VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE

FAKULTA INFORMATIKY A STATISTIKY

VYŠŠÍ ODBORNÁ ŠKOLA INFORMAČNÍCH SLUŽEB

REFAKTORING SOFTWAROVÉ APLIKACE SCIODAT

Bakalářská práce

Vedoucí bakalářské práce: PhDr. Kateřina Julišová

Praha, 2006

Martin Arnhold

Prohlášení

Prohlašuji, že jsem bakalářskou práci na téma „Refaktoring softwarové aplikace ScioDat“ zpracoval samostatně a použil pouze zdrojů, které cituji a uvádím v seznamu použité literatury.

V Praze dne 23. dubna 2006

.....

Martin Arnhold

Poděkování

Děkuji tímto vedoucí své bakalářské práce, paní Ing. Kateřině Julišové, nejenom za vedení, cenné rady a podnětné připomínky při tvorbě této práce, ale také za obětavý přístup k nám studentům.

Obsah

1. Anotace

V této bakalářské práci, se zabývám problematikou refaktorování, jelikož je to oblast, se kterou jsem přišel v současném zaměstnání do styku a bez které si již řádný vývoj aplikací nedokáži představit.

Postupně vysvětluji jeho principy a snažím se zodpovědět otázky, proč a kdy refaktorovat, jaké jsou s refaktorováním spojené problémy a jak refaktorování vzniklo. Co se týče struktury této práce, vycházím rámcově ze struktury hlavního zdroje, publikace Martina Fowlera s názvem Refaktorování – Zlepšení existujícího kódu.

V první části práce, se věnuji problematice refaktorování obecně. Nejprve vysvětluji, co to refaktorování vůbec je, uvádím několik definic z odborné literatury a zdůvodňuji výběr tohoto tématu pro zpracování bakalářské práce. V dalších částech práce, pokračuji popisem principů refaktorování, představením a popsáním tzv. code smells (pachy v kódu), jež představují takové konstrukce v něm obsažené, které vypovídají o špatném návrhu a snažím se poskytnout určitá vodítka pro to, aby bylo poznat, které problémy je možné refaktorováním řešit. V neposlední řadě jsem tuto práci rozšířil o možnost refaktorování pomocí automatických nástrojů a jeden z nich, který v současnosti používám, popisuji blíže.

Většinu nejpoužívanějších technik a postupů pak demonstruji na reálné aplikaci ScioDat, která slouží k evidenci a analýze údajů o výsledcích testování studijních výsledků a studijních předpokladů žáků základních a středních škol.

2. Úvod

Pojem „refaktoring“ se poprvé začal používat mezi programátory ve Smalltalku, ale jelikož se používá převážně v souvislosti s objektově orientovaným programováním, tak se velice rychle rozšířil i mezi ostatní programátory. Refaktorovaní je proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu. Tento proces použijeme například ve snaze přesunout určitou část kódu do třídy, kde bude umístění tohoto kódu logičtější.

Jak už jsem uvedl, samotná refaktorizace obvykle nemění funkčnost kódu, ale pouze zlepšuje návrh poté, co byl napsán. Můžeme říci, že pouze třídíme a systematicky řadíme části kódu takovým způsobem, aby lépe odpovídaly své logice, abychom si ulehčili pozdější vývoj aplikace, abychom se vyhnuli duplicitám kódu a aby kódu nerozuměl pouze překladač, ale i člověk, programátor, který se zapojí do vývoje aplikace v pokročilejším stádiu a nemusel nad kódem přemýšlet, ale pouze ho číst a rozumět mu. Zlepšujeme jeho průhlednost a jasnost. Při refaktorizaci tak může dojít například k rozdělení jedné třídy na víc tříd, nebo naopak k slučování tříd. U některých tříd může být vypozorován společný základ a může vzniknout společná bázová třída, ze které jsou nové verze původních tříd odvozeny, a podobně.

Úspěch refaktorování spočívá i v tom, že se můžeme chopit i špatného návrhu a přepracovat jej v kód dobře čitelný. Důležitou podmínkou je však postupovat v krátkých, jasně definovaných krocích, čímž se vyhneme zbytečným chybám, a musíme jasně oddělovat situace, kdy zrovna programujeme a kdy refaktorujeme.

Refaktorizace patří ke klíčovým postupům jedné z nových metodik vývoje softwarových systémů, která je známa jako „eXtreme Programming“ (dále XP) [1]. A protože každá změna — jakkoliv dobře míněná — může do programu vnést chyby, patří ke klíčovým postupům v XP i psaní testů známých jako unit tests (izolované testy funkčnosti pokud možno všech částí kódu) a acceptance tests (tedy testy přijatelnosti, které testují požadovanou funkčnost z pohledu uživatele). Testy se píší před implementací vlastního kódu, během ní i dodatečně. Pokud kód projde testy i po refaktorizaci, pak při ní s vysokou pravděpodobností nebyly do kódu zavlečeny

nechtěné chyby. I v testech může být skrytá chyba, ale styl jejich psaní se obvykle liší od psaní kódu, který testují. V tom je skrytá jistá záruka, že chyby v testech budou odhaleny dobře implementovaným kódem a chybně implementovaný kód bude naopak odhalen testy. Pokud je přesto zjištěna chyba v aplikaci, postupuje se tak, že se nejdříve doplní test, který tuto chybu odhaluje. Na základě tohoto testu pak můžeme poznat, zda se nám chybu v implementaci podařilo opravit. Důležité je, že pro budoucí refaktorizace takto postupně budujeme záchrannou síť testů, která je stále spolehlivější. Důvěryhodnost kódu roste a současně odpadají obavy o to, že zásahy do fungujícího kódu způsobí jeho nefunkčnost.

Klasickým odkazem na refaktorování je kniha Martina Fowlera [2]. Ačkoliv úprava kódu existuje již dlouhá léta, tak prvním známým dokumentem, který zkoumá a popisuje refaktoring, je disertační práce z roku 1993, jejímž autorem je William Opdyke [3]. Oba tyto zdroje poskytují katalog obvyklých metod pro refaktorování, popis jak tyto metody aplikovat a ukazatele, jak poznat, kdy metody použít a kdy ne.

3. Definice refaktorování

Slovo refaktorování má podle kontextu dvě definice – první definice má jmennou formu a druhá má formu slovesnou.

REFAKTOROVÁNÍ (podstatné jméno):
Změna ve vnitřní struktuře softwaru, vedoucí k jeho snazšímu pochopení a usnadnění dalších úprav beze změny jeho vnějšího chování.

REFAKTOROVAT (sloveso):
Změnit strukturu softwaru pomocí řady refaktorování bez změny jeho vnějšího chování.

Refaktorování nám nabízí postupy, jak efektivně a cíleně čistit zdrojový kód. Hlavním smyslem refaktorování je tedy zlepšit srozumitelnost kódu a tím i možnost dalších změn softwaru. V programu můžeme provést mnoho změn, které nemění jeho chování. Za refaktorování však považujeme pouze takové změny, které zlepší jeho vnitřní strukturu.

Typickým protikladem je optimalizace výkonu. Podobně jako refaktorování, tak i optimalizace výkonnosti nemění chování programu (kromě rychlosti). Její účel je však jiný. Výsledkem optimalizace bývá nejen větší rychlost, ale i větší složitost programu, což je pravý opak účelu refaktorování.

4. Principy refaktorování

4.1. Dva klobouky

Jak už jsem řekl v úvodu, je důležité rozlišovat, zda-li refaktorujeme nebo přidáváme novou funkcionalitu. Pokud přidáváme novou funkcionalitu, neměli bychom měnit stávající kód, ale pouze přidávat nové schopnosti. Tento proces můžeme označit jako tzv. *střídání klobouků*. Během celého vývoje střídá programátor tyto klobouky velmi často. Pokud například zjistí, že by bylo snazší implementovat novou metodu po předchozí úpravě stávajícího kódu, pak „nasadí klobouk refaktorování“, upraví stávající kód, opět „vystřídá klobouk“ a implementuje novou metodu. Důležité však je stále vědět, kterou činností se právě zabývá.

4.2. Výhody refaktorování

Refaktorování nedokáže vyřešit všechny problémy, které souvisí s vývojem softwaru. Může se však stát prostředkem, který nám umožní rychlejší vývoj a levnější údržbu softwaru. Následující kapitoly popisují hlavní výhody refaktorování.

4.2.1. Zlepšení návrhu kódu

Postupem času ztrácí program přidáváním nových a nových metod svou srozumitelnost. Pochopení kódu je pak čím dál více těžší, až se v něm ztratí programátor úplně. Refaktorování je způsob, jakým těmto problémům předejít a strukturu programu zjednodušit. Pokud je program špatně napsaný, například obsahuje duplicitní kód, je velmi obtížné ho dále rozšiřovat a je velká šance, že přibude dalších duplicit. Odstraněním těchto nedostatků nezrychlíme samotný program, ale zvýšíme jeho čitelnost a tím i celý vývoj.

4.2.2. Zlepšení srozumitelnosti kódu

Pokud píšeme nějaký kód, pak proto, abychom donutili vykonat počítač to, co potřebujeme. V průběhu vývoje, se celý program zvětšuje a reakce počítače stále více odpovídá našim konečným představám. Bohužel se často dostaneme do situací, kdy už si nepamatujeme, jakou má určitá část kódu plnit funkci. Z toho vyplývá, že není vždy nutné, aby počítač zpracoval určitou část kódu rychleji, na úkor jeho srozumitelnosti.

Když si uvědomíme, že my sami nerozumíme kódu, který jsme napsali, co teprve programátor, který převezme projekt po nás. Místo několika hodin či dnů, pak stráví několik týdnu nebo dokonce měsíců, jenom aby kód pochopil. Refaktorování pak tedy pomáhá zvýšit čitelnost kódu. Na začátku máte kód, který funguje, ale nemá ideální strukturu. Po chvíli refaktorování, vyjadřuje kód svůj smysl lépe.

4.2.3. Snazší hledání chyb

Když refaktorujeme kód, pochopíme hlouběji jeho smysl. Ujasněním struktury programu si také ujasníme své domněnky o programu do té míry, že si nakonec všimneme i jeho chyb.

4.2.4. Zrychlení programování

Dobrý návrh je základem pro rychlý vývoj softwaru. Bez dobrého návrhu sice budeme zpočátku postupovat rychle, ale postupem času nás tento špatný návrh velice zpomalí. Budeme se muset zdržovat opravami špatného kódu, odstraňováním duplicit a porozuměním dlouhých metod. Dobrý návrh je tedy základem pro udržení rychlosti vývoje softwaru.

4.3. Nevýhody refaktorování

Refaktorování má samozřejmě i své nevýhody, které jsou však ve většině případech vykompenzovány jeho výhodami nebo možností jejich odstranění.

4.3.1. Žádné nové funkce

Refaktorování nepřidává do programu žádné nové funkce a programátoři jsou placeni právě za přidávání těchto nových funkcí. Tato nevýhoda je však vyvážena rychlejším vývojem a lehčí udržovatelností softwaru s lepším designem.

4.3.2. Výskyt chyb

Refaktorováním mohou vzniknout chyby, které pak pokazí správný chod programu. V takovém případě je třeba rozdělit refaktorování do menších kroků a po každém z nich otestovat funkčnost programu.

4.3.3. Zkušenosti programátora

Vyhledání „páchnoucích“ částí (kapitola 4) je proces, který záleží na zkušenostech a intuicích programátora. Když není nikde přesně určené, jak vypadá správný design a jak vypadá špatný design, můžou různí programátoři zhodnotit tu samou část kódu úplně jinak.

4.3.4. Rychlost programu

Při refaktorování dochází ke změnám zdrojového kódu programu a tím obvykle i ke změně jeho výkonnosti. Ve většině případech platí nepřímá úměrnost: Čím je program efektivnější, tím je jeho kód hůře srozumitelný pro člověka a naopak.

4.4. Kdy refaktorovat

Refaktorování není činnost, pro kterou musíme plánovat zvláštní čas. Refaktorujeme proto, že chceme dělat něco jiného a refaktorování nám v tom pomůže.

4.4.1. Refaktoring před přidáním funkcionality

Nejčastěji refaktorujeme před přidáním nové funkcionality. Hlavním důvodem obvykle bývá snaha pochopit kód, který potřebujeme změnit. Kdykoliv musíme přemýšlet nad smyslem nějaké části kódu, je lepší se zastavit a popřemýšlet, zda-li by se kód nedal refaktorováním převést do srozumitelnější podoby. Dalším důvodem je existující návrh, který nám neumožňuje snadno přidat novou funkcionalitu. Po refaktorování je možné přidat novou funkcionalitu rychleji a snáz.

4.4.2. Refaktoring při revizi kódu

Většina organizaci pravidelně reviduje svoje kódy. Revize kódu pomáhají rozšiřovat znalosti v rámci vývojového týmu. Revize jsou velmi důležité i pro psaní srozumitelného kódu. Refaktorování nám pak pomáhá při revizích cizího kódu. Refaktorování také vede k mnohem konkrétnějším výsledkům revizí kódu. Nevycházejí z nich už jen připomínky, ale často přímo jejich okamžité implementace. Revize kódu pak má mnohem efektivnější výsledky.

4.5. Proč refaktorovat

[Kent Beck][2]
Programy mají dvojí hodnotu: to, co dělají nyní, a to, co mohou udělat v budoucnu. Při programování jsme většinou soustředěni na to, co od programu chceme dnes. Ať už opravujeme chybu nebo přidáváme novou funkci, zvyšujeme momentální hodnotu programu zlepšením jeho schopnosti.

Při programování si však brzy uvědomíte, že nejde jen o to, co systém dokáže dnes. Pokud se vám dnes daří zvládnout požadovanou práci tak, že zítřejší práci již zítra nezvládnete, jste na tom vlastně špatně. Všimněte si ale, že sice víte, co musíte udělat dnes, ale nejste si už tak úplně jisti co bude třeba udělat zítra.

Možná to, možná ono, možná něco, co jste si vůbec nedokázali představit. Vím dost, abych mohl udělat dnešní práci. Nevím ale dost pro zítřejší. Když ale pracuji jen pro dnešek, nebudu zítra schopen pracovat vůbec.

Refaktorování je jedním ze způsobů, jak z toho ven. Když zjistíte, že včerejší rozhodnutí již dnes nedává smysl, změníte ho a dnešní práci můžete udělat. Zítra se však i dnešní způsob chápání ukáže jako naivní, a tak jej změníte také. Refaktorování je tedy proces, ve kterém vezmete existující program a zvýšíte jeho hodnotu nikoliv změnou jeho chování, ale přidáním uvedených vlastností, které umožňují pokračovat v rychlém vývoji.

5. Pachy v kódu

Umět se správně rozhodnout, kdy začít s refaktorováním a kdy ho ukončit, je stejně důležité, jako znát jeho principy. V této kapitole se snažím popsat několik vodítek, které nám pomáhají poznat problémy, které se dají řešit refaktorováním. Na závěr pak ke každému „pachu“ uvádím seznam metod, které je vhodné použít pro jeho odstranění. Seznam příznaků je převzat volně podle knihy „Refaktoring – Zlepšení existujícího kódu“ [2]

5.1. Příznaky

- Duplicitní kód

Najdeme-li na více místech stejnou strukturu programu, je jisté, že program zlepšíme, pokud se nám jej podaří sjednotit.

- Dlouhá metoda

Čím je procedura delší, tím je obtížnější ji pochopit. Zkracováním metod dospějeme mnohem větší srozumitelnosti kódu. Hlavním pravidlem je: Pokud cítíme potřebu něco komentovat, napíšeme místo toho metodu. Komentáře jsou tedy dobrým vodítkem k rozpoznání kódu, který si zaslouží vyjmutí a vytvoření nové metody.

- Velká třída

Pokud třída dělá příliš mnoho věcí, obvykle se to projevuje velkým množstvím instančních proměnných a s tím se obvykle vyskytuje i duplicitní kód.

- Dlouhý seznam parametrů

V objektově orientovaných programech již není nutné předávat metodám všechno v parametrech. Mnoho věcí, které potřebuje, je k dispozici v její vlastní třídě a metoda si tak může požádat objekt o určitý údaj.

- Protichůdné změny

K protichůdným změnám dochází, pokud změnou kódu v jedné situaci znemožním použití tohoto kódu v situaci jiné. V tomto případě je lepší vytvořit dva objekty místo jednoho.

- Rozptýlené úpravy

Rozptýlené úpravy jsou ve své podstatě protichůdné změny, ale obráceně. Pocítíme je například, když při každé změně musíme provést řadu drobných úprav v mnoha třídách.

- Chybějící schopnosti

Základní princip objektů spočívá ve spojení dat s procesy, které na nich probíhají. Metoda, která se mnohem víc zajímá o jinou třídu, by měla být přesunuta právě do této třídy.

- Datové shluky

Často najdeme tři nebo čtyři datové položky společně na různých místech: pole v několika třídách, parametry v mnoha metodách. Skupiny dat, které se drží pohromadě, si zaslouží vlastní objekt.

- Příkazy switch

Potíž s tímto příkazem je především v duplicitě. Často najdeme tyto příkazy na mnoha místech programu. Když potom chceme přidat novu větev, musíme najít a změnit všechny výskyty příkazu. Řešení se nám nabízí v podobě polymorfie.

- Paralelní hierarchie dědičnosti

Problém nastává, pokud při každém vytvoření podtřídy z jedné třídy, musíme vytvářet podtřídu i pro jinou třídu. Takovéto duplicity se můžeme zbavit způsobem, kdy se instance v jedné hierarchii odkazují na instance v druhé hierarchii.

- Líná třída

Jako línou třídu označujeme třídu, která nedělá dost, aby se vyplatila, a proto by měla být vhodným způsobem odstraněna. Často to bývá třída, která o svou funkci přišla při refaktorování.

- Spekulativní obecnost

Na tento „pach“ narazíme, pokud do kódu vkládáme možnosti, o kterých si říkáme, že budou někdy potřeba, ale v současné chvíli pro ně není žádné využití. Výsledkem je pak

složitější údržba a nesrozumitelnost. Tyto nástroje pouze překáží a je potřeba se jich zbavit.

- Dočasná položka

S dočasnou položkou se setkáme v objektu, jehož instanční proměnná je nastavovaná jen za určitých okolností. Takový kód je pak těžké pochopit, protože se snažíme pochopit, proč je tam položka, která není používaná.

- Zřetězené zprávy

Tyto zprávy najdeme tam, kde klient žádá objekt o jiný objekt, po kterém pak chce další objekt, od něj další objekt atd. Pak vzniká dlouhý řádek navazujících metod *get* nebo sekvence dočasných proměnných. Klient je pak provázán strukturou objektů a jakákoliv jeho změna znamená nutnost měnit i klienta.

- Prostředník

Zapouzdření je jednou ze základních vlastností objektů. Se zapouzdřením pak často souvisí delegování. Pokud ale při pohledu na rozhraní třídy najednou zjistíme, že polovina metod deleguje zprávy na jinou třídu, je vhodné odstranit prostředníka a komunikovat přímo s objektem, který ví co se opravdu děje.

- Nevhodná důvěrnost

Pokud se některé třídy chovají navzájem příliš důvěrně a tráví mnoho času v soukromých oblastech svých protějšků, je načase je oddělit. K přílišné důvěrnosti často vede dědičnost, kdy podtřídy o svých rodičích vědí více, než by rodičům bylo milé.

- Datová třída

Datové třídy jsou jednoduchá uložiště dat a ostatní třídy do nich hluboko zasahují. Zpočátku mají tyto třídy většinou veřejné položky a proto je dobré, tyto položky okamžitě zapouzdřit.

- Komentáře

Komentáře často signalizují špatný kód. Pokud narazíme na komentář, je vhodné z daného úseku kódu vytvořit metodu a pojmenovat ji takovým způsobem, aby byl její název samovysvětlující.

5.2. Metody refaktorování

Tato tabulka slouží jako inspirace, když si nejsme jisti, jaké refaktorování použít. Každý řádek obsahuje druh „zápachu“ a seznam metod, které je vhodné použít. U každé metody je pak uvedeno číslo stránky z knihy Refaktoring – Zlepšení existujícího kódu [2], na které se můžeme o každé metodě dozvědět více informací.

Zápach	**Obvyklá refaktorování**
Duplicitní kód	Vyjmout metodu (121) Vyjmout třídu (153) Přesunout metodu výš (298) Vytvořit šablonovou metodu (317)
Dlouhá metoda	Vyjmout metodu (121) Nahradit dočasnou proměnnou dotazem (129) Nahradit metodu objektem metody (142) Rozložit podmínku (227)
Velká třída	Vyjmout třídu (153) Vyjmout podtřídu (305) Vyjmout rozhraní (314) Nahradit datovou položku objektem (175)
Dlouhý seznam parametrů	Nahradit parametr metodou (272) Zavést objekt pro parametry (275) Zachovat celý objekt (269)
Protichůdné změny	Vyjmout třídu (153)
Rozptýlené úpravy	Přesunout metodu (146) Přesunout položku (150) Vložit třídu (157)
Chybějící schopnosti	Přesunout metodu (146) Přesunout položku (150) Vyjmout metodu (121)
Datové shluky	Vyjmout třídu (153) Zavést objekt pro parametry (275) Zachovat celý objekt (269)
Příkazy switch	Nahradit podmínku polymorfizmem (241) Nahradit kód typu podtřídami (216) Nahradit kód typu stavem nebo strategií (219) Nahradit parametr explicit. metodami (266) Zavést objekt *null* (245)

Paralelní hierarchie dědičnosti	Přesunout metodu (146) Přesunout položku (150)
Lína třída	Vložit třídu (157) Zrušit hierarchii (316)
Spekulativní obecnost	Zrušit hierarchii (316) Vložit třídu (157) Odstranit parametr (260) Přejmenovat metodu (256)
Dočasná položka	Vyjmout třídu (153) Zavést objekt *null* (245)
Zřetězené zprávy	Skrýt delegáta (159)
Prostředník	Odstranit prostředníka (162) Vložit metodu (127) Nahradit delegování dědičností (327)
Nevhodná důvěrnost	Přesunout metodu (146) Přesunout položku (150) Změnit obousměrné propojení na jednosměrné (197) Nahradit dědičnost delegováním (324) Skrýt delegáta (159)
Datová třída	Přesunout metodu (146) Zapouzdřit položku (202) Zapouzdřit kontejner (203)
Komentáře	Vyjmout metodu (121) Zavést předpoklad (252)

6. Refaktorování a rychlost

Při refaktorování provádíme změny, které zlepšují srozumitelnost, na druhou stranu, tyto změny často způsobí, že výsledný program pracuje pomaleji. Na rychlosti je zajímavé to, že program při svém běhu, stráví nejvíce času v malé části kódu. Pokud tedy optimalizujeme celý program rovnoměrně, pak 90% optimalizací přijde nazmar, protože tento kód není vykonávaný často. Refaktorování nám pak nabízí spolehlivé řešení. Nejprve píšeme program tak, aby byl dobře čitelný, bez ohledu na jeho rychlost. Teprve na závěr přistoupíme k optimalizaci, ve které vyladíme rychlost programu. Pomocí profileru sledujeme jeho průběh a zaznamenáváme, ve které části kódu stráví program nejvíce času. Díky refaktorování je tato část kódu poměrně malá a je mnohem snazší ji vyladit na potřebný výkon.

Během refaktorování je tedy software dočasně pomalejší, ale výsledný kód jde mnohem lépe optimalizovat a konečný výsledek je mnohem rychlejší.

7. Problémy s refaktorováním

Učíme-li se novou techniku, která výrazně zvýší naší produktivitu, jen těžko si všimneme, kdy se nehodí. Učíme se ji často na jediném projektu a v určitém kontextu. Je tedy složité určit, kdy je daná technika méně efektivní a kdy dokonce může škodit.

To stejné platí i pro refaktorování. Víme, že do naší práce může vnést citelné zlepšení, ale nemáme s ním dostatečné zkušenosti, abychom poznali, kdy skončit a dále nerefaktorovat.

7.1. Databáze

Jednou z problematických oblastí refaktorování jsou databáze. Většina aplikací má základ postaven na určité struktuře databáze. To je jeden z důvodů, proč je databázi těžké změnit. Další důvod je migrace dat. I když je systém rozdělen do vrstev, tak změna datového schématu vyžaduje migraci dat, což může být dlouhý a obtížný proces.

7.2. Změny rozhraní

Na refaktorování je nepříjemné, že řada refaktorování mění rozhraní. I nejčastější refaktorování ***přejmenovat metodu*** je změna rozhraní. Pokud máme přístup k celému kódu, který metodu používá, není přejmenování metody problém. Pokud je ale rozhraní použité v kódu, který nemůžeme změnit, nastává problém. Jedná se o tzv. publikovaná rozhraní, která nemůžeme bezpečně měnit, protože nemůžeme zajistit snadnou změnu jeho volání. V této situaci musíme zachovat starou i novou verzi rozhraní. Funkci původního rozhraní obvykle můžeme zachovat tím, že staré rozhraní volá metody rozhraní nového.

7.3. Kdy se refaktorovat nevyplatí

Jednou z podmínek proveditelnosti refaktoringu je, že by měl kód, před samotným refaktorováním, fungovat víceméně správně. Je-li existující kód v takovém nepořádku, že by bylo jednodušší, napsat ho celý znovu, je lepší se do refaktoringu vůbec nepouštět. Signálem pro takový postup je, že program prostě nefunguje a je velmi obtížné ho stabilizovat.

Dalším případem, kdy je lepší refaktorování vynechat, je když máme krátce před termínem dokončení. Zvýšená produktivita, kterou by nám refaktorování přineslo, by se tak projevila až po dokončení projektu, a tedy pozdě.

8. Nástroje pro refaktorování

V počátcích, byla jednou z největších překážek refaktorování skutečnost, že neexistoval vhodný nástroj pro automatické refaktorování. V současné době se na trhu vyskytuje již mnohá řada těchto automatizovaných nástrojů, avšak stále platí, že i dnes provádíme velké množství refaktorování ručně.

8.1. Refaktorování s využitím nástroje

Hlavní výhodou refaktorování pomocí nástroje je skutečnost, že není třeba provádět žádné testy, pomocí kterých bychom zjistili, zda-li nedošlo k chybě. Pokud chceme například použít refaktorování – „Vyjmout metodu“, stačí pouze označit danou část kódu a z nabídky menu vybrat položku nazvanou *Vyjmout metodu (Extract Method)*. Nástroj poté sám rozhodne, zda je přípustné text vyjmout, sám vypočítá, kolik parametrů bude metodě předáno, vybídne nás, abychom zadali jméno metody a určili pořadí předávaných parametrů. Poté nástroj vyjme kód z původní metody a nahradí jej voláním, které se odkazuje na refaktorováním nově vzniklou metodu.

Jak už jsem poznamenal na začátku, má refaktorování pomocí nástrojů velký vliv na testování. Vždy budou existovat refaktorování, která nebude možné automatizovat, ale na druhou stranu, je velká řada refaktorování, která jsou prováděna automaticky, a je tedy třeba testovat mnohem méně.

8.2. Požadavky na refaktorovací nástroj

Jedním ze základních požadavků, je schopnost vyhledávat v celém programu nejrůznější entity, např. nalézt všechna volání pro určitou metodu, nebo nalézt pro určitou instanční proměnou všechny metody, které ji čtou nebo do ní zapisují. Programátor pak může nechat vyhledat odkazy na jakýkoliv prvek programu, což je umožněno dynamickým překladem kódu.

Dalším požadavkem je přesnost. Refaktorování, která nástroj implementuje, musí zachovávat chování programu. Zachovat toto chování absolutně je ale někdy nemožné. Pokud například refaktorování program o několik milisekund zrychlí nebo zpomalí, pak

v programech, na které jsou kladené vysoké nároky na práci v reálném čase, může tato změna způsobit nekorektní chování tohoto programu.

Rychlost je další klíčový požadavek, který by měl nástroj mít. Vždy je ale třeba vzájemně poměřit nároky na čas a na přesnost. Když se ukáže, že by byla analýza programu příliš pomalá, je jedním z řešení přesunout odpovědnost na programátora a požádat ho o potřebné informace.

Jednou z posledních vlastností je integrace tohoto nástroje do vývojového prostředí. Je prokázáno, že už jenom samotná integrace do vývojového prostředí, bez jakýchkoliv změn, zvyšuje několikanásobně používání automatizovaného nástroje.

Automatické nástroje pro refaktorování jsou nejlepším způsobem, jak kontrolovat narůstající projekt. Bez těchto nástrojů by se software začínal drolit a mohly by se v něm objevovat chyby.

8.3. Refaktorovací nástroj ReSharper (R#)

V této části představím dnes již jeden z mnoha refaktorovacích nástrojů – ReSharper. Tento nástroj jsem použil při refaktorování v praktických ukázkách, v druhé části této bakalářské práce.

8.3.1. ReSharper obecně

ReSharper je modul do vývojového prostředí VisualStudio .NET, který umožňuje refaktorovat zdrojový kód jazyka C#. Přestože ReSharper není samostatný software, nabízí dostatek zajímavých funkcí. K hlavním funkcím, kromě refaktorování, patří zvýraznění syntaktických chyb v kódu, bez nutnosti jeho kompilace a přehlednou navigaci ve zdrojovém kódu programu pomocí různých druhů zobrazení (barva textu, barva pozadí atd.). ReSharper verze 1.5 poskytuje celkem 17 refaktorovaní. Kromě těch často používaných, obsahuje také některé zajímavé, jako je například změna abstraktní třídy na rozhraní a naopak změna rozhraní na abstraktní třídu.

8.3.2. Seznam refaktorování v modulu ReSharper

ReSharper podporuje následující typy refaktorování:

- Přejmenovat symbol – metod, proměnných, atd. (*Rename Symbol*)
- Přesunout typ (*Move Type*)
- Kopírovat typ (*Copy Type*)
- Změnit metodu (*Change Method Signature*)
- Vyjmout metodu (*Extract Method*)
- Vyjmout typ do nového souboru (*Extract type to a new file*)
- Zavést proměnnou (*Introduce Variable*)
- Zavést položku (*Introduce Field*)
- Zavést parametr (*Introduce Parameter*)
- Zrušit proměnnou (*Inline Variable*)
- Vyjmout rozhraní (*Extract Interface*)
- Vyjmout rodičovskou třídu (*Extract Superclass*)
- Nahradit metodu vlastností (*Convert Method to Property*)
- Nahradit vlastnost metodou (*Convert Property to Method*)
- Nahradit abstraktní třídu rozhraním (*Convert Abstract Class to Interface*)
- Nahradit rozhraní abstraktní třídou (*Convert Interface to Abstract Class*)
- Zapouzdřit položku (*Encapsulate Field*)

9. Aplikace ScioDat

ScioDat je aplikace, pomocí které zasílají jednotlivé školy výsledky testování jednotlivých žáků do firemní databáze společnosti Scio k dalšímu zpracování. Odpovědi žáků škola přepisuje přímo z vyplněných testů nebo z jednoduchých pomocných listů do webového formuláře. Výsledky je možné zadávat z libovolného počtu počítačů (připojených k internetu) současně. Ihned po zadání údajů jsou škole k dispozici výsledky žáků i třídy. Program je připraven tak, aby pořízení výsledků co nejvíce usnadňoval, předcházel chybám a umožňoval snadnou kontrolu.

9.1. Výhody použití aplikace ScioDat

Zadávání výsledků přes webové rozhraní aplikace ScioDat má tři hlavní výhody:

- Ihned po zadání dat, má škola k dispozici výsledky žáků, je tedy možné předat žákům výsledky ještě týž den.
- Škola má zadávání dat zcela pod kontrolou, sama kontroluje či opravuje chyby, může dodatečně doplnit výsledky absentujícího žáka, apod.
- Škola může po skončení testování provádět on-line analýzu výsledků (srovnávat se s jinými školami, porovnávat třídy, atd.).

9.2. Průběh používání aplikace ScioDat

Aplikace ScioDat je dostupná na adrese www.scio.cz/1/ScioDat. K přihlášení do aplikace potřebuje znát škola své uživatelské jméno a heslo. Oba dva údaje jsou doručeny škole společně s testy na úvodní straně instrukcí.

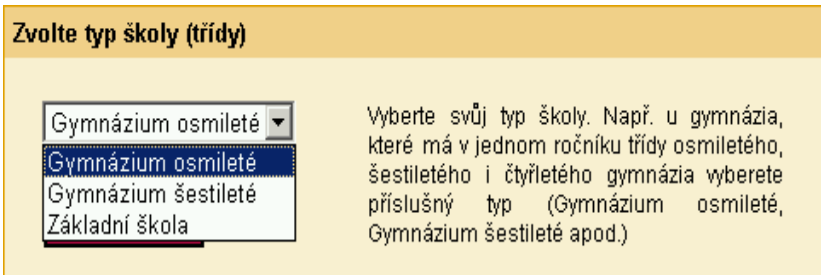
9.2.1. První přihlášení

Na stránce www.scio.cz/1/ScioDat zadá škola uživatelské jméno a heslo. Akci potvrdí tlačítkem „přihlásit se“. Dostane se na úvodní stránku, kde nalezne seznam dostupných testování. Škola vybere aktuální testování podle třídy – např. Testování STZŠ 2006 pro 8. ročník – Klíčové kompetence.

9.2.2. Zadávání tříd

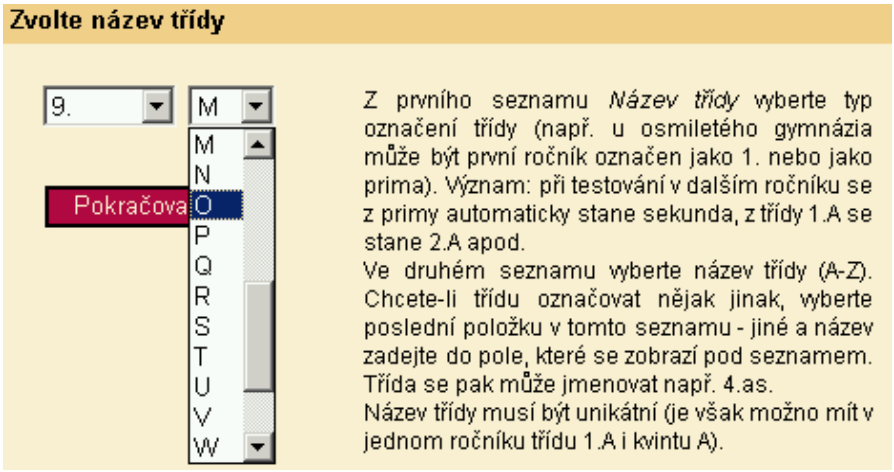
V tabulce „Existující třídy“ se zahájí zadávání nové třídy tlačítkem „Přidat novou třídu“. Pokud škola testujete více ročníků, pak každou třídu založí pouze v příslušném testování – tedy 8. třídu v Testování STZŠ 2006 pro 8. ročník – Klíčové kompetence atd.

Vytváření každé třídy školám ulehčuje podrobný průvodce. Nejprve musí zvolit typ školy pro danou třídu a potvrdit tlačítkem „Pokračovat“.



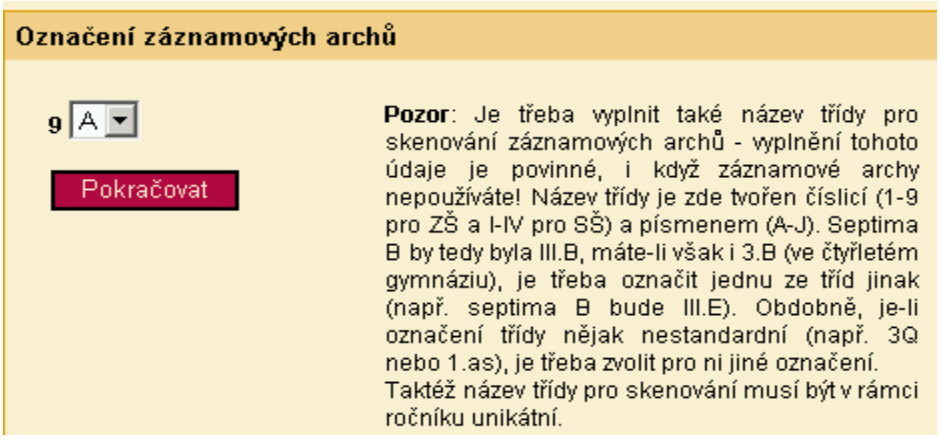
Obrázek 1: volba typu školy

Poté zvolí název třídy pomocí písmen A-Z dle přiložených instrukcí a potvrdí tlačítkem „Pokračovat“.



Obrázek 2: zadání názvu třídy

Ke každé třídě může škola zadat nepovinný údaj jako poznámku. Na závěr škola zvolí označení třídy pro záznamový arch pomocí písmen A – J dle přiložených instrukcí a potvrdí tlačítkem „Pokračovat“. V záznamovém archu, pokud jej škola používá, musí třídu vždy označovat tímto kódem. Škola má povinnost zadat toto označení, i když archy nepoužívá.



Obrázek 3: zadání názvu archu

Po vyplnění všech údajů se zobrazí kontrolní tabulka, kde si škola ověří všechny údaje o třídě, které zadala. Pokud nesouhlasí nebo chcete některé údaje změnit, klikne na odkaz „Upravit“ a údaje opraví. Jestliže jsou všechny údaje správně, potvrdí vše tlačítkem „Uložit třídu“.

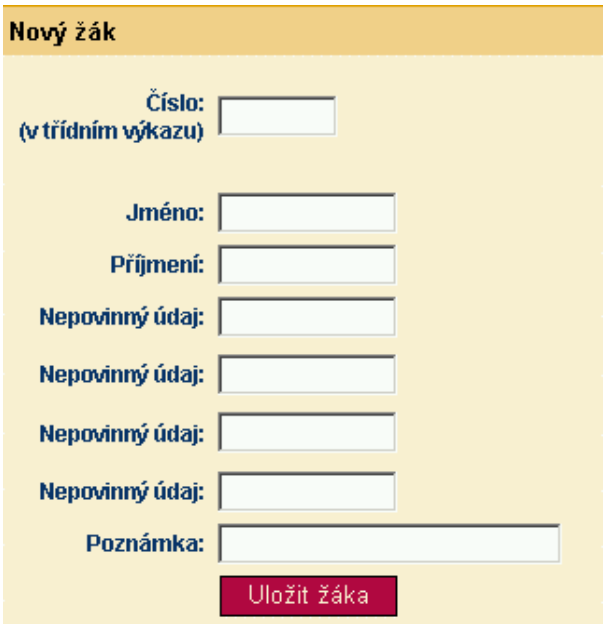
obrázek 4: ověření zadaných údajů

Po uložení se objeví nápis „Třída byla uložena!“. Stejným způsobem, tedy opakováním těchto kroků zadá škola všechny třídy, které se testování účastní. Jednotlivé uložené třídy může škola editovat i později. Odstraněna může být pouze ta třída, která ještě neobsahuje žádná data.

9.2.3. Zadávání žáků

Dříve než škola přistoupí k zadávání žáků, musí mít vytvořenou příslušnou třídu. Všechny vytvořené třídy se zobrazují v tabulce „Existující třídy“. Po kliknutí na název třídy je umožněno přidání nového žáka. Povinné položky jsou číslo žáka

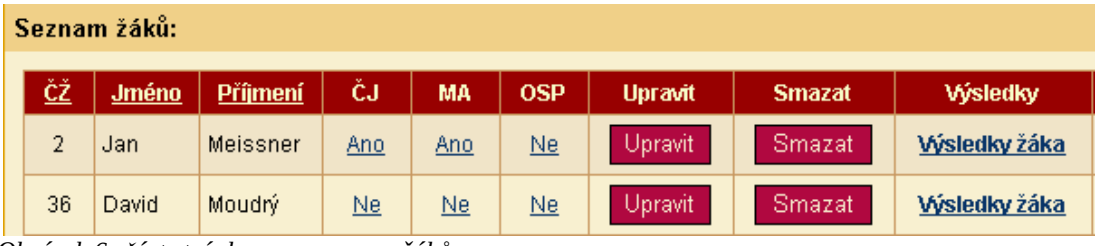
v třídním výkazu, příjmení a jméno. Kliknutím na tlačítko „Uložit žáka“ a po úspěšném uložení se objeví nápis „Žák byl uložen!“. Importovat žáky do třídy je taky možno hromadně, pomocí XML nebo CSV souboru.



Obrázek 5: editační formulář pro vytvoření žáka

9.2.4. Zadávání výsledků

Výsledky se zadávají vždy pro jedno testování. V rámci testování lze zadávat výsledky po předmětech nebo po žácích. U každého žáka se pod jednotlivými předměty zobrazuje Ano (pokud byly výsledky zadány a uloženy) nebo Ne (pokud výsledky nebyly zadány).



Obrázek 6: část stránky se seznamem žáků

Kliknutím na odkaz „Ne“ u každého žáka pod daným předmětem, se zobrazí formulář pro zadání výsledků. Do jednotlivých okének se píší písmena reprezentující vybrané odpovědi v tištěných testech. Pokud žák úlohu neřešil nebo není jednoznačná odpověď, vyplní se okénko pomlčkou “-“. K zadávání výsledků může být použita i numerická klávesnice, což může zadávání urychlit (číslo 1 je pak rovno odpovědi A,

číslo 2 odpovědi B atd.). Zadání výsledků každého testu u každého žáka se potvrdí kliknutím na tlačítko „Uložit odpovědi“.

Obrázek 7: formulář na odpovědi

9.2.5. Vyhodnocení

Přehled odpovědí u každé třídy se dá zjistit kliknutím na odkaz „Výsledky“. Výsledky jednotlivých žáků se zobrazí kliknutím na odkaz „Výsledky žáka“, který je na konci řádku. Kliknutím na Tiskové sestavy se zobrazí seznam žáků třídy a u nich dosažené skóre, čistá úspěšnost nebo hrubá úspěšnost, podle toho, který typ sestavy je zvolen. Všechny sestavy jsou vhodné pro tisk. Tyto výsledky jsou pouze předběžné. Kompletní výsledky jsou spolu s grafy a tabulkami součástí souhrnné zprávy, která je pak jednotlivým školám zasílána.

10. Výběr z použitých refaktorování

Většina refaktorování jsou pouze „jednoduché“ změny kódu, kdy názvy refaktorovacích metod naznačují i jejich účel. Skládají se z malých kroků, pomocí kterých se stává vzhled kódu čitelnějším. M. Fowler popsal ve své knize [2] více než 70 nejčastěji používaných refaktorování. V této části bakalářské práce, popíší některé z těchto metod, které jsem použil při refaktorování zdrojového kódu aplikace ScioDat.

Každý z těchto příkladů bude obsahovat **název** refaktorování (podkapitoly), krátké **shrnutí** situace, ve které je refaktorování potřeba, **postup** (seznam kroků v modulu ReSharper, pokud refaktorování podporuje, jinak ruční postup) a názorný **příklad** na aplikaci ScioDat.

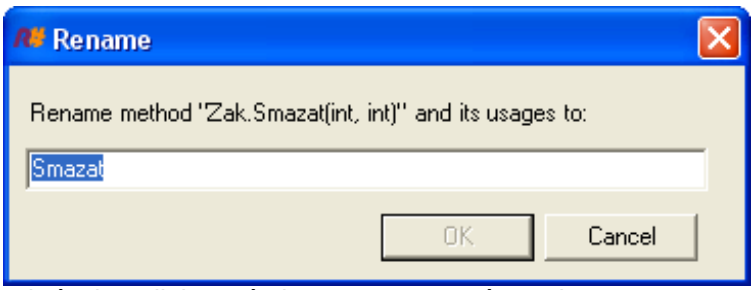
10.1. Přejmenovat symbol, metodu, proměnnou

- **Shrnutí**

  Toto refaktorování patří mezi nejjednodušší a nejvíce používané. Nejčastěji se jedná o změnu názvu proměnné, metody a nebo třídy. Hlavním cílem tohoto refaktorování je dosáhnout lepší čitelnosti a tím i srozumitelnosti zdrojového kódu. Dobrý kód by měl být „samovysvětlující“ a proto bychom měli tomuto refaktorování věnovat velkou pozornost.

- **Postup**

  Označíme část textu, kterou chceme přejmenovat a zmáčkneme „shift + F6“. Poté se nám otevře dialogové okno, do kterého zapíšeme nové jméno a potvrdíme. ReSharper pak automaticky změní jméno daného elementu (metody, proměnné, třídy) a zároveň všechna volání tohoto elementu.



Obrázek 8: dialogové okno pro změnu názvu elementu

- **Příklad**

```
public string Smazat(int z, int t) {
    string value = String.Empty;
    if (z > 0 && t > 0) {
        //cast kodu
    }
    else {
        //cast kodu
        value = "Žáka nelze smazat, chybné ID žáka!";
    }
    return value;
}
```

Obrázek 9: část programu, se špatně pojmenovanou metodou a parametry

Jak je vidět, tak srozumitelnost metody **Smazat,** stejně jako jejích parametrů, není příliš velká a proto je třeba jednotlivé části přejmenovat. Z této části kódu, není vůbec zřejmé, co tato metoda provádí, kromě toho, že něco maže.

```
public string SmazatZaka(int pZakID, int pTestovaniID)
{
    string lChybovaHlaska = String.Empty;
    if (pZakID > 0 && pTestovaniID > 0) {
        //cast kodu
    }
    else {
        //cast kodu
        lChybovaHlaska = "Žáka nelze smazat, chybné ID žáka!";
    }
    return lChybovaHlaska;
}
```

Obrázek 10: zdrojový kód po refaktorování

Po refaktorování je již jasné, že se jedná o mazání žáků a jako parametry jsou předávány ID žáka a ID testování, které má žák přidělen.

- **Poznámka**

  Při psaní kódu je dobré, když se programátor drží určitých programovacích standardů. Mě osobně se osvědčily tyto názvové konvence:

  - Používat český jazyk bez diakritiky + CamelCase
  - Metody tříd začínat velkým písmenem

- Fieldy začínat malým písmenem, pokud je třeba k fieldu přistupovat vždy přes property, pak field má před jménem znak podtržítka
- Properties začínat vždy velkým písmenem
- Jména lokálních proměnných začínat na „l“, jména globálních proměnných na „g“, jména parametrů začínat na „p“ a jména requestů začínat na „r“
- Konstanty psát velkým písmem s podtržítky jako oddělovači slov
- Ve slovesech používat infinitiv
- Přistupovat ke členům (member) třídy vždy přes *this*
- Raději napsat delší název než nejasný název
- Název by měl odpovídat funkci objektu

10.2. Nahradit magické číslo konstantou

- **Shrnutí**

  Magická čísla jsou čísla, která mají velmi konkrétní hodnotu, ale ta nám velmi často není vůbec zřejmá. Pokud používáme toto číslo na více místech programu, působí to velké potíže. Když je potřeba toto číslo náhle změnit, je poměrně zdlouhavé najít všechny výskyty tohoto čísla a nahradit je číslem novým. Pokud zapomeneme změnit jen jedno jediné z těchto čísel, pak program v jistých případech selže.

- **Postup**

  Nejprve deklarujeme konstantu (velkým písmem s podtržítky jako oddělovači slov) a nastavíme ji na hodnotu magického čísla. Poté vyhledáme všechny výskyty magického čísla a zaměníme je konstantou. Kód přeložíme a otestujeme.

- **Příklad**

```
private void grdVysledky_ItemDataBound(object sender, DataGridItemEventArgs e) {
    if (e.Item.ItemType != ListItemType.Header) {
        e.Item.Cells[7].Text =
                    Import.UpravitOdpovediVGridu(e.Item.Cells[7].Text);

        //cast kodu
    }
}
```

Obrázek 11: v kódu se vyskytuje magické číslo 7

V této metodě je jasně vidět nebezpečí používání magických čísel. Tato metoda nedělá nic jiného, než že při plnění datagridu daty, formátuje určitým způsobem texty ve sloupci odpovědi. Sloupec pro odpovědi je na 8 místě v datagridu – tedy index buňky v řádku je 7 (indexy počítány od nuly). Pokud bychom přidali do datagridu před sloupec odpovědí ještě nějaký nový sloupec, pak by se index buňky pro odpovědi změnil. Poté bychom museli najít všechny výskyty čísla 7 a nahradit jej za nové odpovídající číslo.

```
private const int ODPOVEDI = 7;

private void grdVysledky_ItemDataBound(object sender, DataGridItemEventArgs e) {
    if (e.Item.ItemType != ListItemType.Header) {
        e.Item.Cells[ODPOVEDI].Text =
                    Import.UpravitOdpovediVGridu(e.Item.Cells[ODPOVEDI].Text);

        //cast kodu
    }
}
```

Obrázek 12: nahrazení magického čísla 7 konstantou

Po nahrazení magického čísla konstantou již můžeme přidávat do datagridu libovolné sloupce a pokaždé stačí upravit pouze danou konstantu.

- **Poznámka**

  Než se pustíme do tohoto refaktorování, je dobré se podívat, jakým způsobem je magické číslo používáno. Pokud například vyjadřuje toto magické číslo délku pole, je lepší toto číslo nahradit výrazem `pole.length`.

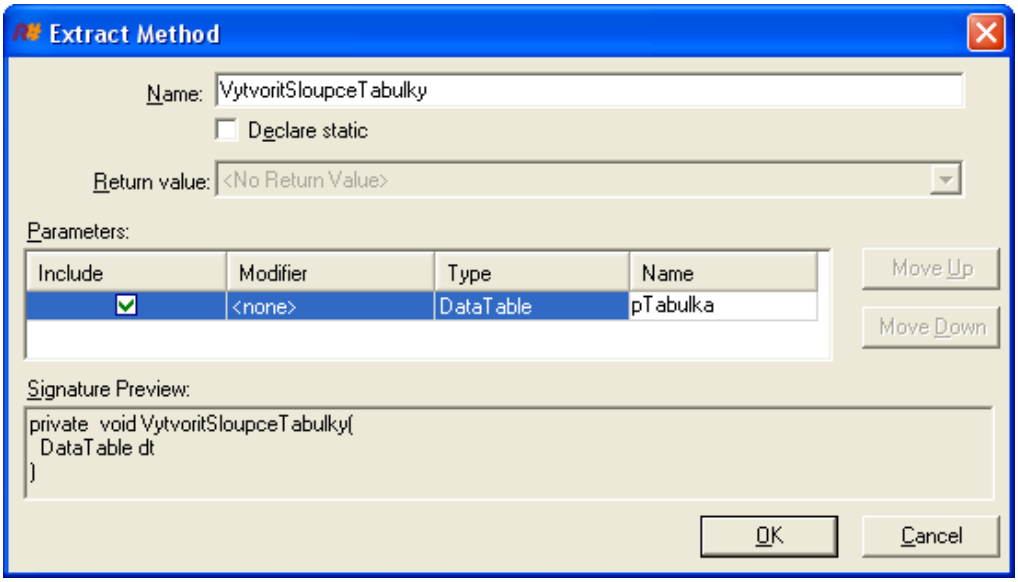
10.3. Vyjmout metodu

- **Shrnutí**

  Toto refaktorování je vhodné použít, pokud narazíme na metodu, která je příliš dlouhá, nebo na úsek kódu, u kterého je komentář. Cílem tohoto refaktorování je přesunout kus zdrojového kódu do samostatné metody tak, aby byla původní metoda čitelnější, a její obsah jsme mohli vnímat jako sekvenci komentářů.

- **Postup**

  Úsek kódu, z kterého chceme vytvořit metodu, označíme, z hlavní nabídky vybereme ReSharper-> Refactor-> „Extract Method...“ a otevře se nám následující dialogové okno.



Obrázek 13: dialogové okno pro vyjmutí metody

Do položky „Name“ napíšeme jméno nově vytvořené metody a v části „Parameters“ upravíme jména parametrů. Celou akci potvrdíme. ReSharper za nás vytvoří novou metodu, nazvanou „VytvoritSloupceTabulky“ a do původní metody vloží volání, na nově vzniklou metodu.

- **Příklad**

```
private void ZobrazitInformaceOTride() {
    //cast kodu

    DataTable dt = new DataTable();
    //vytvoreni sloupcu tabulky
    dt.Columns.Add(new DataColumn("Typ skoly", typeof (string)));
    dt.Columns.Add(new DataColumn("Nazev tridy", typeof (string)));
    dt.Columns.Add(new DataColumn("Poznamka", typeof (string)));
    dt.Columns.Add(new DataColumn("Oznaceni na ZA", typeof (string)));

    //cast kodu
}
```

Obrázek 14: část kódu, která obsahuje komentář

```
private void ZobrazitInformaceOTride()
{
    //cast kodu

    DataTable dt = new DataTable();
    this.VytvoritSloupceTabulky(dt);

    //cast kodu
}

private void VytvoritSloupceTabulky(DataTable pTabulka) {
    pTabulka.Columns.Add(new DataColumn("Typ skoly", typeof (string)));
    pTabulka.Columns.Add(new DataColumn("Nazev tridy", typeof (string)));
    pTabulka.Columns.Add(new DataColumn("Poznamka", typeof (string)));
    pTabulka.Columns.Add(new DataColumn("Oznaceni na ZA", typeof (string)));
}
```

Obrázek 15: výsledný kód po refaktorování

Jak je vidět, tak po vyjmutí části kódu, je původní metoda „ZobrazitInformaceOTride“ mnohem čitelnější. Pokud bychom prováděli toto refaktorování ručně (bez použití nástroje), musíme si dát pozor na předávání lokálních proměnných.

10.4. Vložit metodu

- **Shrnutí**

  Refaktorování „Vložit metodu“ je pravým opakem k „Vyjmout metodu“. Toto refaktorování přichází na řadu v situacích, kdy samotné tělo metody je stejně jasné jako její název. Tehdy je vhodné nahradit volání metody jejím kódem a metodu zrušit.

- **Postup**

  „Vložit metodu“ je jedním z refaktorování, které ReSharper nepodporuje, a proto je nutné ho provést ručně. Nejprve musíme zjistit, zda-li metoda není polymorfní. Pokud bychom zrušili polymorfní metodu, došlo by k chybě, protože podtřídy, které tuto metodu předefinovávají, by ji po odstranění nemohli předefinovat. Poté najdeme všechna volání této metody a každé z nich nahradíme tělem dané metody. Přeložíme, otestujeme a definici metody odstraníme.

- **Příklad**

```
private void BtnImportDotaznikuZobrazit() {
    string lDotaznikID = Ruzne.DB.SelectStr(String.Format(@"
                        SELECT dotaznikID
                        FROM tbTestovani
                        WHERE testovaniID = {0}"
                        , this.TestovaniID));
    if (JeTestovaniID129()) {
        this.btnImportDotazniku.Visible =  lDotaznikID != "" ? true : false;
    }
}

private bool JeTestovaniID129() {
    return this.TestovaniID == 129;
}
```

Obrázek 16: zbytečné použití metody v podmínce

```
private void BtnImportDotaznikuZobrazit() {
    string lDotaznikID = Ruzne.DB.SelectStr(String.Format(@"
                        SELECT dotaznikID
                        FROM tbTestovani
                        WHERE testovaniID = {0}"
                        , this.TestovaniID));
    if (this.TestovaniID == 129) {
        this.btnImportDotazniku.Visible =  lDotaznikID != "" ? true : false;
    }
}
```

Obrázek 17: nahrazení metody v podmínce jejím tělem a zrušení metody

10.5. Přesunout metodu

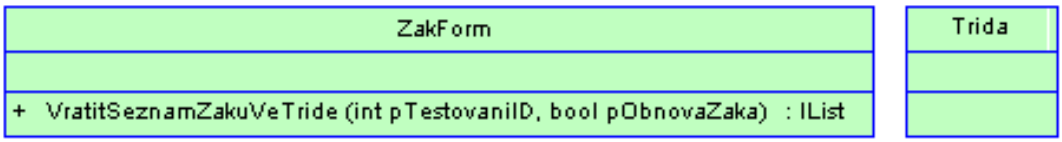
- **Shrnutí**

  Když se třídy starají o příliš moc věcí, nebo pokud jsou třídy příliš těsně provázány, je vhodné přesunout danou metodu do správné třídy. Aplikováním tohoto postupu snížíme vzájemnou provázanost mezi třídami.

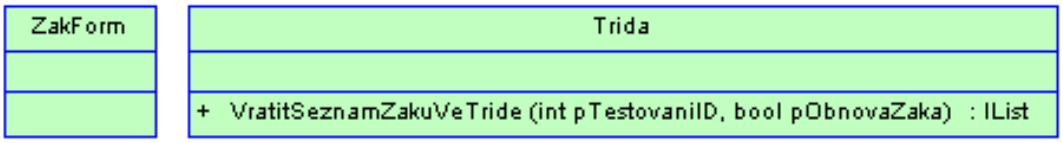
- **Postup**

  Nejprve zkontrolujeme potomky a předky zdrojové třídy, zda-li neobsahují jiné deklarace dané metody. Pokud ne, pak deklarujeme metodu v cílové třídě. Zkopírujeme kód ze zdrojové metody do cílové a upravíme metodu tak, aby v nové třídě fungovala. Kód přeložíme a otestujeme. Pokud je vše v pořádku, pak rozhodneme, jestli můžeme zdrojovou metodu odstranit nebo jestli je třeba ji ponechat jako delegující. Jestliže zdrojovou metodu odstraníme, pak nahradíme všechny původní odkazy novými odkazy na cílovou metodu. Kód přeložíme a opět otestujeme.

- **Příklad**



Obrázek 18: metoda "VratitSeznamZakuVeTride" je chybně umístěna ve třídě "ZakForm"



Obrázek 19: po refaktorování je metoda přesunuta do třídy "Trida"

UML schéma [7] znázorňuje přesunutí metody „VratitSeznamZakuVeTride“ z třídy „ZakForm“ do třídy „Trida“. K tomuto refaktorování byly dva důvody. Prvním důvodem bylo, že metoda „VratitSeznamZakuVeTride“, se svou povahou více hodí do třídy „Trida“ (chceme získat seznam žáků objektu třída). Druhým důvodem byl fakt, že i samotné tělo dané metody se odkazuje na další dvě metody (obr. 20), které patří do třídy „Trida“.

```
public IList VratitSeznamZakuVeTride(int pTestovaniID, bool pObnovaZaka)
{
    ArrayList lSeznamZaku = new ArrayList();
    if (pObnovaZaka) {
        lSeznamZaku = (ArrayList)this.VratitSmazaneZaky(pTestovaniID, this.TridaID);
    }
    else {
        lSeznamZaku = (ArrayList)this.VratitZakyATesty(pTestovaniID, this.TridaID);
    }
    return lSeznamZaku;
}
```

Obrázek 20: tělo metody "VratitSeznamZakuVeTride"

10.6. Nahradit pole objektem

- **Shrnutí**

  Pole by se mělo používat pouze k uchovávání podobných objektů, ve stanoveném pořadí. Pokud narazíme na skutečnost, že se v něm skladuje spousta různých věcí, je lepší nahradit toto pole objektem, který bude mít zvláštní položku pro každý prvek pole.

- **Postup**

  Nejprve vytvoříme novou třídu, která později nahradí pole. Tam kde se pole používá, jej nahradíme novou třídou a pro každý prvek pole vytvoříme ve třídě položku. Kód přeložíme a otestujeme.

- **Příklad**

```
//cast kodu

string[] lCisloPrijmeni = null;
lCisloPrijmeni[0] = lZak.VratitCisloZaka().ToString();
lCisloPrijmeni[1] = lZak.VratitPrijmeniZaka();

//cast kodu
```

Obrázek 21: část kódu, která používá nevhodně pole

V původním kódu obsahovaly jednotlivé položky v poli číslo a příjmení žáka. Jelikož bylo pole v tomto případě typu string, bylo navíc nutné překonvertovat návratovou hodnotu metody „VratitCisloZaka“ také na string.

```
//cast kodu

UdajeZaka lUdajeZaka = new UdajeZaka();
lUdajeZaka.CisloZaka = lZak.VratitCisloZaka();
lUdajeZaka.PrijmeniZaka = lZak.VratitPrijmeniZaka();

//cast kodu
```

Obrázek 22: pole nahrazeno objektem "UdajeZaka"

```
public class UdajeZaka {
    public UdajeZaka() {}

    private string lCisloZaka;
    public string CisloZaka {
        get { return lCisloZaka;}
        set { lCisloZaka = value;}
    }

    private string lPrijmeniZaka;
    public string PrijmeniZaka {
        get { return lPrijmeniZaka; }
        set { lPrijmeniZaka = value; }
    }
}
```

Obrázek 23: nově vytvořený objekt "UdajeZaka"

Tyto všechny nepříjemnosti byly naráz odstraněny nahrazením pole objektem „UdajeZaka“ (obr. 23). Tato datová třída obsahuje pouze konstruktor a dvě veřejné položky „CisloZaka“ a „PrijmeniZaka“. Na obr. 22 je vidět, jak vypadá kód, po nahrazení pole objektem „UdajeZaka“. Nejprve je vytvořena instance nové třídy a následně jsou jednotlivým položkám přiřazeny nové hodnoty. V tomto případě již není nutné konvertovat metodu „VratitCisloZaka“ na string, jelikož je datová položka „CisloZaka“ stejného typu jako daná metoda (int).

10.7. Nahradit vnořenou podmínku varovnými klauzulemi

- **Shrnutí**

  Toto refaktorování je dobré použít tehdy, když metoda obsahuje podmíněný příkaz, u kterého není patrné, co je normální průběh. Pokud podmínka testuje neobvyklou situaci, pak tuto podmínku otestujeme a v případě její platnosti použijeme příkaz `return`.

- **Postup**

  Místo každého testu vložíme varovnou klauzuli, která obsahuje buď `return`, nebo vyvolává podmínku.

- **Příklad**

```
private string VratHeslo(object sender, EventArgs e) {
    //cast kodu

    string lHeslo;
    if (sender == this.btnFastLoginAAA) {
        lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
    } else {
        if (sender == this.btnFastLoginZSPB) {
            lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
        } else {
            if (sender == this.btnFastLoginBDGKM) {
                lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
            } else {
                lHeslo = "????";
            }
        }
    }
    return lHeslo;
}
```

Obrázek 24: metoda, která má zbytečně složité větvení

```
private string VratHeslo(object sender, EventArgs e) {
    //cast kodu

    string lHeslo = "????";
    if (sender == this.btnFastLoginAAA)
        return lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
    if (sender == this.btnFastLoginZSPB)
        return lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
    if (sender == this.btnFastLoginBDGKM)
        return lHeslo = Ruzne.DB.SelectStr("select top 1 heslo from .... ");
    return lHeslo;

    //cast kodu
}
```

Obrázek 25: vzhled metody po refaktorování

10.8. Odstranit přístupovou metodu pro zápis

- **Shrnutí**

  Toto refaktorování použijeme, pokud má být hodnota položky nastavena při vytváření objektu a dále by se již neměla měnit. Pokud tedy nechceme, aby se po vytvoření objektu položky již neměnily, pak nebudeme vytvářet metody pro zápis.

- **Postup**

  Nejprve ověříme, zda-li je metoda pro zápis volána pouze v konstruktoru, nebo v metodách, které volá konstruktor. Poté změníme konstruktor tím způsobem, že bude přistupovat k položce přímo. Na závěr odstraníme metodu pro zápis, kód přeložíme a otestujeme.

- **Příklad**

```
public class Otazka {
    public Otazka(int pTestOtazkaID,
                  int pCisloOtazky,
                  int pPocetMoznosti) {
        NastavOtazku(pTestOtazkaID,
                     pCisloOtazky,
                     pPocetMoznosti);
    }

    private int testOtazkaID;
    public int TestOtazkaID {
        get { return testOtazkaID; }
    }

    private int pocetMoznosti;
    public int PocetMoznosti {
        get { return pocetMoznosti; }
    }

    private int cisloOtazky;
    public int CisloOtazky {
        get { return cisloOtazky; }
    }

    private void NastavOtazku(
                int pTestOtazkaID,
                int pCisloOtazky,
                int pPocetMoznosti) {
        testOtazkaID = pTestOtazkaID;
        cisloOtazky = pCisloOtazky;
        pocetMoznosti = pPocetMoznosti;
    }
}
```

Obrázek 26: datová třída, která obsahuje metodu pro zápis

```
public class Otazka {
    public Otazka(int pTestOtazkaID,
                  int pCisloOtazky,
                  int pPocetMoznosti) {
        testOtazkaID = pTestOtazkaID;
        cisloOtazky = pCisloOtazky;
        pocetMoznosti = pPocetMoznosti;
    }

    private int testOtazkaID;
    public int TestOtazkaID {
        get { return testOtazkaID; }
    }

    private int pocetMoznosti;
    public int PocetMoznosti {
        get { return pocetMoznosti; }
    }

    private int cisloOtazky;
    public int CisloOtazky {
        get { return cisloOtazky; }
    }
}
```

Obrázek 27: datová třída, kde konstruktor přistupuje k položkám přímo

10.9. Vyjmout rodičovskou třídu

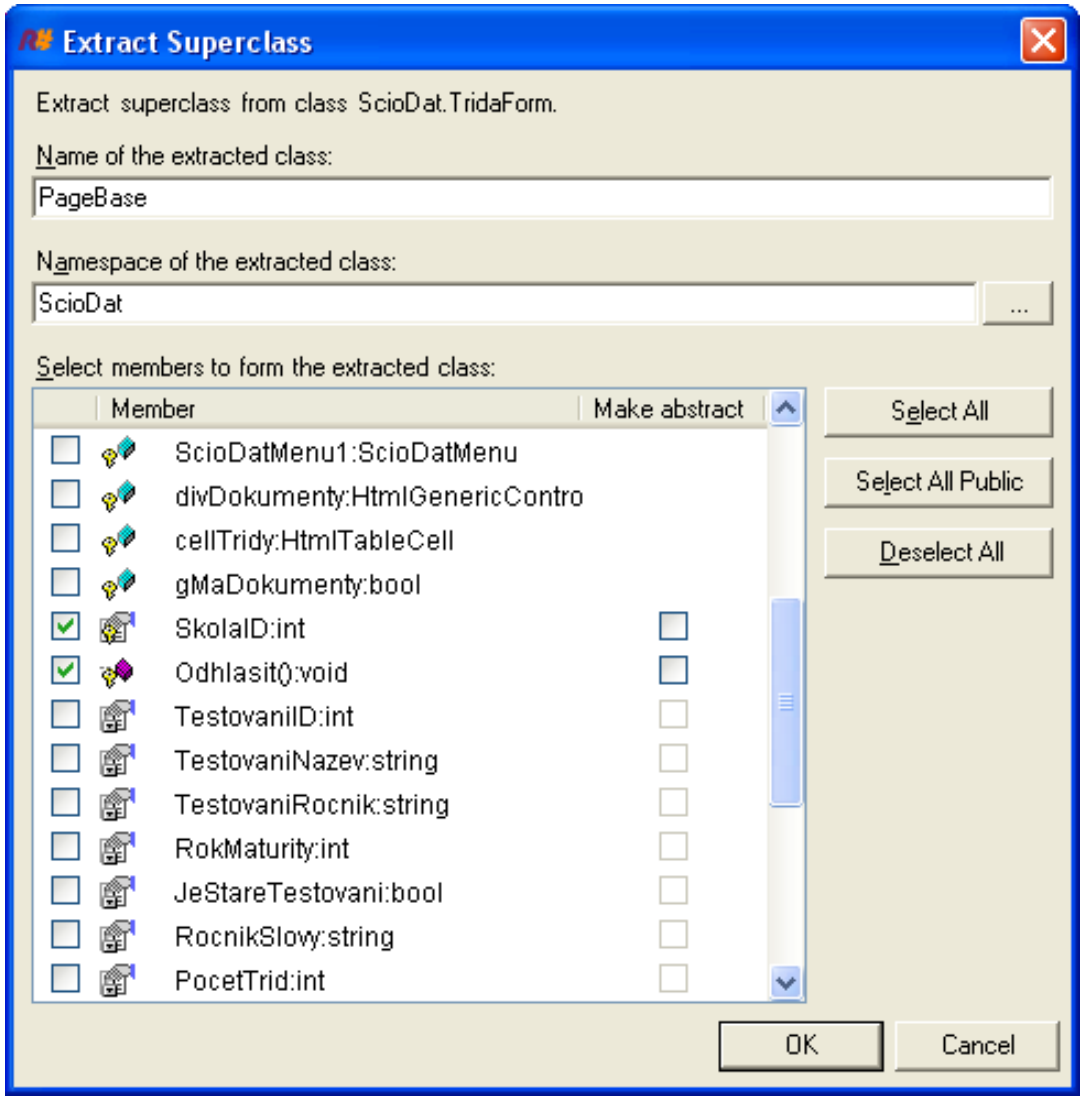
- **Shrnutí**

  Najdeme-li na více místech stejnou strukturu programu, je jisté, že program zlepšíme, pokud se nám jej podaří sjednotit. Jedním ze způsobů, jak duplikovat kód, je mít dvě třídy, které buď dělají skoro totéž stejným způsobem, nebo dělají skoro totéž odlišnými způsoby. Díky objektům, můžeme tento problém vyřešit pomocí dědičnosti.

- **Postup**

  Nejprve vytvoříme rodičovskou třídu a všechny původní třídy, které obsahují duplicitní kód, od této rodičovské třídy odvodíme. Do této rodičovské třídy pak postupně přesuneme všechny společné prvky – pomocí refaktorování přesunout položku výš, přesunout metodu výš, přesunout tělo konstruktoru výš.

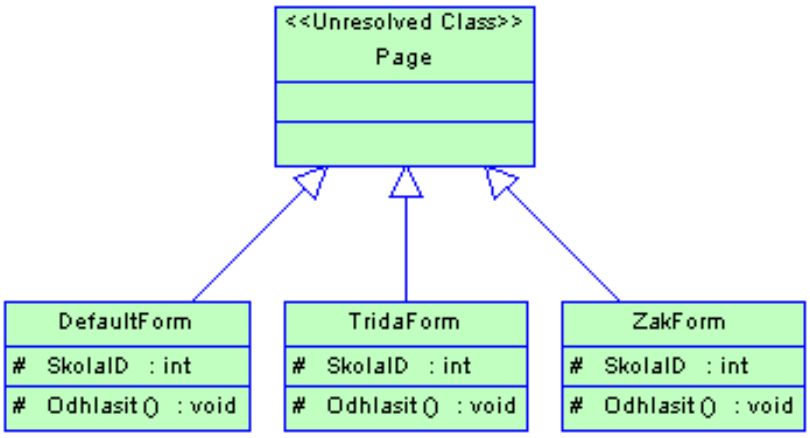
V modulu Resharper je postup ještě jednodušší. Z hlavní nabídky vybereme ReSharper-> Refactor-> „Extract Superclass...“ a otevře se nám následující dialogové okno.



Obrázek 28: dialogové okno pro vyjmutí rodičovské třídy

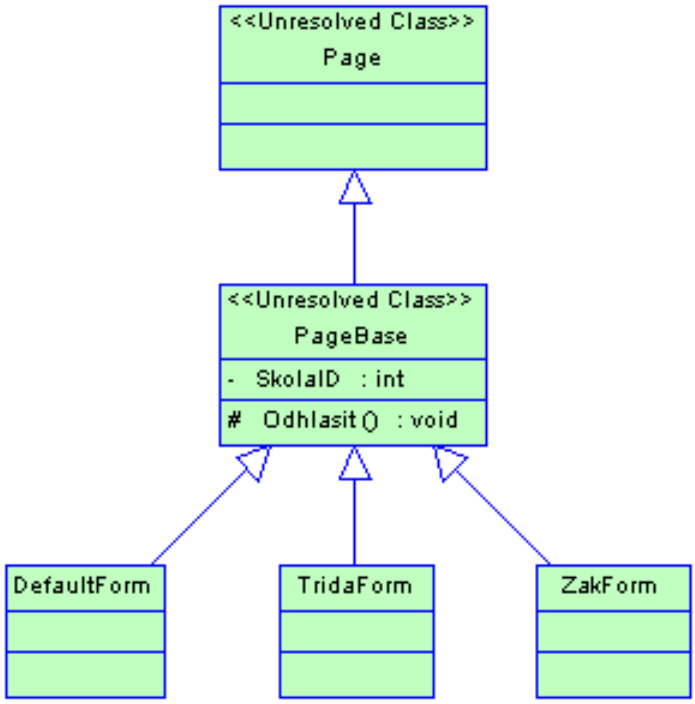
Zde zadáme název rodičovské třídy, dále jmenný prostor, do kterého má rodičovská třída patřit a na závěr vybereme všechny položky základní třídy, které mají být přesunuty výš. Celou akci potvrdíme. ReSharper za nás přesune všechny zvolené položky do nové rodičovské třídy a zároveň původní třídu od této nové třídy podědí.

- **Příklad**



Obrázek 29: duplicitní kód ve třídách

Na obrázku výše je jasně vidět, jak formulářové třídy „Default“, „Trida“ a „Zak“ obsahují duplicitní kód. Přesně se jedná o vlastnost „SkolaID“ a metodu „Odhlasit“.



Obrázek 30: duplicitní kód přesunut do rodičovské třídy

Po refaktorování byla vytvořena rodičovská třída „PageBase“, která dědí od základní třídy „Page“ a do které byla přesunuta vlastnost „SkolaID“ a metoda „Odhlasit“. Od této rodičovské třídy pak dědí ostatní formulářové třídy.

11. Závěr

Cílem této bakalářské práce bylo shrnout získané poznatky o technice refaktorování a pomocí názorných ukázek, na mnou spravované aplikaci ScioDat, některé metody refaktorování názorně předvést. V práci jsem se snažil o lehkou pochopitelnost textů i bez hlubších znalostí dané problematiky. V některých částech je však třeba aspoň základních vědomostí o objektově orientovaném programování.

V prvních kapitolách jsem se snažil zaměřit hlavně na vysvětlení základních principů refaktorování a popsání některých problémů, které se dají refaktorováním vyřešit. V další kapitole jsem se soustředil na refaktorování pomocí softwarových nástrojů, kde jsem shrnul základní požadavky na tyto nástroje a jeden z nástrojů jsem podrobněji popsal. Po této části následuje kapitola o aplikaci ScioDat, která posloužila jako zdroj k ukázkám některých metod refaktorování. Závěrečná kapitola již patří názorným ukázkám nejčastěji používaných refaktorování a jednotlivé ukázky jsou demonstrovány právě na aplikaci ScioDat.

I když úprava kódu jako taková, existuje již mnoho let, tak pojem refaktorování je poměrně mladý. V současné době vznikají stalé nové metody refaktorování a hlavní pozornost se věnuje vývoji nových a dokonalejších nástrojů pro automatický refaktoring.

12. Literatura

[1] BECK, Kent. *EXtreme Programming eXplained: Embrance Change*, Boston: Addison-Wesley, 2005.

[2] FOWLER, Martin. *Refactoring – zlepšení existujícího kódu*, Praha: Grada, 2003. s. 394

[3] OPDYKE, William F. *Refactoring Object-Oriented Frameworks: disertační práce*, University of Illinois at Urbana-Champaign, 1992

[4] KERIEVSKY, Joshua. *Refactoring to Patterns*, Boston: Adison-Wesley, 2005. s. 384

[5] BROWN, William J. *AntiPatterns: refactoring software, architectures, and projects in crisis*, New York: Wiley, 1998. s. 309

[6] WAKE, William C. *Refactoring workbook*, Boston: Addison-Wesley Professional, 2004.

[7] FOWLER, Martin. UML Distilled, Addison-Weslez Professional, 3rd Edition, 2003

[8] JetBrains [web site]. Dostupné z: http://www.jetbrains.com [05. 05. 2006]. Domovská stránka firmy JetBRAINS, která se specializuje na vývoj softwaru, který usnadňuje uživatelům vývoj aplikací.

[9] Visual C# Express 2005: Refactoring v praxi [web site]. Dostupné z: http://www.pcsvet.cz/misc/search.php?kde=1&co=Refactoring+v+praxi [28. 04. 2006]. Seriál popisující refaktoring v praxi.

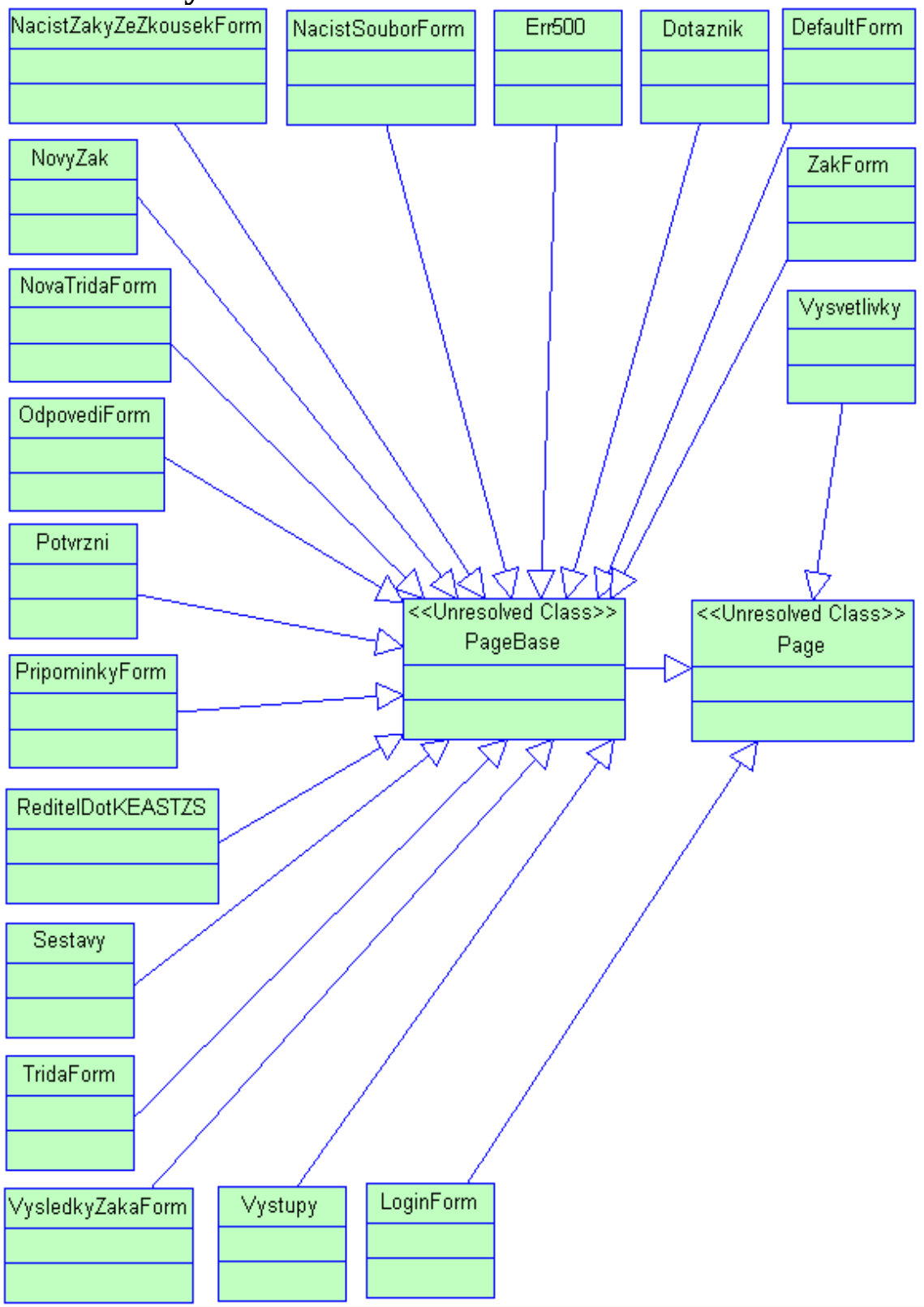
[10] Refactoring – Encyklopedie Wikipedie [web site]. Dostupné z http://en.wikipedia.org/wiki/Refactoring [28. 04. 2006].
Wikipedie je svobodná encyklopedie založená na Wiki. Svobodná znamená, že její obsah je vytvářen komunitou a přispívat může naprosto kdokoliv.

[11] Refactoring [web site]. Dostupné z http://www.refactoring.com/ [04. 05. 2006]. Tato stránka je dílem Martin Fowlera a veškerý její obsah je věnován problematice refaktorování.

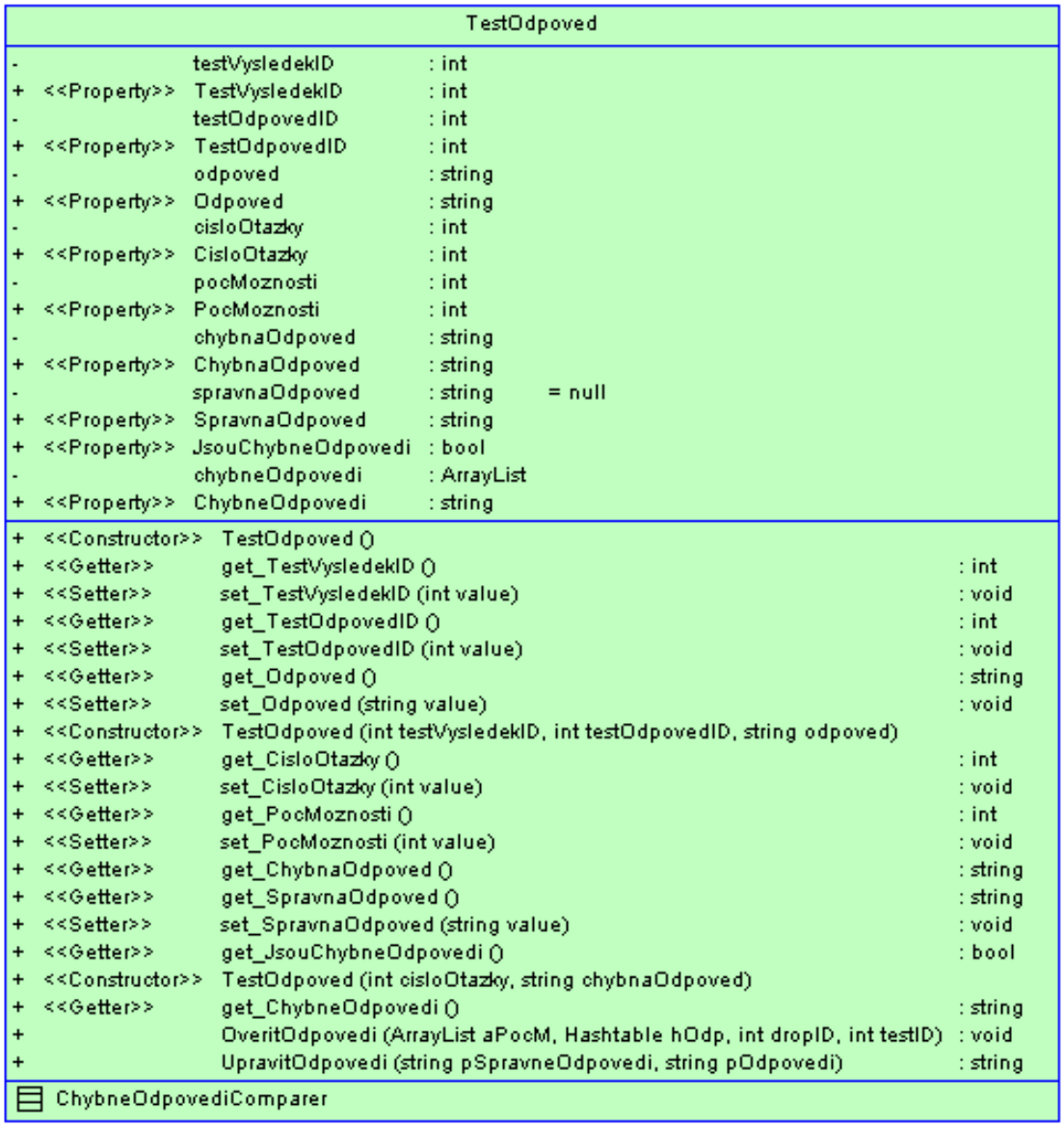
[12] Database refactoring [web site]. Dostupné z http://databaserefactoring.com/ [17. 05. 2006]. Tato stránka obsahuje katalog veškerých známých metod sloužících k refektorování.

13. Přílohy

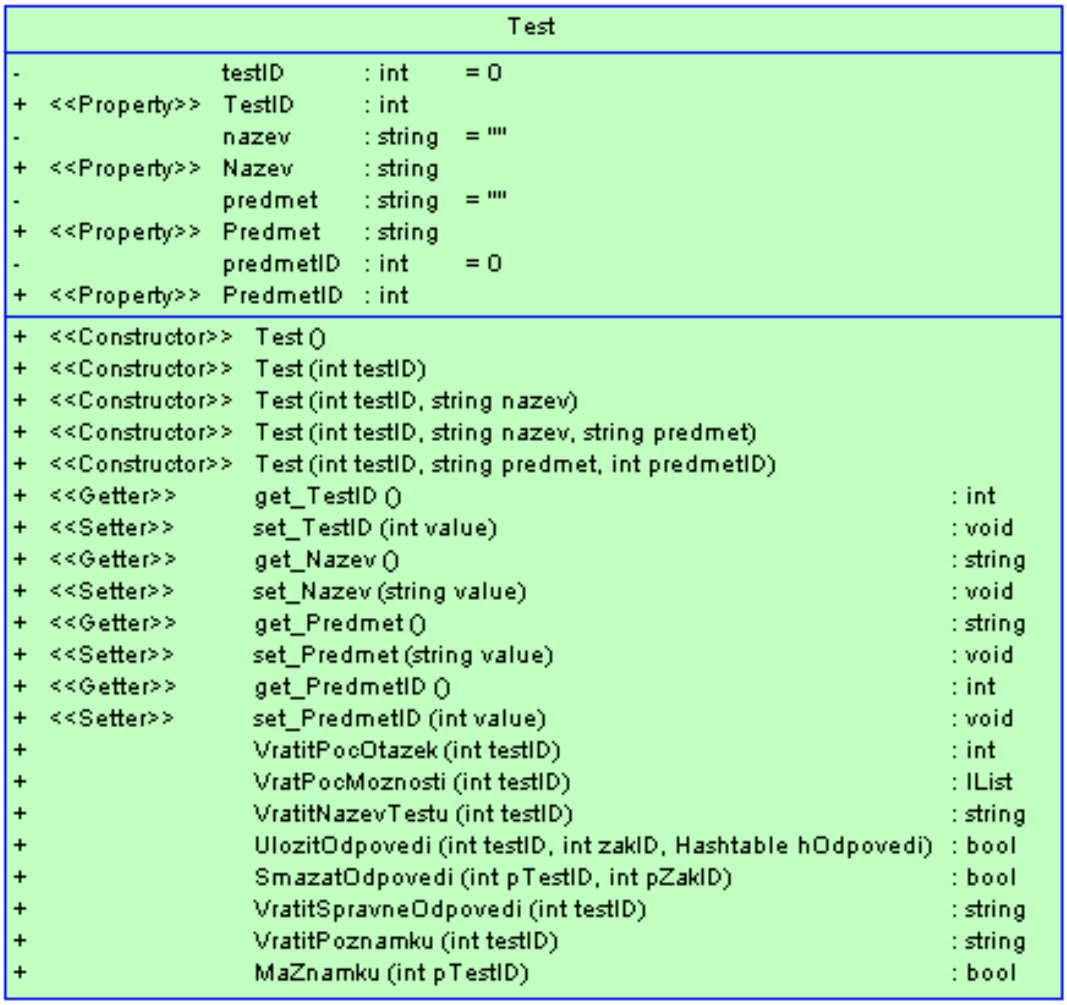
13. Přílohy

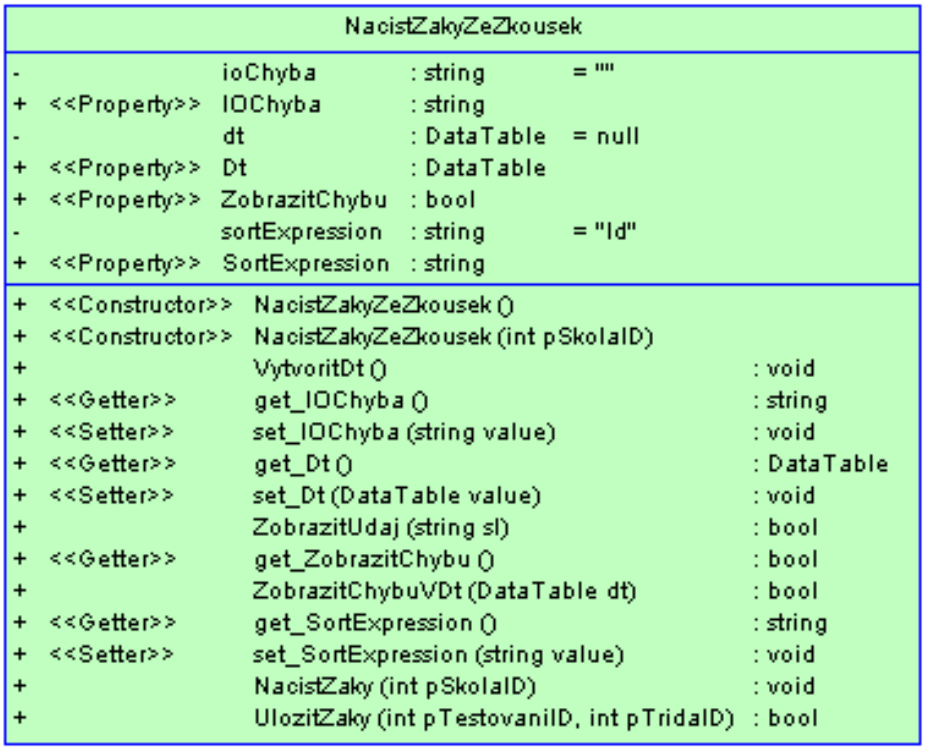


obrázek 31: UML schéma tříd webových formulářů

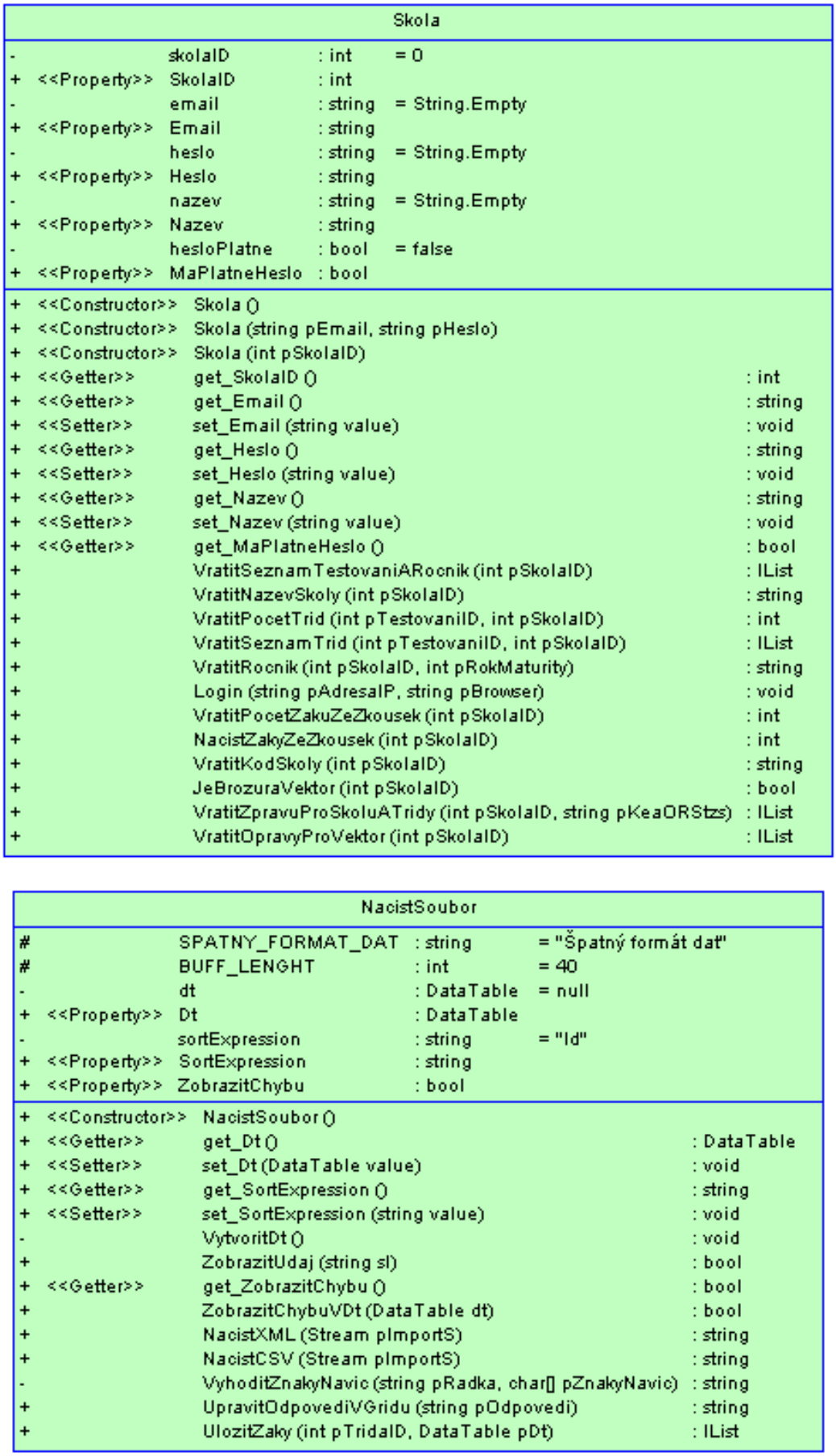


obrázek 32: UML schéma datových tříd

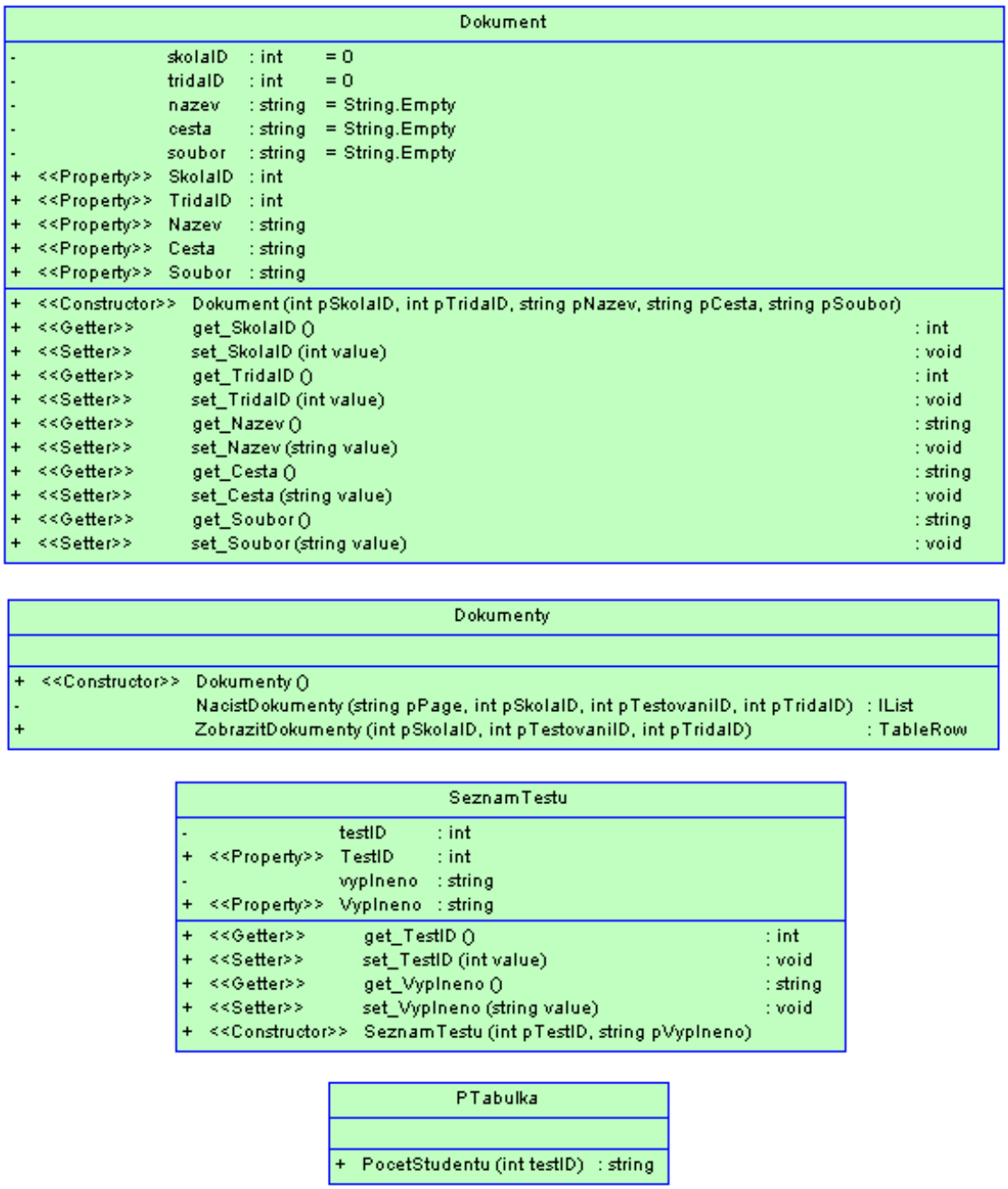




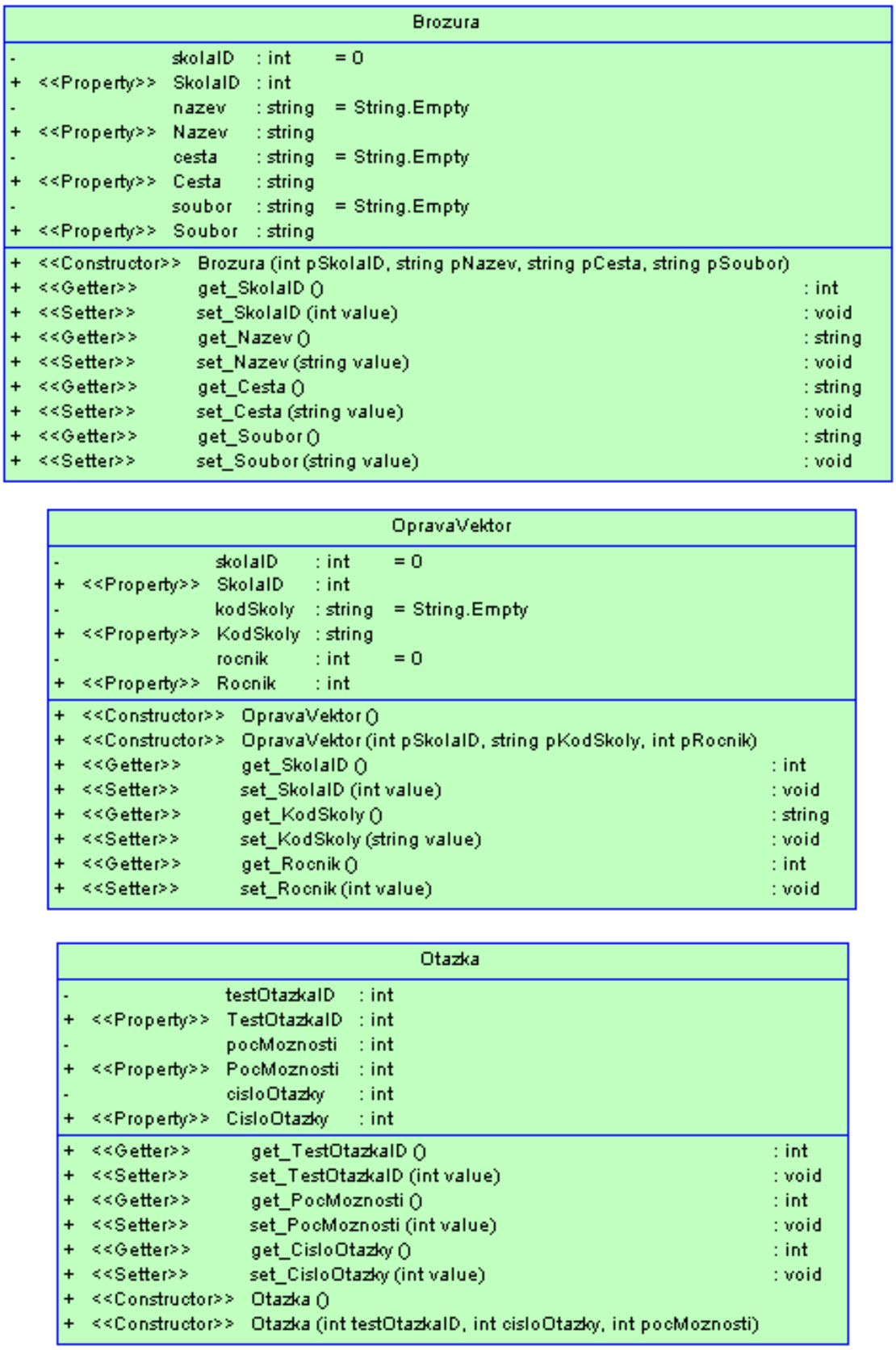
obrázek 33: UML schéma datových tříd



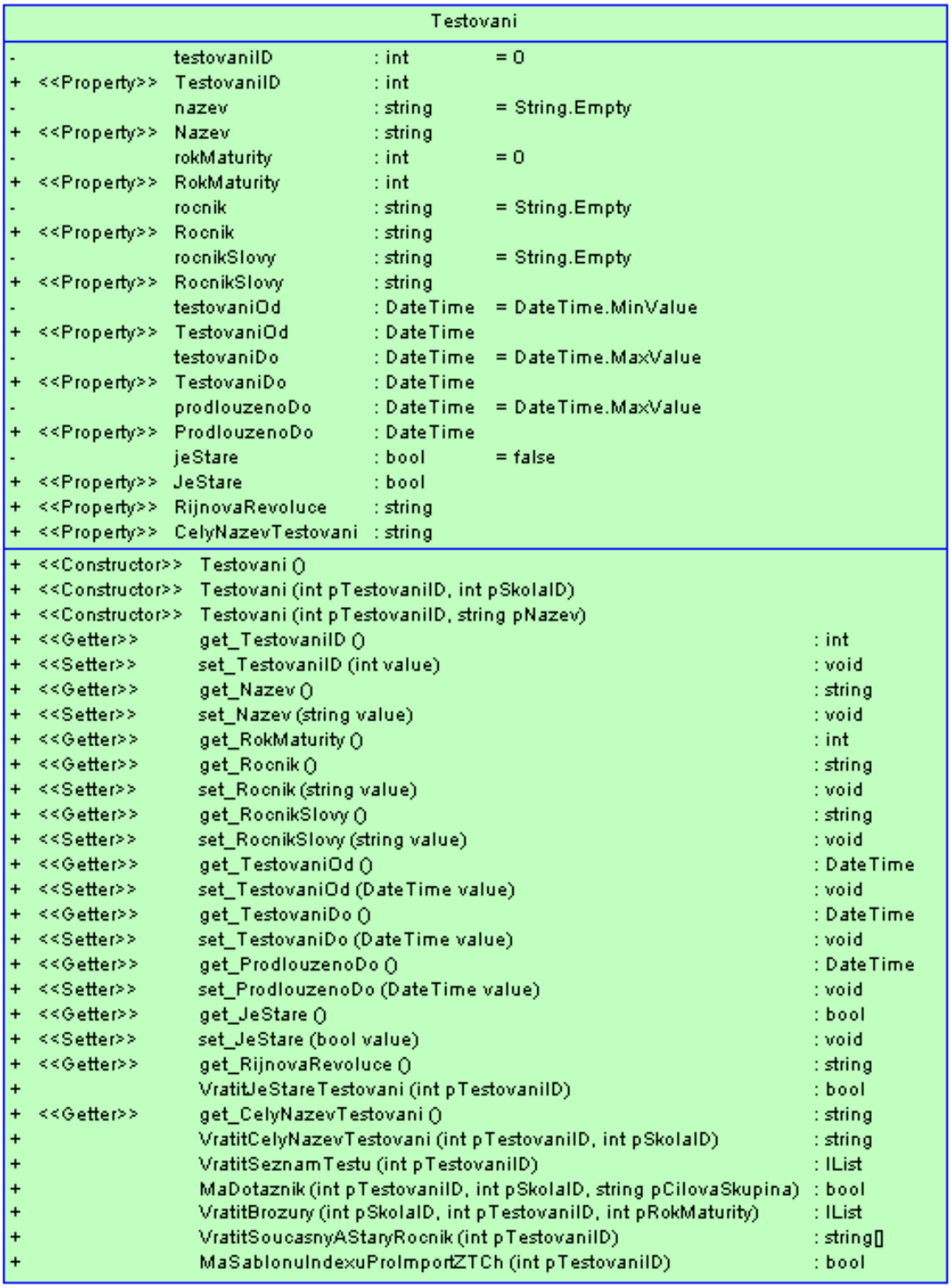
obrázek 34:UML schéma datových tříd



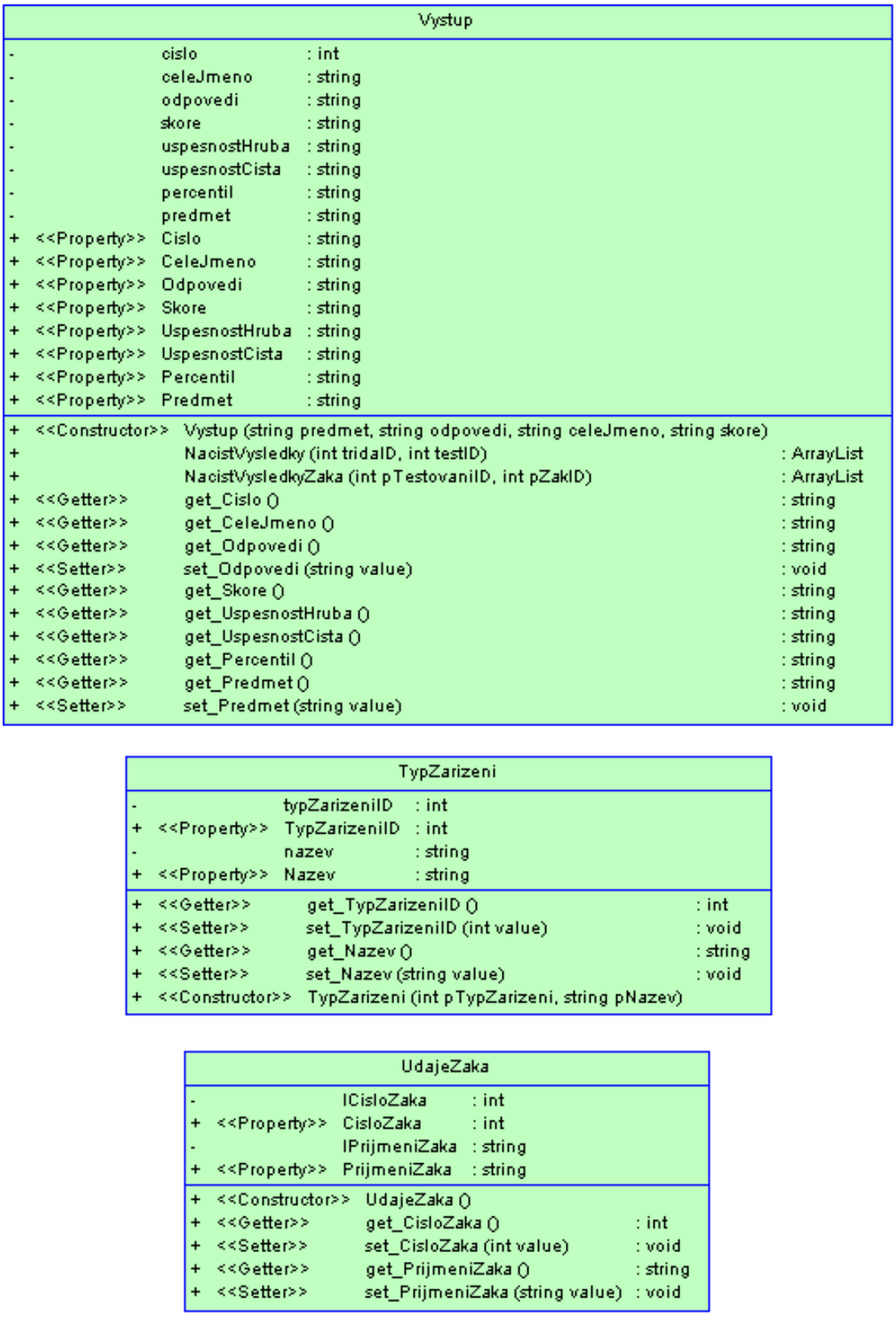
obrázek 35: UML schéma datových tříd



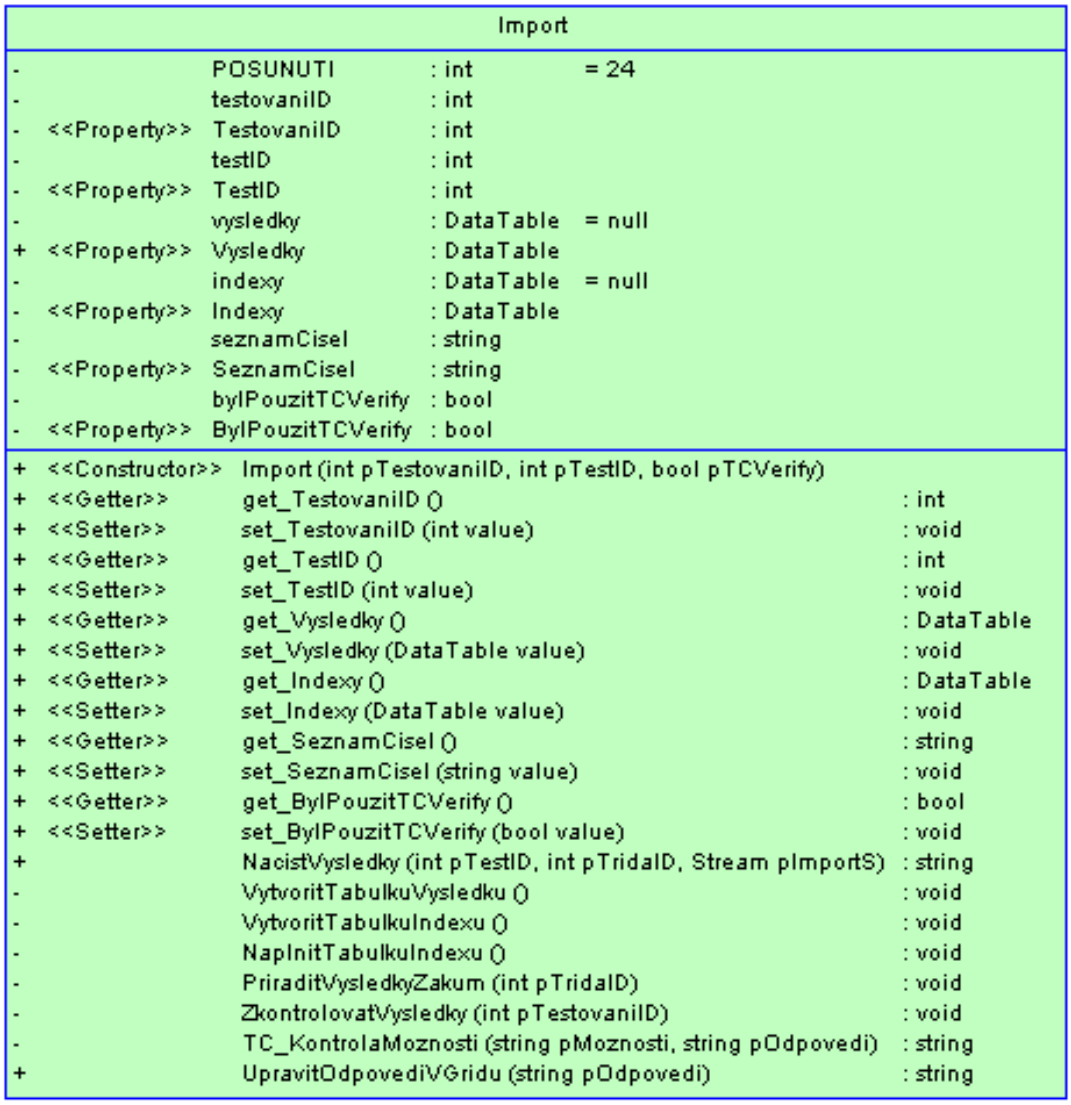
obrázek 36: UML schéma datových tříd



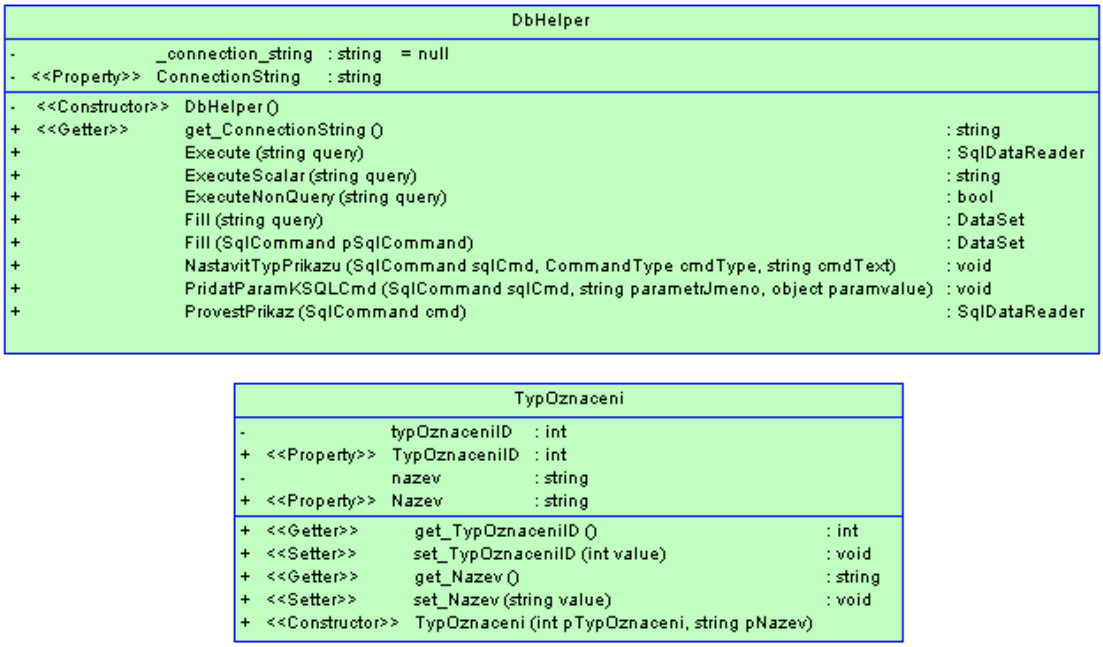
obrázek 37: UML schéma datové třídy



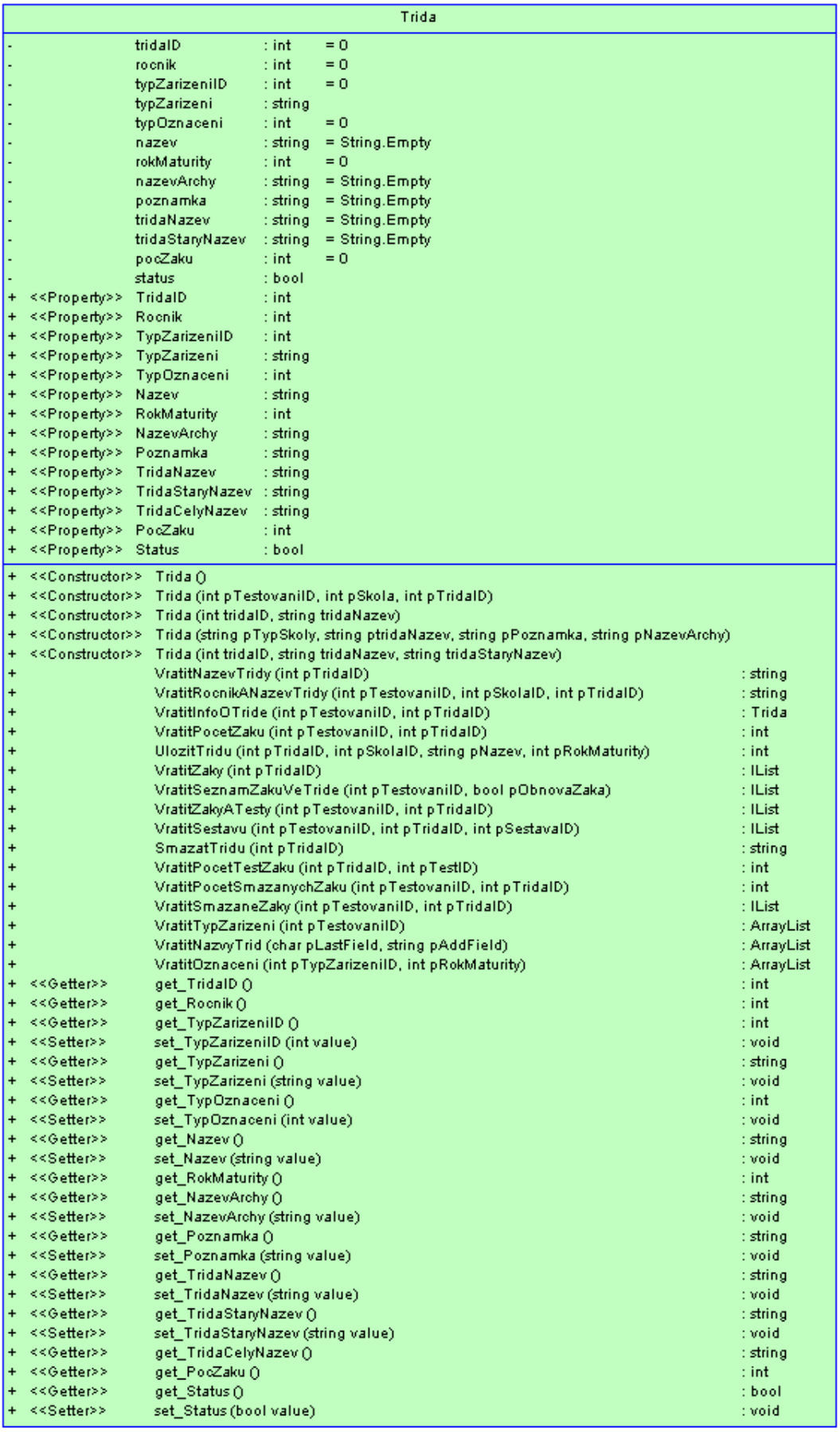
obrázek 38: UML schéma datových tříd



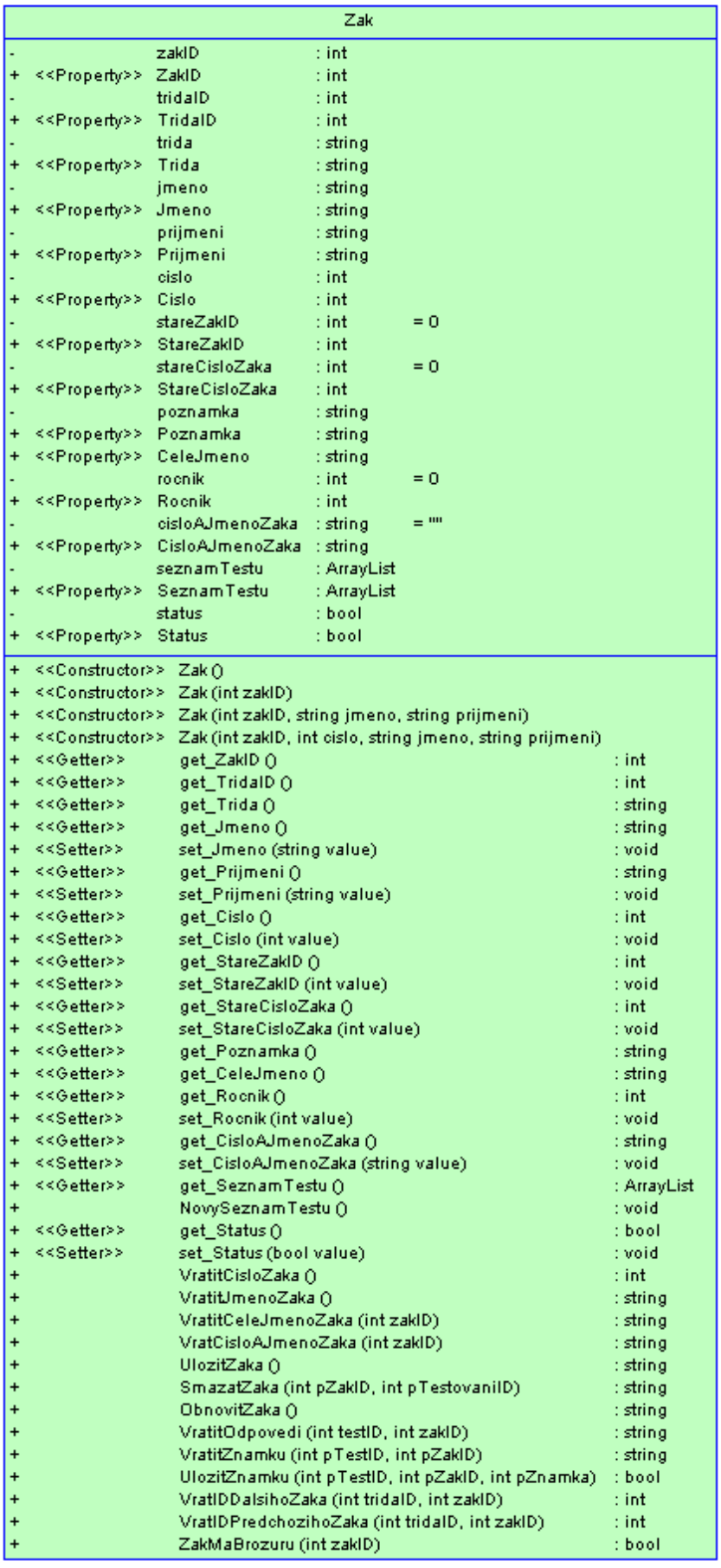
obrázek 39: UML schéma datových tříd



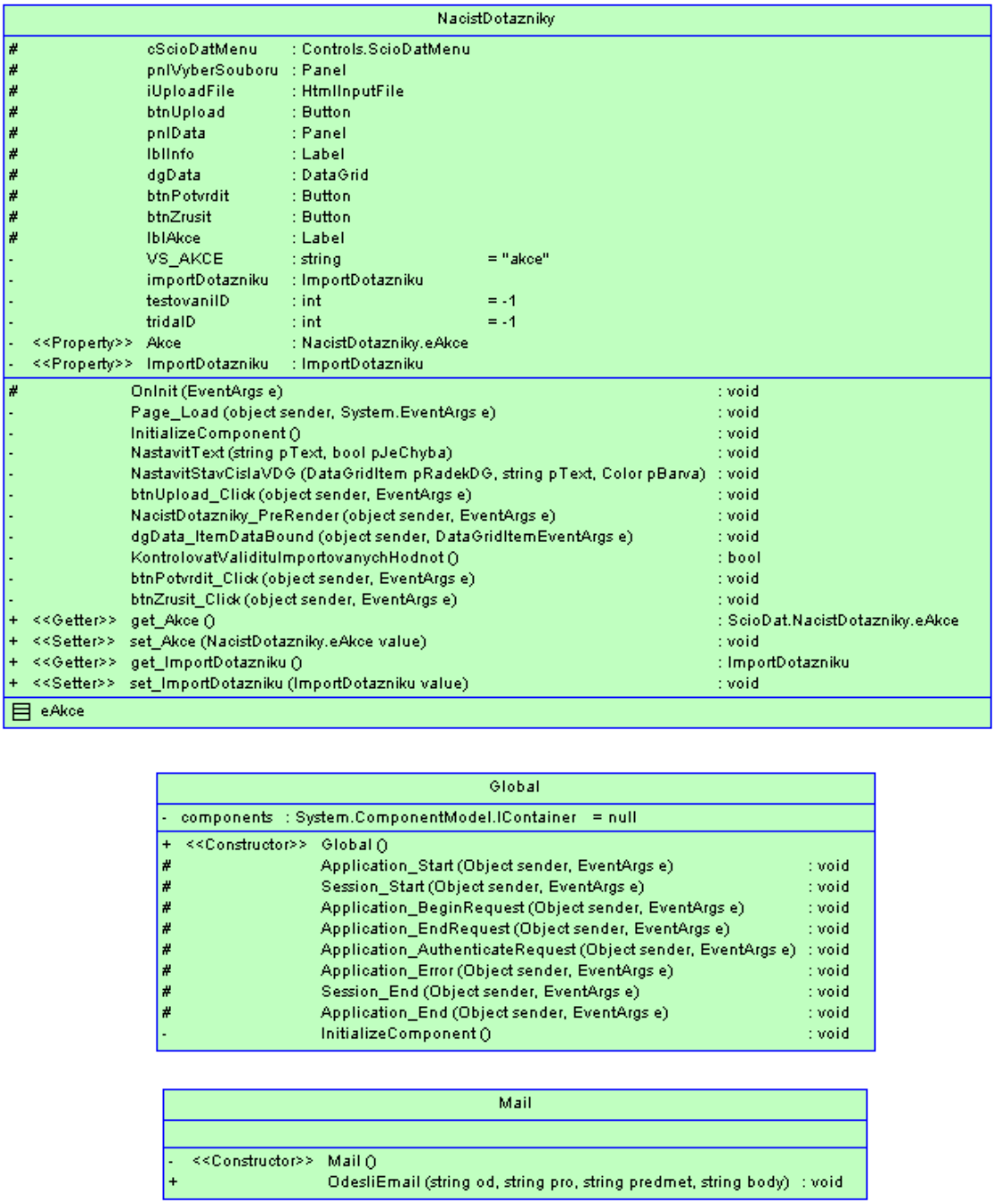
obrázek 40: UML schéma datových tříd



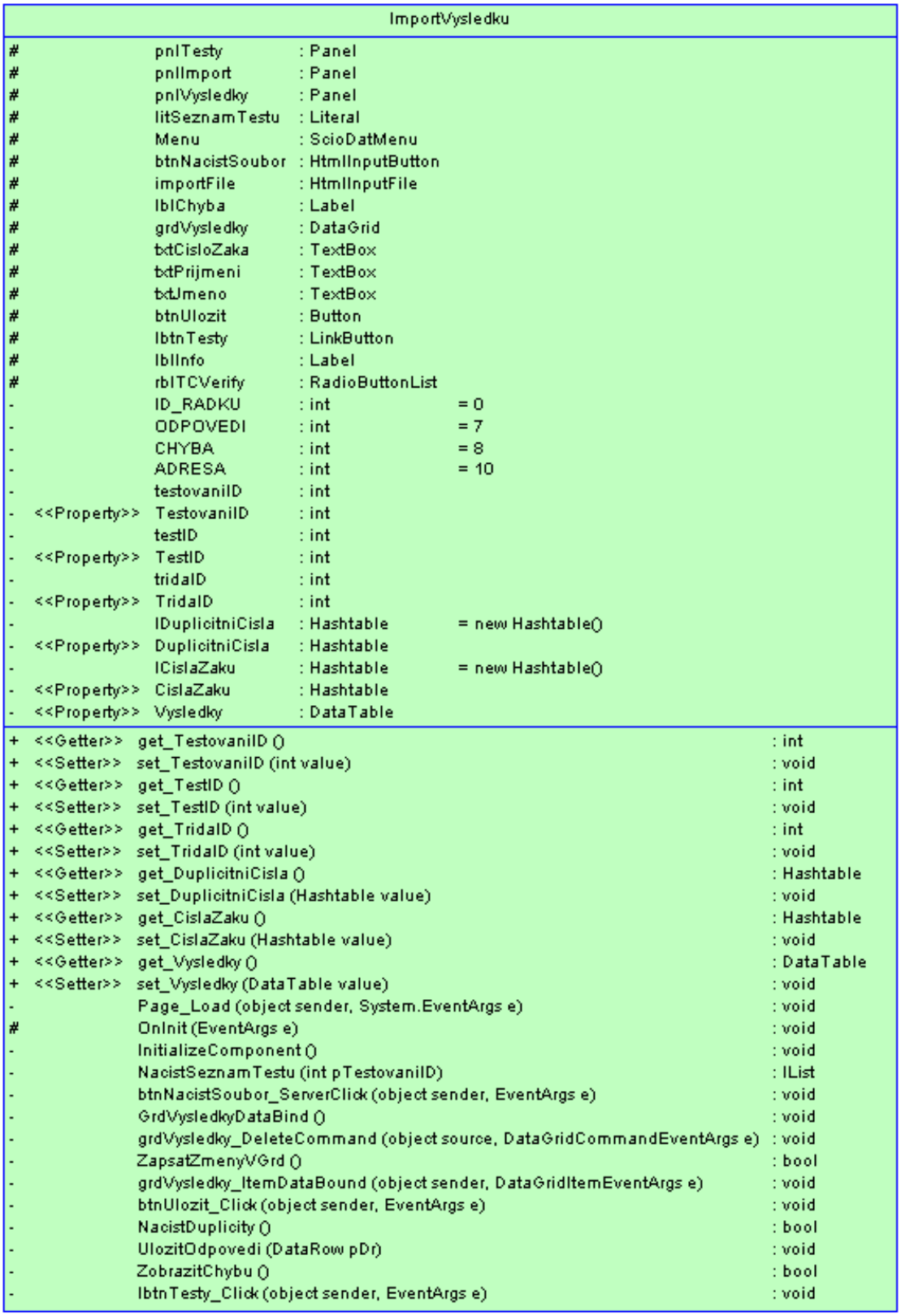
obrázek 41: UML schéma datové třídy



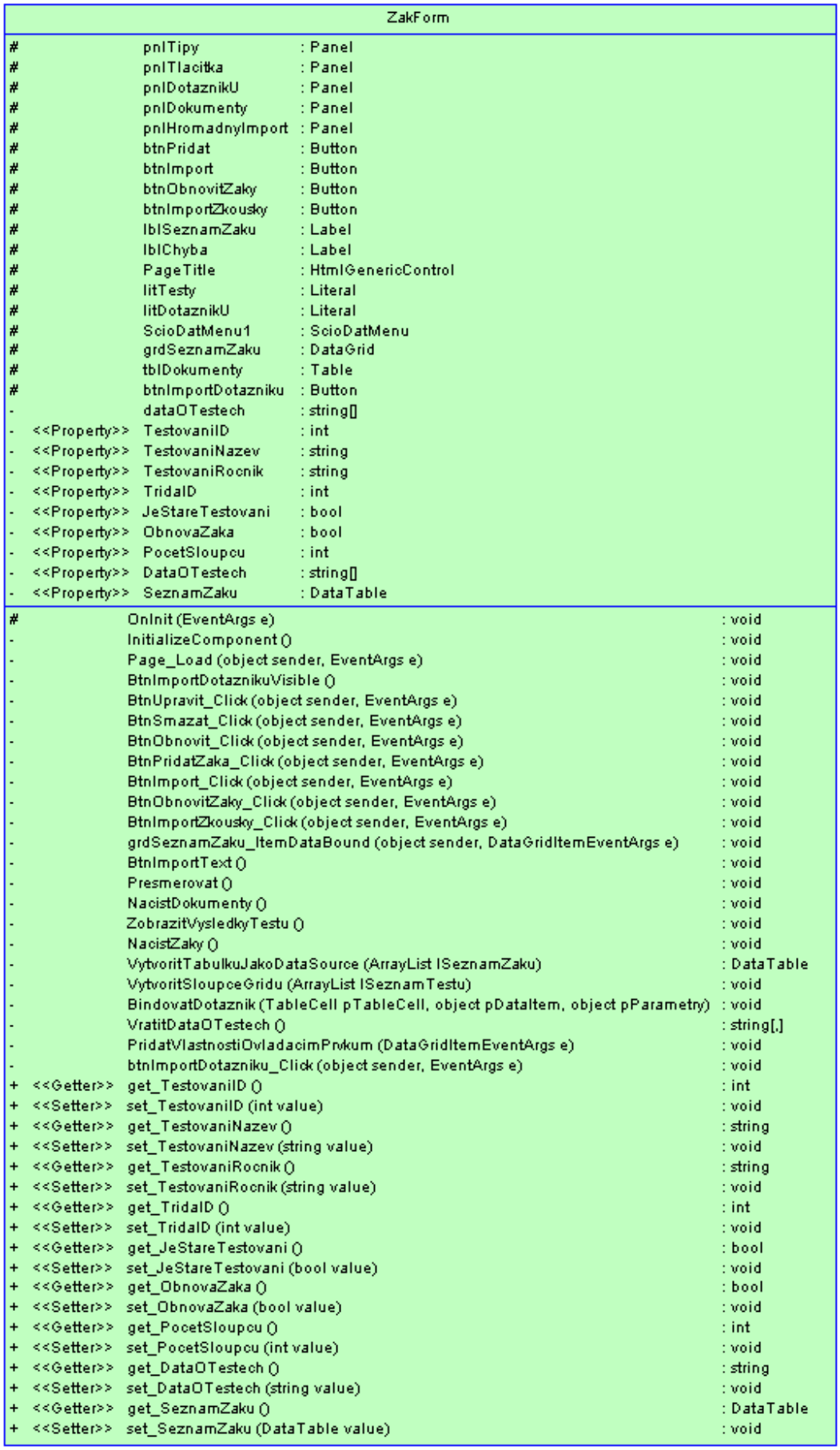
obrázek 42: UML schéma datové třídy



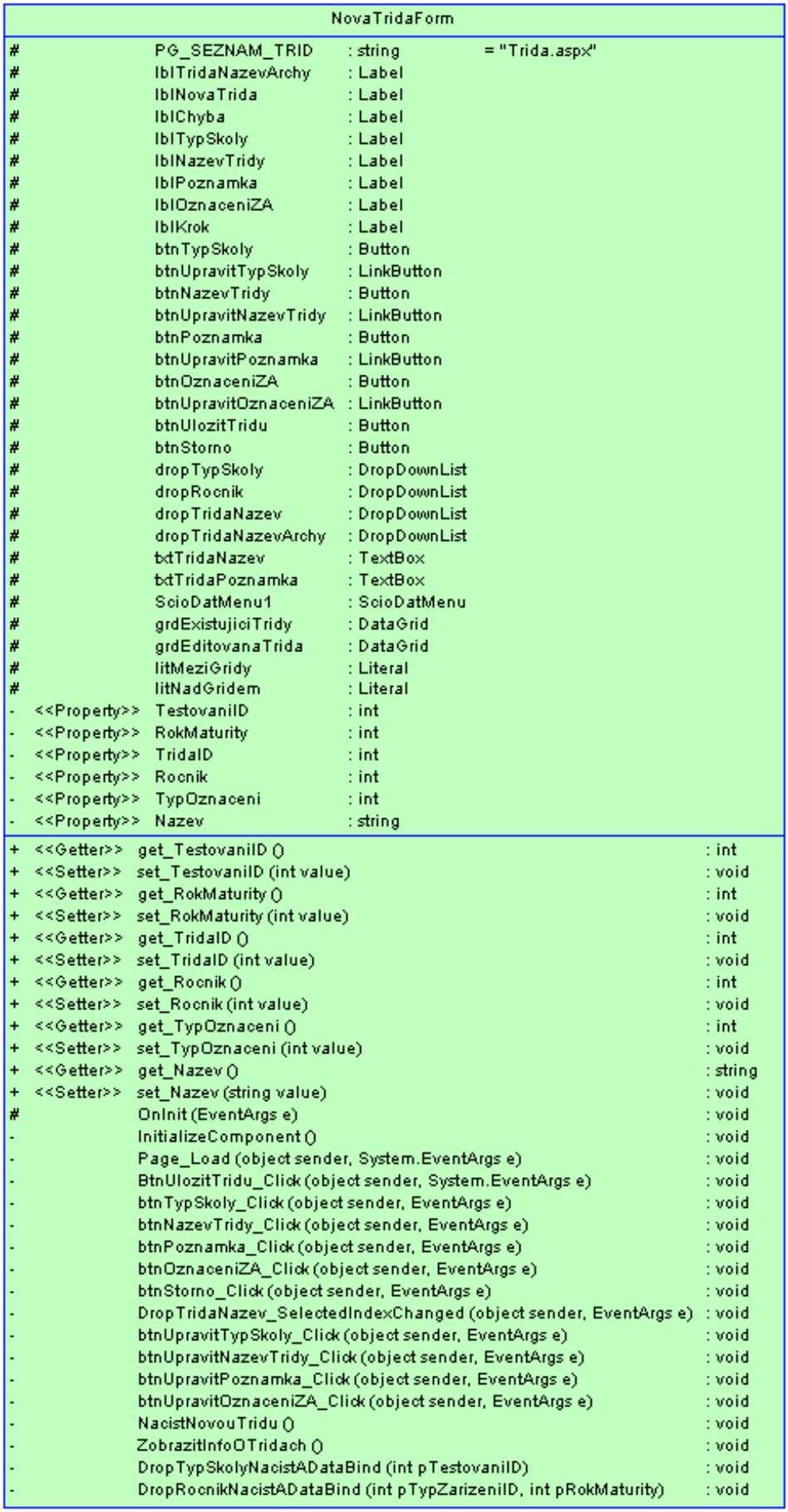
obrázek 43: UML schéma webových formulářů



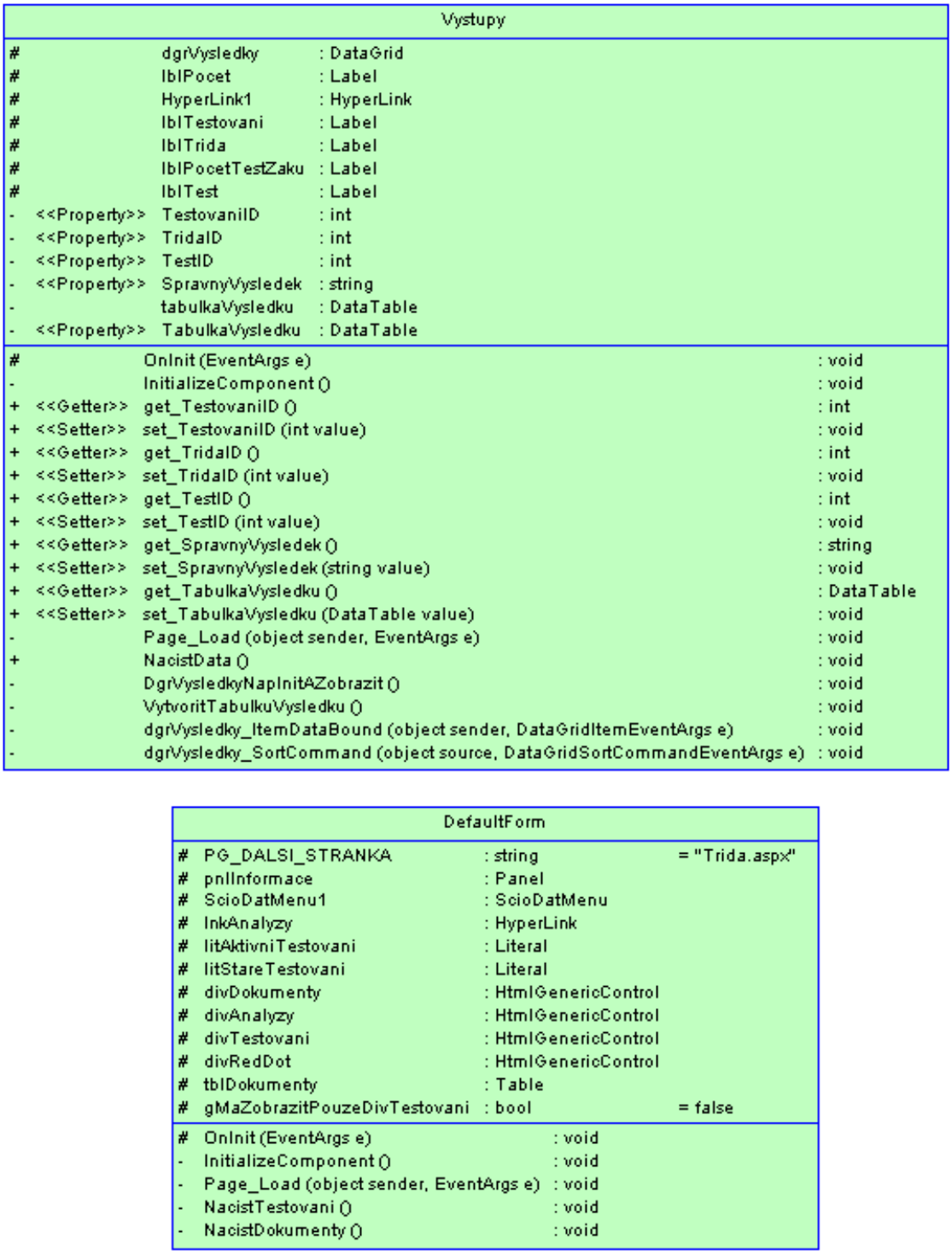
obrázek 44: UML schéma webového formuláře



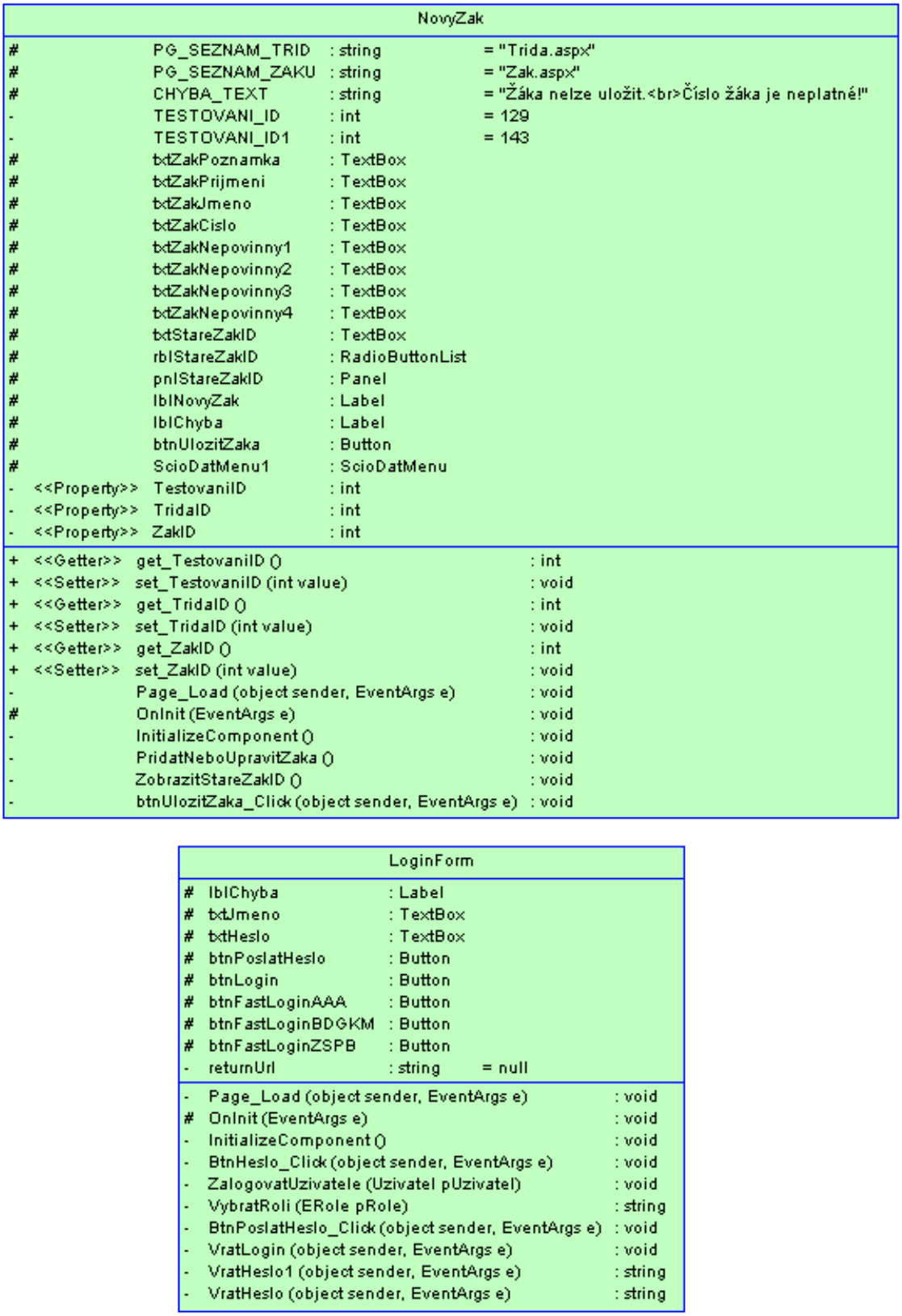
obrázek 45: UML schéma webového formuláře



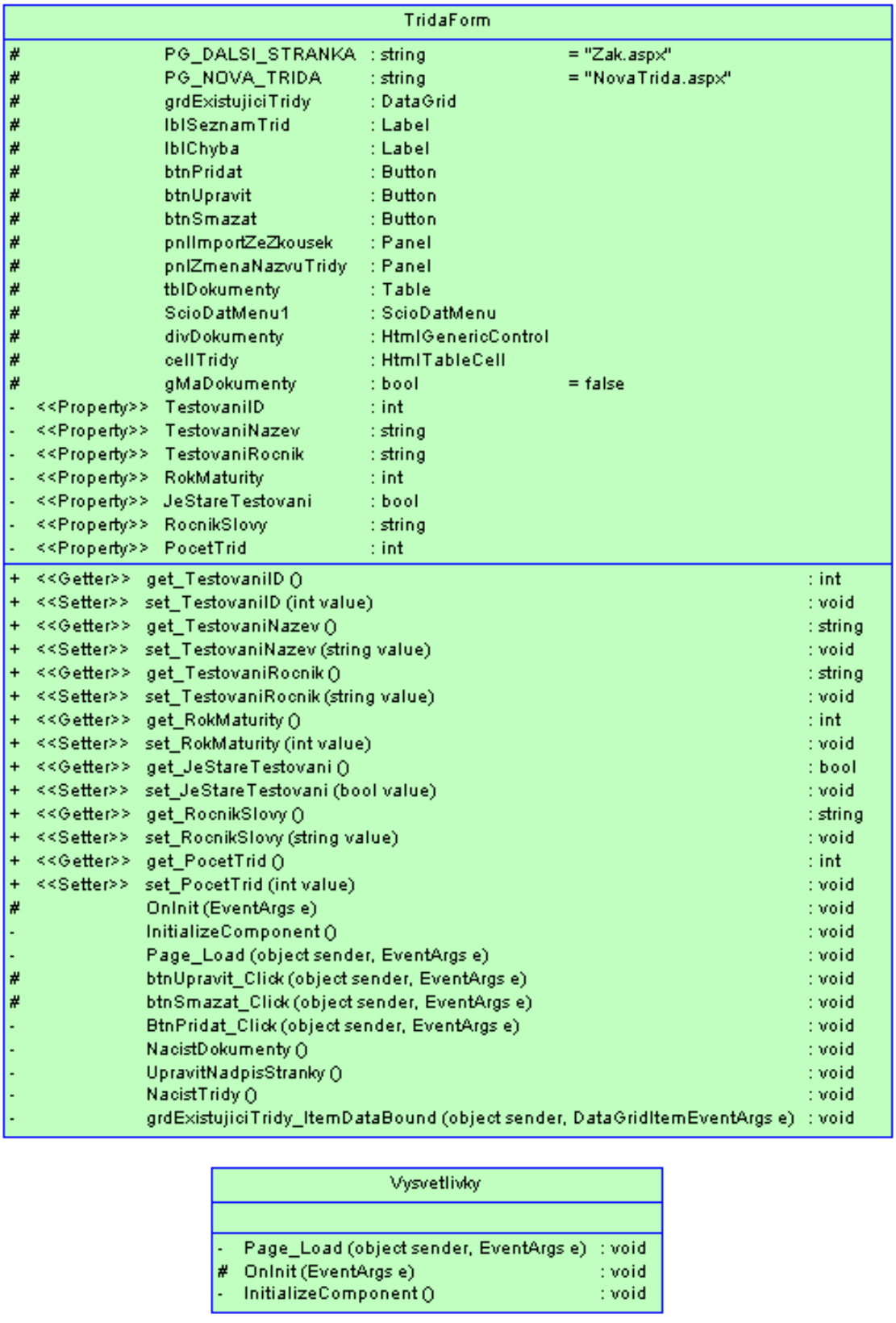
obrázek 46: UML schéma webového formuláře



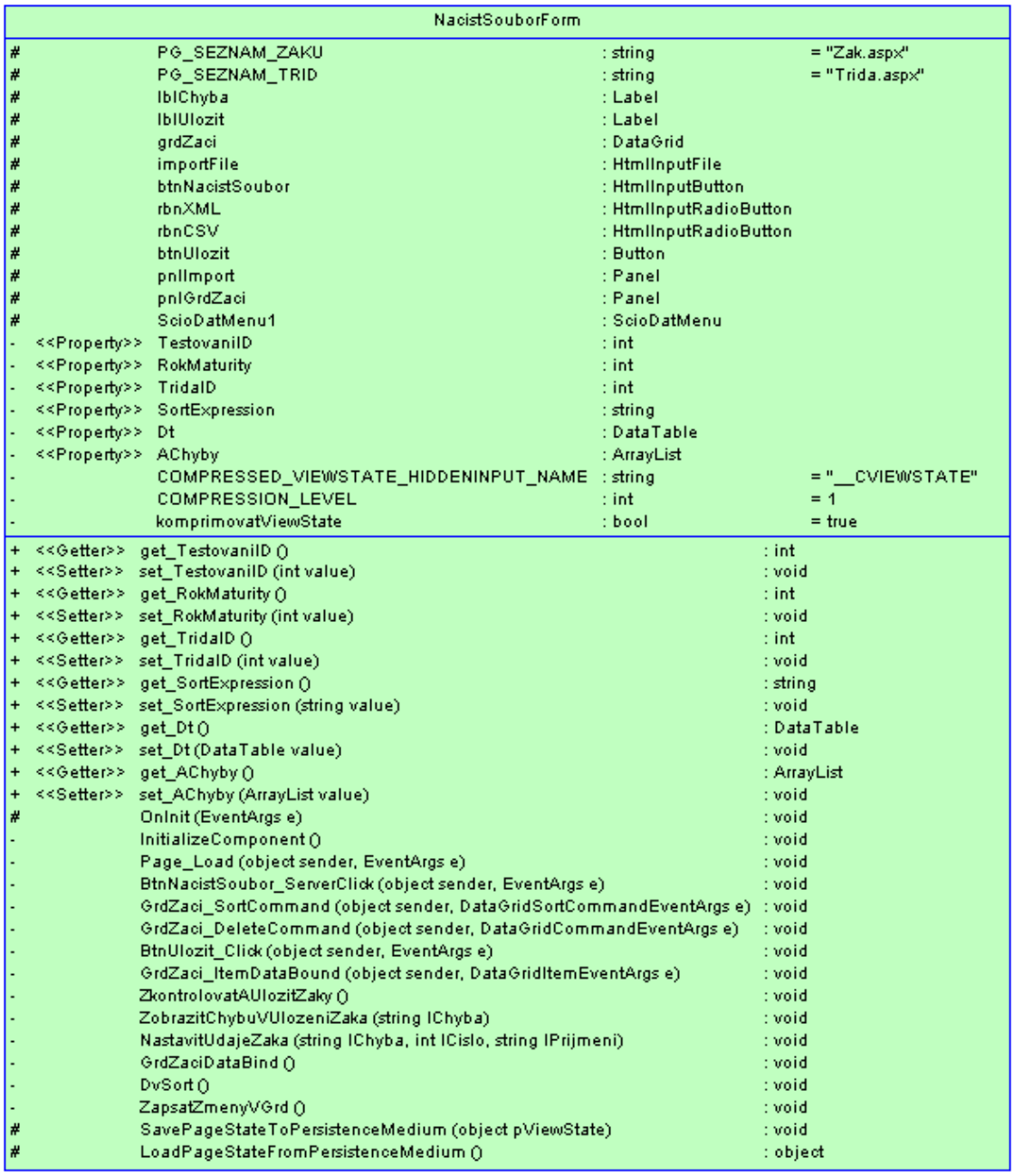
obrázek 47: UML schéma webových formulářů



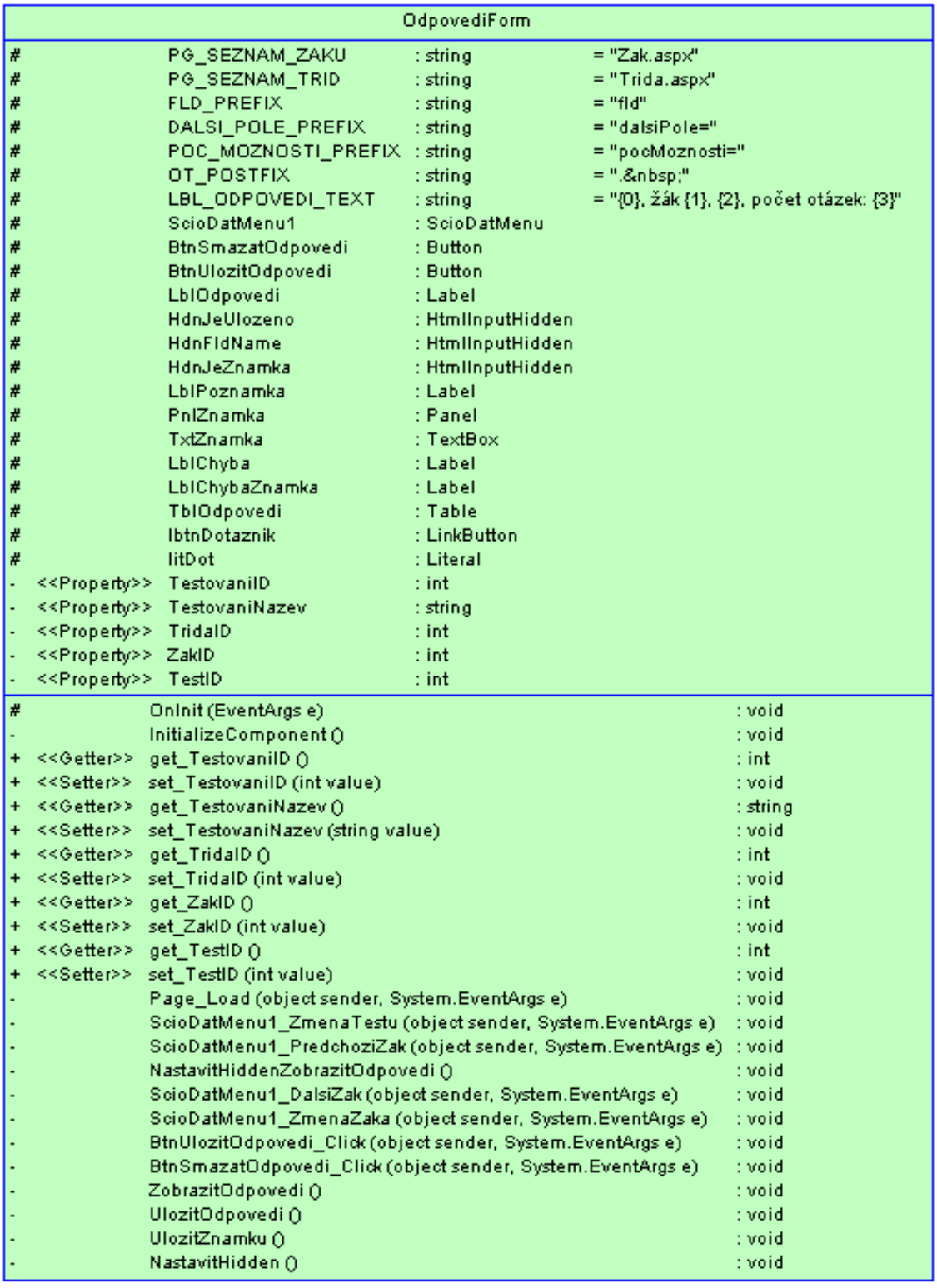
obrázek 48: UML schéma webových formulářů



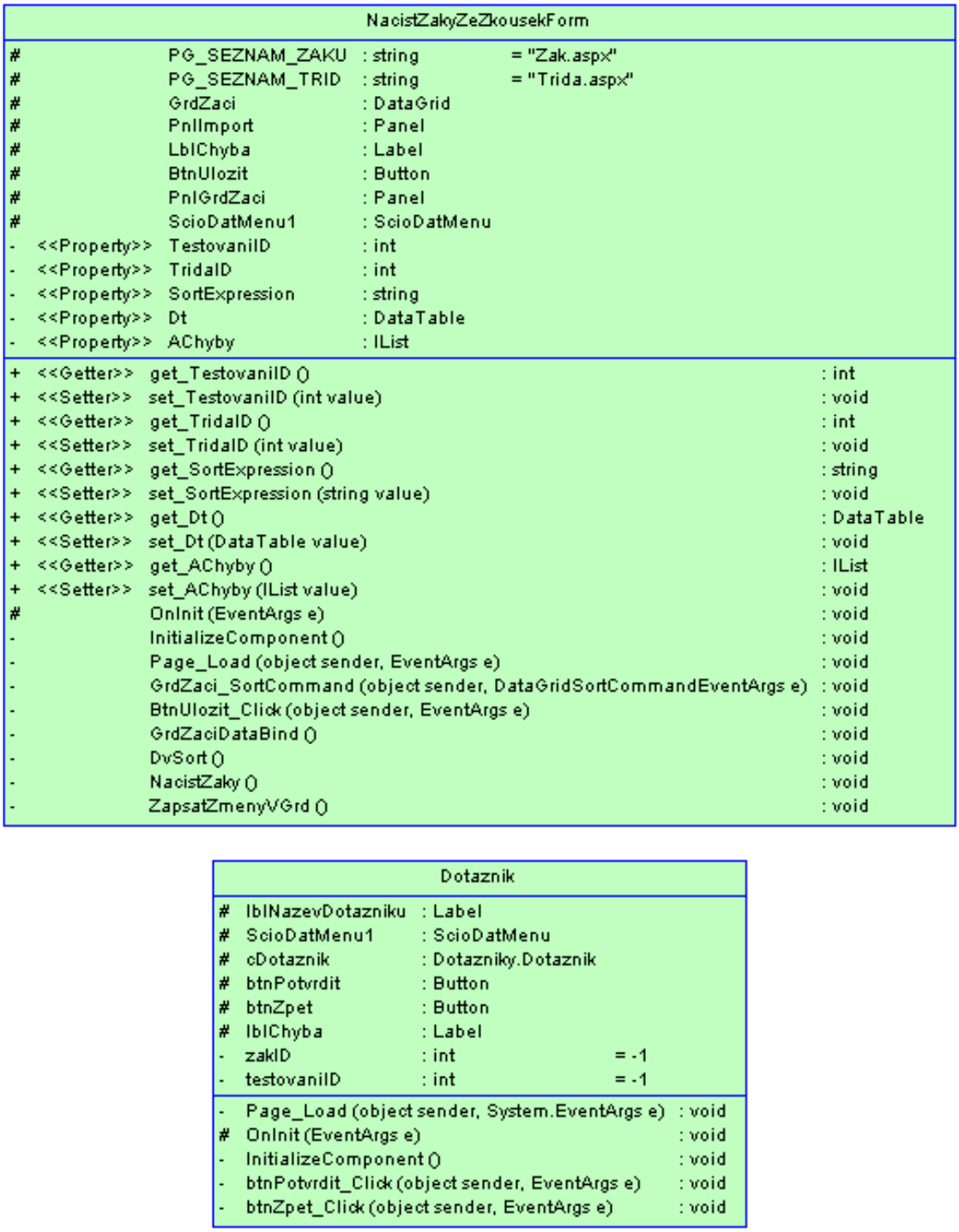
obrázek 49: UML schéma webových formulářů



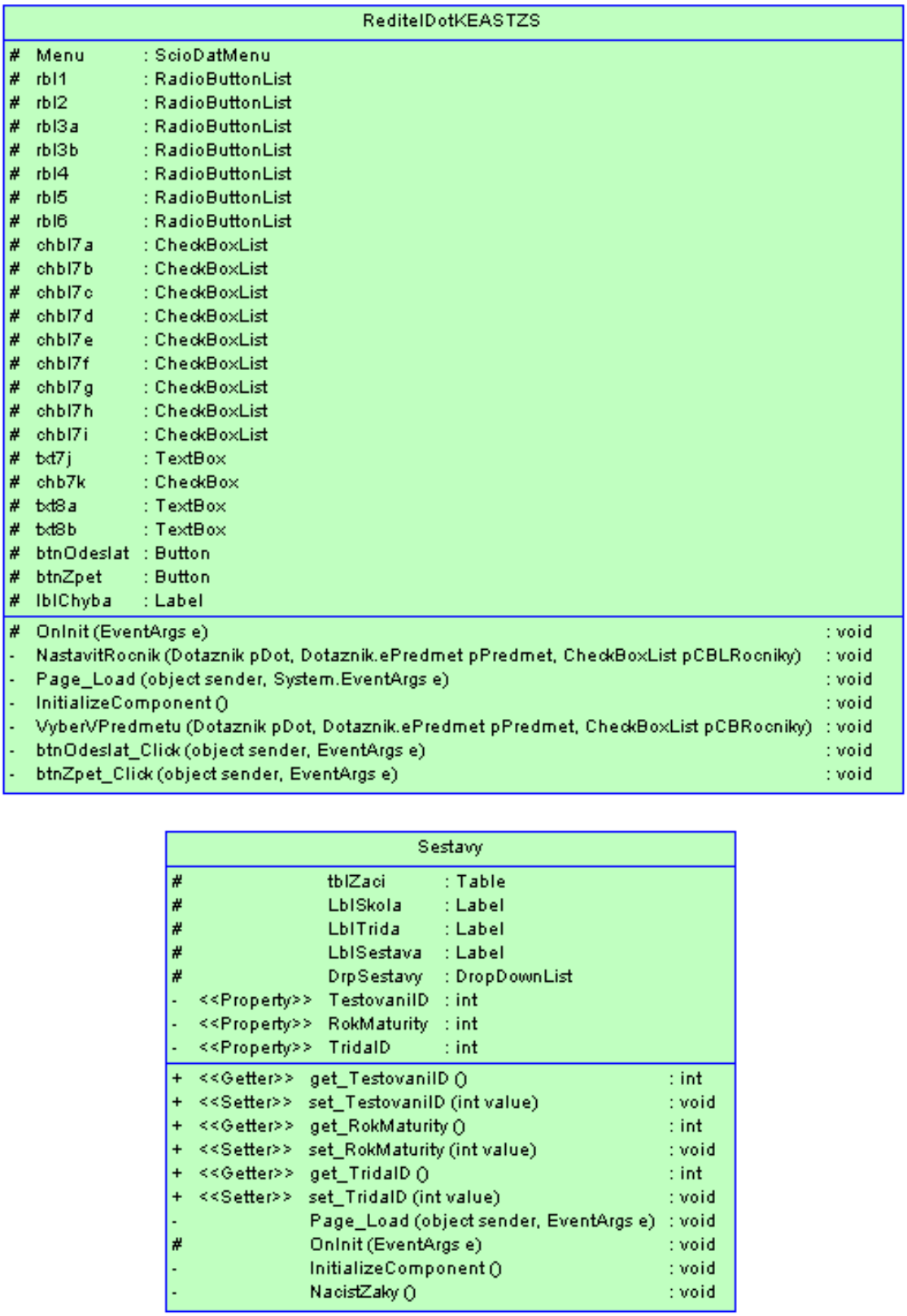
obrázek 50: UML schéma webového formuláře



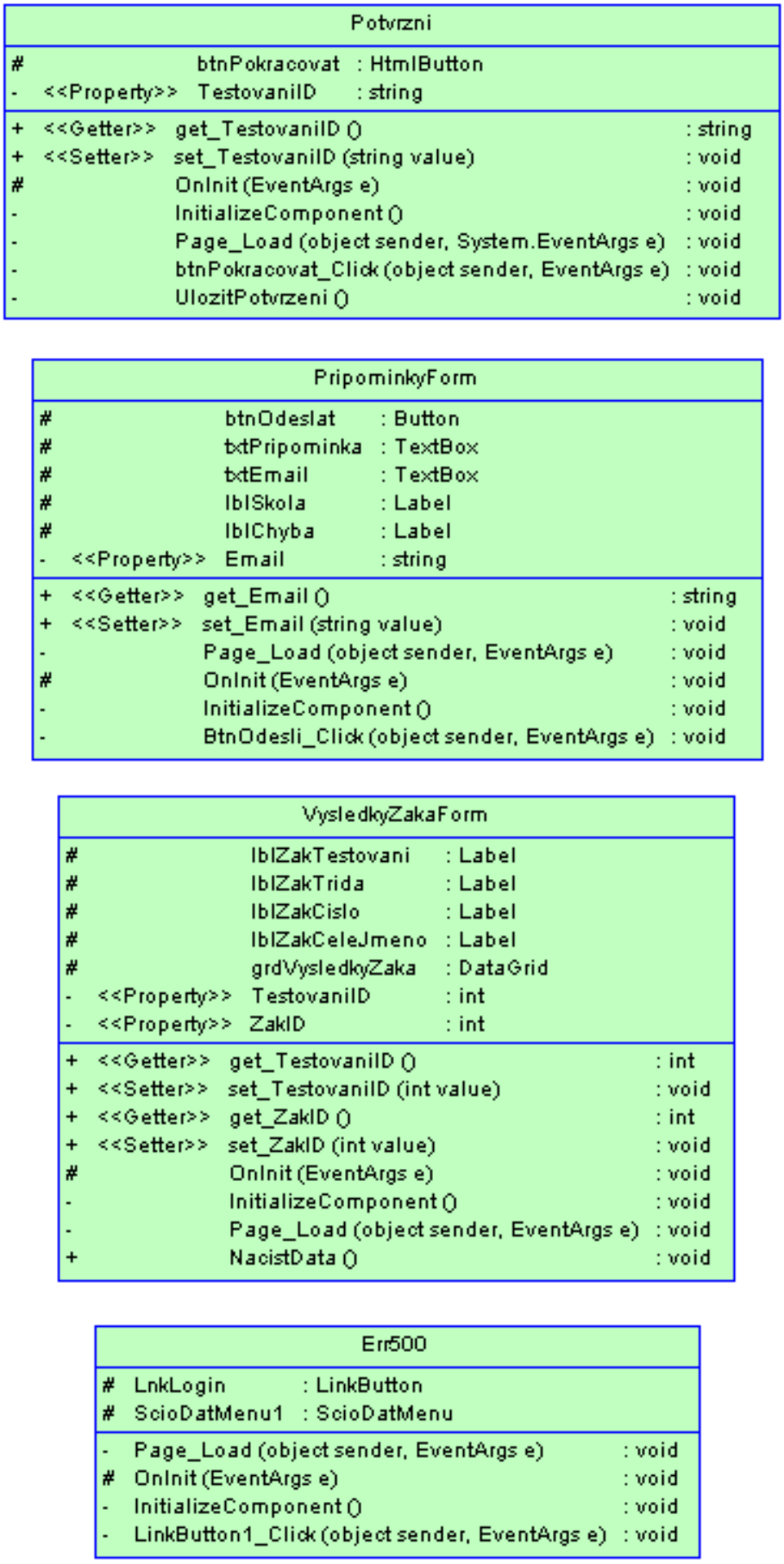
obrázek 51: UML schéma webového formuláře



obrázek 52: UML schéma webových formulářů



obrázek 53: UML schéma webových formulářů



obrázek 54: UML schéma webových formulářů