvenčním postupem prací můžeme releasovat přímo z `master` branch. Vybereme si commit, který chceme releasovat (obvykle poslední) a spustíme jeho release pipeline.



## Základní podoba release

Nasazování probíhá z `release` branch (`/release/název`), která se před započítím prací na dalším release oddělí z `master`, tj. například pracuji v `master` na iteraci 10. Někteří vývojáři už by se ale mohli pustit do prací na iteraci 11. Pokud to nejsou práce zcela bezrizikové a konzistentní, pak přes započítím těchto prací vytvořím z `master` branch `/release/iteration10`, kde dočišťuji sprint pro jeho release, zatímco práce na iteraci 11 pokračují v `master`.

Názvem release je sekvenční označení, např. `release/iteration1`, nebo číslo verze `/release/2.3`, dle organizace projektu.



V této `release` branch se dočišťuje konkrétní release (např. dle poznatků z testování na STAGE), změny se průběžně mergují do `master` (bez rebase).

Z této `release` branch se následně spouští i ADOS deployment pipeline, která vytvoří na nasazovaném commitu příslušný tag (např. `release /production/003.gor-2021-03-09.02r`).

Release branches se nemažou, obvykle slouží i pro hotfixování daného release (viz níže).

## Rozšiřující variace release

Pokud by to nějak významně usnadnilo práci s toolingem (zejm. Azure DevOps Pipelines) lze pro jednotlivá prostředí držet long-lived release branches.

Pro každé prostředí pak držíme long-lived "marker" branch (*/stage*, */prod*), která ukazuje na commit na daném prostředí nasazený. Na těchto branches vývoj neprobíhá, pouze se do nich mergeje (*--ff-only*) stav při nasazení, resp. automatizace má tyto branches nastaveny jako zdroj pro nasazování (na vybraných projektech lze dokonce automaticky s pushem/PR spouštět nasazování).

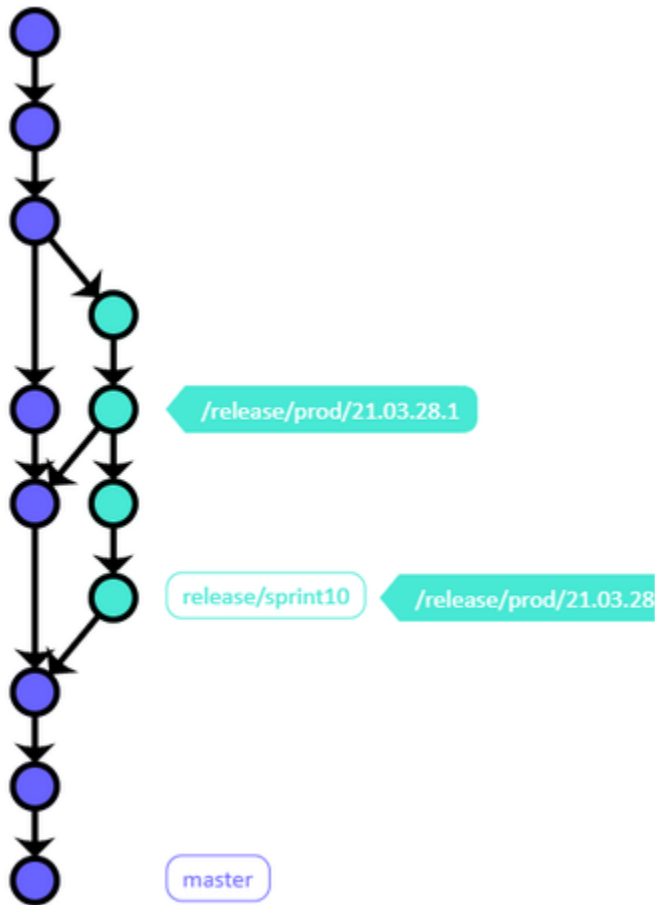
### Hotfixování

Short-story: Hotfixování není nic jiného, než pokračování v úpravách daného release v jeho release branch (pokud existuje, jinak ji musíme z příslušného tagu v *master* vytvořit).

Potřebujeme branch, která vychází z posledního nasazeného commitu. Pokud potřebujeme udělat opravu již nasazené verze (nebo verze teprve připravované pro produkční nasazení na *stage*), potřebujeme branch, která vychází z posledního nasazeného commitu (ten máme označený tagem).

Pokud jsme již nějaký hotfix dělali, nebo máme oddělenou release branch daného nasazování, pak již potřebná branch existuje, např. */release/iteration10* a můžeme v ní s hotfixy daného release pokračovat. Pokud neexistuje, musíme ji založit.

Změny se průběžně mergejí do *master* (bez rebase), abychom se nám v dalším release chyba nevrátila. Každé nasazení (mimo CI) se automaticky taguje (*/release/env/version*).



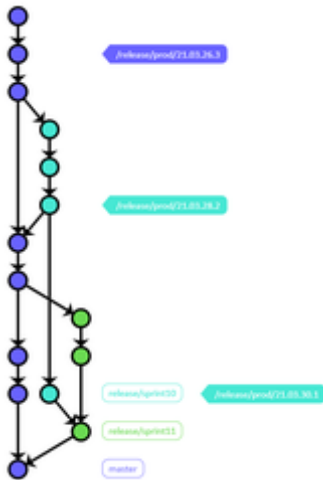
### Specifické scénáře

#### Hotfix staršího release (PROD), když už došlo k oddělení release připravovaného (UAT)

Není to úplně běžná situace, aby bylo nasazování takto zpožděno za vývojem, ale může se stát. V produkci máme */release/iteration10*, ladíme k nasazení již založenou branch */release/iteration11*, protože v *master* už probíhá vývoj iterací 12. Potřebujeme udělat fix iterací 10 do produkce.

Hotfix klasicky uděláme nad produkčními zdrojovými kódy, tj. do */release/iteration10*.

Jediným specifikem je, že merge hotfix commitů neprovádíme z `/release/iteration10` rovnou do `master`, ale nejprve uděláme merge do `/release/iteration11` a ten pak do `master`.



#### Požadavek na vyjmutí určité funkčnosti z připravovaného release

1. Pokud je to opravdu potřeba, preferujeme feature-switches, tj. konfigurační eliminace dané funkčnosti, např. white-listing uživatelů, kterým se daná funkce projeví.
2. Nemělo by nastat, je proti našim předpokladům vývoje.
3. Nemělo by nastat, je proti našim předpokladům vývoje.
4. Nemělo by nastat, je proti našim předpokladům vývoje.
5. Pokud není jiného zbylí, přichází čas na lokální kreativitu. Např. vytvoříme "orphan" release branch, kde danou funkčnost "odprogramujeme" (při troše štěstí revertujeme některé commity, např. feature branch merge).
  - a. Takovou branch pak naopak nemerujeme do `master`, abychom si tam hotovou feature nezahlavili.
  - b. Rizikové se to stává s následujícími hotfixy takového release, protože tam už se dostáváme na hranici nežádoucího cherry-pickingu. Obvykle je jednodušší dostávat dané hotfixy do `master` jako replay změn z orphaned release commitu.
6. Nemělo by nastat, je proti našim předpokladům vývoje.

#### Rebase upřednostňujeme před merge

- linearizace historie, přehlednost
- doporučení nastavení do global config
  - `pull-rebase = true` nebo **merges**
  - `auto-stash`

#### GIT Global Configuration (from Command Line)

```
#git config --global pull.rebase true
#git config --global pull.rebase preserve    <= DEPRECATED
git config --global pull.rebase merges
git config --global rebase.autoStash true
```



#### Visual Studio - Known Issue

## Synchronization | 002.HFW-HavitFramework

❌ The pull operation failed. See the [Output window](#) for details. ❌

If you set `pull.rebase` to `merges (preserve)`, Visual Studio will likely be unable to interpret the result of the pull operation. Team Explorer in Visual Studio will state *The pull operation failed. See the Output window for details.* right after your Pull and this is what you will find at the end of your Output window (Source Control - Git):

```
...
Successfully rebased and updated refs/heads/master.
Operation is not valid due to the current state of the
object.
```

- netýká se hotfixů (nechceme změny z master releasovat, ani měnit historii)
- ! nikdy nedělat rebase u sdílených feature branches
- [jak zapnout rebase \(link\)](#)

### Tagování releasů

- tam, kde nasazujeme aplikace pomocí Release Management, jsou zdrojové kódy tagovány konvencí "release/Environment/Release Name" (např. release/production/149.gbp-2018-08-20.01r).
- v ostatních případech, pokud chceme informaci o releasech v SCM, tak tagovat commity v master, konvence pojmenování takových tagů: "release/rrrr-mm-dd...." ev. "release/environment/rrrr-mm-dd...."
- nutnost pro [ChangeLogGenerator](#)

### Lokální invencím se meze nekladou, chceme však Continuous Integration

- co si kdo dělá na lokálním repozitáři, je nám jedno, pravidla platí pro centrální repozitáře ADOS
- "soukromou" feature branch si mohou zálohovat do Azure DevOps (právo na force-push má vlastník/autor branche)

## Source Code Management (SCM)

### Základní zásady

- vždy **buildovatelná** revize (pokud se nezadaří, okamžitě opravit)
- jeden checkin = jeden work item (s jistou mírou tolerance)
- associated work items = work item v TFS, který je checkinem implementován
- **komentář** revize = název work-itemu v TFS včetně čísla, např "1234 Subjekt-Edit.aspx - padá na NullReferenceException"
  - ve Visual Studiu můžeme v okně Pending Changes přenést popisec asociovaného work-itemu snadno pomocí Copy + Paste (ze sekce Related Work Items do Comment), stačí pak odmazat poslední slovo Resolve/Associate

### Automated tests

- je možné provádět checkin, při jehož buildu selžou automated testy, ale pouze tehdy, pokud je to tak dopředu zamýšleno (například nejprve napíšeme unit testy a později implementujeme funkcionální aplikaci)
- pokud selhávají unit testy (a není to zamýšleno dle předchozího bodu), je nutné okamžitě opravit
- projekt, jehož unit testy selhávají, není hotový
- projekt, jehož unit testy selhávají, není nasaditelný

## SQL Coding Standards

### Navigace

- [Schéma databáze](#)
  - [Naming Guidelines](#)
    - [Pojmenování tabulek](#)
    - [Pojmenování sloupců](#)
    - [Pojmenování indexů/klíčů](#)

- [Pojmenování uložených procedur](#)
- Business-layer-generator specific
  - [NULL vs. NOT NULL na sloupcích typu nvarchar](#)
- T-SQL Coding Standards
  - [Formátování kódu](#)
    - [Odsazování \(indent\)](#)
  - [Komentáře](#)
    - [Magic numbers v kódu](#)
  - [Nedoporučované/zakázané konstrukce](#)
    - [Cursors, Triggers](#)

Schéma databáze

Naming Guidelines

## Pojmenování tabulek

- Tabulky se pojmenovávají tak, jak se jmenují odpovídající business-object třídy (entity).
- Tabulka reprezentující entitu se jmenuje jako entita, v jednotném čísle, PascalCase. Např. `KontaktniOsoba`.
  - Pro tabulku s nastaveními aplikace používáme pojmenování `Nastaveni` nebo `ApplicationSettings` (na "settings" koukáme jako na pomnožené, byť to dle anglické gramatiky není úplná pravda a singulár "setting" existuje).
- Tabulka reprezentující vztah M:N s vlastnostmi se pojmenuje vlastním názvem, jako by se jednalo o entitu. Např. `Membership`.
- Tabulka reprezentující vztah M:N bez vlastností se zpravidla pojmenuje tak, že se podtržítkem oddělí názvy tabulek, které se vztahu účastní. Např. `Uzivatel_Role`. Pokud není M:N vztah mezi tabulkami jednoznačný, lze i zde použít vlastní název, nebo název hlavní entity ve vztahu a název kolekce, např. `Equipment_SpotrebnMaterial` nebo `Equipment_DoplnekovySortiment`.

## Pojmenování sloupců

Názvy sloupců odpovídají názvům vlastností dle designu.

Pro konkrétní případy existují vyhrazené názvy sloupců:

- primární klíč - Sloupec se jmenuje vždy stejně jako tabulka se suffixem ID.
- cizí klíč - Sloupec se jmenuje jako vlastnost, kterou cizí klíč reprezentuje, končí suffixem ID. Pokud není potřeba rozlišení, zpravidla pojmenováváme stejně, jak se jmenuje cílová tabulka.
- `Created` - Sloupec, který reprezentuje okamžik založení záznamu a má nastaven `DEFAULT GETDATE()`,
- `Deleted` - sloupec, který označuje smazané záznamy, resp. okamžik smazání (datetime). V případě bit/bool se hodí `IsDeleted`.

Sloupec `Created` a `Deleted` jako zvyklost umísťujeme v tabulce vždy jako poslední, PK samozřejmě jako první.

## Pojmenování indexů/klíčů

Indexy a unikátní klíče (indexes/unique keys) pojmenováváme dle konvencí (názvy jednotlivých sloupců jsou odděleny podtržítkem `_`):

- `PK_NazevTabulky`, `FK_NazevTabulky_NazvySloupcu` - primární a cizí klíče (automaticky vytvořeny) - pouze při přejmenování tabulky je potřeba ručně upravit názvy
- `FKX_NazevTabulky_NazvySloupcu` - vytvářeny BusinessLayer-generátorem
- `IDX_NazevTabulky_NazvySloupcu` - běžný index
- `UIDX_NazevTabulky_NazvySloupcu` - unikátní index

Podrobnější informace jsou k dispozici v [článku o indexech BusinessLayer-generátoru](#).

## Pojmenování uložených procedur

Pokud je generován wrapper do business-layeru, zásadně držíme pattern pojmenování `Class_MethodName`, kde *Class* je třída, do které se výsledná metoda vygeneruje (obvykle též návratový typ uložené procedury).

Business-layer-generator specific

## NULL VS. NOT NULL na sloupcích typu nvarchar

- Rozhodnutí volby null nebo not-null sloupců typu `nvarchar` se určuje stejně, jako pro jakékoliv jiné sloupce, tj. not-null je tehdy, je-li hodnota povinná.
- Například `EnterpriseLabel` dle této volby rozhoduje, zda má zobrazit hvězdičku indikující povinný údaj.
- Sloupce mají mít vždy výchozí hodnotu prázdný text.

**Odsazování (indent)**

SQL kód odsazujeme tabelátory, jednotlivé statementy odsazujeme dle jejich logické stavby:

```
ALTER PROCEDURE [dbo].[Zbozi_SearchZbozi]
(
    @Nazev nvarchar(100) = NULL,
    @CenaMaloobchodniOd decimal = NULL,
    @CenaMaloobchodniDo decimal = NULL,
    @CenaVelkoobchodniOd decimal = NULL,
    @CenaVelkoobchodniDo decimal = NULL,
    @BarvaID int = null,
    @JenZboziSkladem bit = NULL,
    @VcetneVelkoobchodu bit = 0
)
AS
BEGIN
    SET NOCOUNT ON;

    ;WITH VysledneSkupinyZbozi(SkupinaZboziID) AS
    (
        SELECT SkupinaZboziID FROM SkupinaZbozi WHERE (Nazev like
@Nazev AND Zobrazovat = 1)
        UNION ALL
        SELECT SkupinaZbozi.SkupinaZboziID
        FROM SkupinaZbozi
        INNER JOIN VysledneSkupinyZbozi ON
(VysledneSkupinyZbozi.SkupinaZboziID = SkupinaZbozi.ParentID)
        WHERE (Zobrazovat = 1)
    )
    SELECT SkupinaZboziID INTO #VysledneSkupinyZbozi FROM
VysledneSkupinyZbozi

    SELECT DISTINCT Zbozi.ZboziID
    FROM Zbozi
    LEFT JOIN Zbozi_SkupinaZbozi ON (Zbozi_SkupinaZbozi.ZboziID
= Zbozi.ZboziID)
    WHERE
        (Zbozi.Deleted = 0)
        AND (Zbozi.NabizetEshop = 1)
        AND ((@Nazev IS NULL)
            OR (Nazev LIKE @Nazev)
            OR (NazevVarianty LIKE @Nazev)
            OR (Poznamka LIKE @Nazev)
            OR (Soft4SaleMatID LIKE @Nazev)
            OR (Zbozi_SkupinaZbozi.SkupinaZboziID IN (SELECT
SkupinaZboziID FROM #VysledneSkupinyZbozi))
```



```

        )
        AND ((@CenaMaloobchodniOd IS NULL) OR
(CenaMaloobchodniBezDph >= @CenaMaloobchodniOd))
        AND ((@CenaMaloobchodniDo IS NULL) OR
(CenaMaloobchodniBezDph <= @CenaMaloobchodniDo) OR
(CenaMaloobchodniBezDph IS NULL))
        AND ((@CenaVelkoobchodniOd IS NULL) OR
(CenaVelkoobchodniBezDph >= @CenaVelkoobchodniOd))
        AND ((@CenaVelkoobchodniDo IS NULL) OR
(CenaVelkoobchodniBezDph <= @CenaVelkoobchodniDo) OR
(CenaVelkoobchodniBezDph IS NULL))
        AND ((@BarvaID IS NULL) OR (BarvaID = @BarvaID))
        AND ((@JenZboziSkladem IS NULL) OR (@JenZboziSkladem = 0)
OR (SkladAktualniMnozstvi > SkladMinimalniMnozstvi))
        AND ((@VcetneVelkoobchodu = 1) OR (Zbozi.
NabizetJenVelkoobchodne = 0))

    DROP TABLE #VysledneSkupinyZbozi
END

```

## Komentáře

## Magic numbers v kódu

Veškeré magic-numbers (typicky hodnoty enum) používané v kódu SQL musí být komentovány takto:

```

WHERE (ObjectStateID = 5 /* Ready */) OR (ObjectStateID = 8 /* Canceled
*/)

```

## Nedoporučované/zakázané konstrukce

## Cursors, Triggers

Bez schválení nebo instrukcí od JK/RH se zakazuje použití kurzorů a triggerů.

(Zejména z didaktických důvodů, protože 99% kurzorových operací se dá přepsat na efektivnější set-based operace a úkony typicky řešené v triggerech zase děláme obvykle přímo v BusinessLayeru.)

## Vizuální standardy

### Navigace

- [Standardní texty](#)
  - [Validační hlášky](#)
- [Formátování](#)
- [Zarovnávání v tabulkách](#)
  - [Zalamování](#)
- [Řazení záznamů](#)
- [Kdekoliv se zobrazuje seznam záznamů \(Grid, DropDownList, Repeater, ListView\), musí být explicitně řízeno jejich pořadí, např.:](#)
- [Řazení v GridView](#)
- [DefaultButton, DefaultFocus](#)
- [Položky číselníků / DropDownListů](#)
- [Title / Nadpis](#)
- [AutoFill/AutoComplete na standardních input-fieldech](#)

## Standardní texty

- property-labels začínají velkým písmenem (*Uživatelské jméno*), nekončí dvojtečkou (pokud je požadována, není součástí resources, ale hardcodována do EditPage)
- hodnoty číselníků, resp. DropDownListů začínají typicky malým písmenem (*modrá, červená*)
  - speciální položky uvozujeme mínusy před a za (*-nerozhoduje-*)
- Text **Messengeru** (Toastr) vždy jako věta s velkým písmenem na začátku a tečkou/vykřičníkem na konci
  - *Zákazník byl úspěšně založen.* (Insert)
  - *Zákazník byl úspěšně uložen.* (Update)
  - *Zákazník byl úspěšně smazán.* (Delete)

## Validační hlášky

**ErrorMessage** validátorů je vždy věta s velkým písmenem na začátku a tečkou na konci:

- **RequiredFieldValidator** - *E-mail musí být zadán. / E-mail musí být vyplněn.*
- *Datum objednání nemá správný formát.*
- *Datum objednání nesmí být v minulosti.*

U validace formátu má validační hláška obsahovat instrukce, jak má vypadat správný formát, popř. s příkladem. U formátu data/čísla/e-mailu se předpokládá, že je všeobecně známý korektní formát (pokud není vyžadován nějaký speciální).

- *Adresu zadejte ve formátu <http://www.example.com>, včetně počátečního <http://> nebo <https://>.*

## Formátování

Není-li výslovně určeno jinak, zásadně formátujeme

- **datum** DateTime (date, TDate) jako {0:d} (CZ: 15.6.2007)
- **datum a čas** DateTime (smalldatetime) jako {0:g} (CZ: 15.6.2007 15:30)
- **celá čísla** jako {0:n0} (CZ: 1 500)
- **desetinná čísla** jako {0:n2} (CZ: 1 500,00), případně jiný počet desetinných míst dle kontextu
- **peněžní částky (Money)** jako {0:n2} (CZ: 1 500,00), pokud známe měnu, pak tato následuje po mezeře (tvrdé, nebo musíme jinak zajistit, aby se nezalomilo)
  - pokud haléře nejsou relevantní (velké objemy, celé částky), pak používáme {0:n0}
  - vyhýbáme se formátování měny {0:c}, přináší to nejistotu ohledně podoby výstupu a hlavně 100 Kč není \$100 € (po přepnutí na en-US)

Zvláště nepříjemnou chybou je zobrazování složky, která v datovém typu není obsažena, např. 12.3.2014 0:00 pro datum, nebo 12.3.2014 12:56:00 pro smalldatetime.

## Zarovnávání v tabulkách

Není-li výslovně určeno jinak, zásadně zarovnáváme v tabulkách

- **datum** na střed (ItemStyle-CssClass="center")
- **čísla** doprava (ItemStyle-CssClass="right")

## Zalamování

- měna u částky (ani jiné jednotky, nebo číslo samotné) se nesmí zalamovat na samostatný řádek (platí i pro měnu/jednotky za TextBoxem/NumericBoxem)
- pokud je titulek gridu evidentně výrazně delší než hodnoty ve sloupci, měl by být zalomen, nebo zkratkou
  - odstrašujícím příkladem budiž ikonka 16x16px ve sloupci nadepsaném "Stav záznamu" (správně sloupec bez titulky, nebo max. "Stav")

## Řazení záznamů

Kdekoliv se zobrazuje seznam záznamů (Grid, DropDownList, Repeater, ListView), musí být explicitně řazeno jejich pořadí, např.:

- `QueryParams.OrderBy`
- `ORDER BY` ve stored-procedure
- `GridView.DefaultSortExpression`
- `EnterpriseDropDownList.AutoSort="true"`

- `BusinessObjectCollection.Sort()`
- `BLG: GetAll_Sorting, Collection_CollectionName_Sorting`
- atp.

Rozhodně nelze spoléhat na nějaké zdánlivé "výchozí řazení dle ID", které je nesmyslné (stačí, abych přidal vhodný index do DB a pořadí se změní). Pokud není řazení v zadání explicitně určeno, použiju nějaké přirozené řazení vyplývající ze situace (např. ono "výchozí" dle Created nebo ID, ale záměrně, ne omylem)

## Řazení v GridViews

Není-li výslovně stanoveno jinak nebo není-li to nepřiměřeně složité, potom vždy umožňujeme řazení podle všech sloupců GridView. Nezapomenout nastavit `DefaultSortExpression` na gridu, pokud nejsou data předsortěna (a i tak zvážit, jestli to nezmaří otočení směru řazení na příslušném sloupci, podle něž je předsortěno).

## DefaultButton, DefaultFocus

Na místech, kde zvyklosti webových UI předurčují, musí být použito `DefaultButton` a `DefaultFocus`, použití viz též [článek HAVIT KnowledgeBase](#).

Zejména:

- **Login-page** musí používat `DefaultButton` (tlačítko Přihlásit) i `DefaultFocus` (pole Username)
- **Search Box** v hlavičce stránky, ale i jinde musí mít `DefaultButton` způsobující vyhledávání
- **Filtr** by měl mít `DefaultButton` na své tlačítko *Apply*

## Položky číselníků / DropDownListů

- není-li výslovně uvedeno jinak (např. v pravidlech českého pravopisu!), potom textové hodnoty začínáme malým písmenem, např. "nerozhoduje-", "neuhrazeno", "aktivní", "modrá", atp.
- není-li výslovně uvedeno jinak, potom pro neutrální kritérium DropDownListu používáme text "-nerozhoduje-" (v angličtině "-any-")
- zpravidla se vyhýbáme neurčitému literálu "----"
  - ve filtrech má být "-nerozhoduje-", resp. "-any-"
  - v povinných polích, pokud tam vůbec NULL máme, tak je potřeba tam dát obvykle něco jako "-vyberte-"
  - jinde bychom měli vyjádřit význam NULL textem ala "-nenastaveno-", "-zdeděno-", "-není-", atp.
  - opravdu výjimečně je přípustné nechat "----"

## Title / Nadpis

- Každá stránka by měla mít rozlišující Title a Nadpis (pokud je v grafice použit), tak aby bylo možno snadno identifikovat stránku při více otevřených oknech/záložkách.
  - `EditPage` v Title a Nadpisu obvykle obsahuje hlavní uživatelskou identifikaci záznamu, např. `EditedSubjekt.Nazev`, `EditedFaktura.Cislo`, atp. (pozor na případ nového záznamu, i tam je potřeba srozumitelné Title/Nadpis, např. "Nový subjekt")
- obzvláštní péči je potřeba věnovat Title u webových stránek indexovaných vyhledávači (Google)

## AutoFill/AutoComplete na standardních input-fieldech

I když je podpora zatím dost bídná a standardizace v plenkách (ale už se to rýsuje - [WHATWG Autofilling form controls](#)), tak je fajn, pokud uživateli usnadníme vyplnění standardních formulářových polí využitím funkce `AutoFill`.

Podpora je v tomto dosti roztržštěná:

- **ASP.NET WebForms** a jeho `TextBox.AutoCompleteType` funguje v Internet Exploreru na základě `VCARD_NAME` atributu.
- Chrome pracuje s typed autocomplete hodnotami (tj. tak, jak bude nejspíš vypadat standardizace)
- FireFox zdá se používá vlastní nedokumentovanou logiku založenou na ID/name fieldu a hodnotě příslušného labelu.

Přistupujme k tomu tedy zatím individuálně. Přínejmenším použitím `TextBox.AutoCompleteType` ve WebForms žádné úsilí nestojí a můžeme si to pak centrálně doladit dle potřeby.