Jihočeská univerzita v Českých Budějovicích Přírodovědecká fakulta

Vývoj softwarové aplikace pro zpracování vzorků planktonu

Bakalářská práce

Vladimír Kročák

Vedoucí práce: RNDr. Jaroslav Icha

České Budějovice 2016

Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

ZADÁVACÍ PROTOKOL BAKALÁŘSKÉ PRÁCE

Student: Vladimír Kročák
(jméno, příjmení, tituly)

Obor – zaměření studia: Aplikovaná informatika

Katedra/ústav, kde bude práce vypracovávána: Ústav aplikované informatiky

Školitel: Jaroslav Icha, RNDr.,
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Garant z PřF:
(jméno, příjmení, tituly, katedra – jen v případě externího školitele)

Školitel – specialista, konzultant: Michal Šorf, RNDr., Ph.D.
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Téma bakalářské práce: Vývoj softwarové aplikace pro zpracování vzorků planktonu

Cíle práce:

Hlavním cílem bakalářské práce je vytvořit softwarovou aplikaci, která bude sloužit pro rutinní i pokročilé zpracování vzorků planktonu. Při vývoji aplikace bude nutné zohlednit především tyto dílčí cíle:

- Hlavní funkcí aplikace bude určování počtů jedinců jednotlivých druhů živočichů obsažených ve studovaných vzorcích
- Aplikace musí mít uživatelsky příjemné rozhraní, které umožní efektivně a pokud možno bez chyb provádět počítání jedinců ve vzorcích
- S ohledem na předpokládané budoucí nasazení musí mít aplikace české a anglické uživatelské rozhraní
- Aplikace musí umožňovat provádět výpočet výsledných hustot organismů v závislosti na metodě odběru vzorku
- Aplikace musí být navržená tak, aby v budoucnu bylo možné její další rozšiřování o nové funkce či modifikace funkcí stávajících
- Vstupem pro provádění výpočtů jsou vzorky planktonu, pro které musí být navržen vhodný datový model umožňující archivaci vstupů a získaných výsledků v databázi
- Aplikace musí umožňovat export výsledků v různých výstupních formátech dle požadovaného dalšího využití
- Projekt bude koncipován jako open source projekt

Základní doporučená literatura:

Edmonson, W. T.Edmonson, W. T., ed., (1971), *A manual on methods for the assesment of secondary productivity in fresh waters. IPB Handbook N° 17*, Blackwell Scientific Publications, Oxford, chapter Methods for processing samples and developing data, pp. 127—137.

Vedoucí práce: Jaroslav Icha podpis:

Vedoucí oddělení/katedry/ústavu, kde bude práce vypracovávána:

Ing. Jan Fesl podpis:

V Českých Budějovicích dne 30. září 2014 podpis studenta:

Bibliografické údaje

Kročák Vl., 2015: Vývoj softwarové aplikace pro zpracování vzorků planktonu. [Development of software application for sample processing. Bc.. Thesis, in Czech.] – 64 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Tato bakalářská práce se zabývá vytvořením nové aplikace využívané ke kvantitativnímu zpracování vzorků vodních bezobratlých živočichů. Aplikace má nahradit v současnosti používané řešení vytvořené v letech 1997-1999 prof. RNDr. Adamem Petruskem, Ph.D. Výsledná aplikace bude zároveň sloužit jako platforma pro další potřeby Katedry biologie ekosystémů Přírodovědecké fakulty Jihočeské univerzity v Českých Budějovicích. Aplikace a její zdrojové kódy budou následně uveřejněny pod svobodnou licencí (MIT) tak, aby mohl být zajištěn její další vývoj.

Annotation

This bachelor thesis deals with the implementation of a new application for quantitative processing of zooplankton samples and replaces current application in use developed by prof. RNDr. Adam Petrusek, Ph.D. The application will also be used as a framework for further needs of the Department of Ecosystem Biology, Faculty of Science, University of South Bohemia. The application and its source code will be released under MIT license to allow its further development.

Prohlášení

Prohlašuji, že jsem svoji bakalářskou práci vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích, dne 18. dubna 2016

Podpis _____

Poděkování

Zvláště bych chtěl poděkovat všem, kteří vytvořili knihovny použité při vývoji této aplikace.

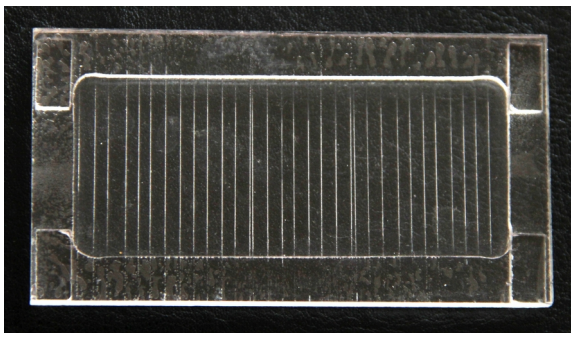
Rád bych poděkoval tvůrcům z freesound.org, jejichž kreativní tvorba byla použita přímo ve vyvíjené aplikaci.

Zároveň bych chtěl poděkovat vedoucímu práce RNDr. Jaroslavu Ichovi za odbornou pomoc při tvorbě této práce a Mgr. Michalu Šorfovi Ph.D za průběžné testování a trpělivost.

Obsah

1 Úvod

V současné době se kvantitativní zpracování vzorků vodních bezobratlých (zejména zooplanktonu) na Katedře biologie ekosystémů Přírodovědecké fakulty Jihočeské univerzity (dále jen KBE) provádí především ručně: odebraný objem vzorku je koncentrován a pracovníkem mikroskopicky analyzován. Z koncentrovaného vzorku se odebírají přesné podíly (podvzorky), jejichž objem je prohlédnut z hlediska druhového určení a početností jednotlivých organismů. Mikroskopické zpracování probíhá často v počítacích komůrkách vybavených mřížkou pro snadnější orientaci ve vzorku.



Obr. 1 Obrázek počítací komůrky

Do komůrek se pipetuje přesný objem, který spolu se zjištěným počtem organizmů v jednotlivých komůrkách a známým objemem původního vzorku umožňuje dopočítat průměrné zastoupení a hustotu jednotlivých druhů.

Tento způsob měření patří mezi nejpoužívanější způsoby provedení podobného druhu analýzy, ať už se jedná o potřebné technické prostředky nebo samotné určování druhového složení zooplanktonu.

Doposud se pro udržování počtu v průběhu analýzy a následných výpočtů zpracovaných vzorků planktonu na KBE používala aplikace *„POČÍTAČ PLANKTONU a nebo v zásadě čehokoliv© 1997-9”* (dále jen původní aplikace) vytvořená prof. RNDr. Adamem Petruskem, Ph.D.

Tato aplikace byla vytvořena pro operační systém MS-DOS. Tato aplikace tedy není již kompatibilní se současnými operačními systémy bez použití virtualizace nebo emulačního softwaru. Samotná aplikace je tedy jak po stránce technologické, tak vlastního rozhraní již zastaralá. Její další vývoj není možný.

Hlavním výstupem této práce je aplikace, která nahradí původní aplikaci a zároveň která bude i po samotném odevzdání této práce nadále vyvíjena jako openSource projekt, a tedy bude možné přizpůsobovat budoucím potřebám KBE. Tento projekt bude možné využit i jako platformu pro další studentské práce nebo pro další rozšíření.

V rámci této práce bude implementována pouze náhrada za manuální rozhraní pro udržování přehledu o naměřeném počtu organizmů a průběhu měření. Zároveň bude implementována možnost spravovat měřené organizmy a lokality, a také možnost exportovat již hotová měření pro další zpracování.

Preferencí před exportem nutným pro další zpracování by měla aplikace umožňovat provádět potřebné analýzy přímo ve svém rozhraní.

2 Cíle práce

Hlavním cílem této práce je vytvořit samotnou softwarovou aplikaci (dále již jen aplikace), která bude sloužit pro rutinní i pokročilé (export a analýza vzorků v aplikaci, atd.) zpracování vzorků planktonu.

Výsledná aplikace musí splňovat následující kritéria:

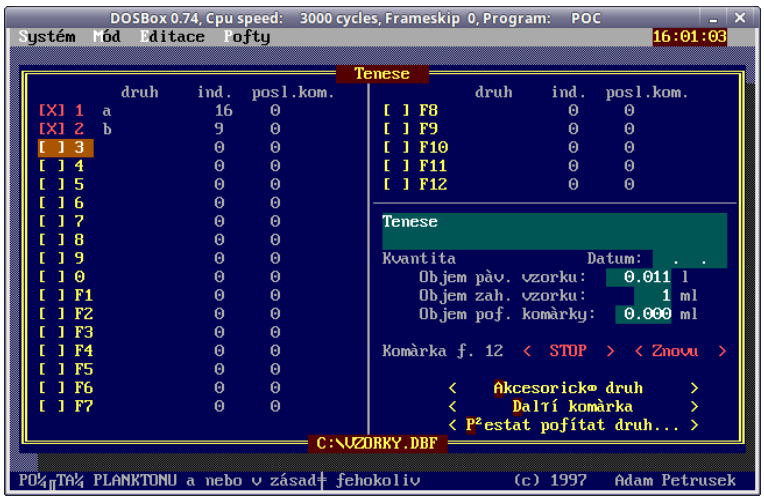
- Hlavní funkcí aplikace bude určování počtů jedinců jednotlivých druhů živočichů obsažených ve studovaných vzorcích.
- Aplikace musí mít uživatelsky příjemné rozhraní, které umožní efektivně a pokud možno bez chyb provádět počítání jedinců ve vzorcích.
- S ohledem na předpokládané budoucí nasazení musí mít aplikace české a anglické uživatelské rozhraní.
- Aplikace musí umožňovat provádět výpočet výsledných hustot organismů v závislosti na metodě odběru vzorku.
- Aplikace musí být navržená tak, aby v budoucnu bylo možné její další rozšiřování o nové funkce či modifikace funkcí stávajících.
- Vstupem pro provádění výpočtů jsou vzorky planktonu, pro které musí být navržen vhodný datový model umožňující archivaci vstupů a získaných výsledků v databázi.
- Aplikace musí umožňovat export výsledků v různých výstupních formátech dle požadovaného dalšího využití.
- Projekt bude koncipován jako open source projekt.

3 Přehled současného stavu řešené problematiky

V současné době se většina aplikací zabývajících se podobnou problematikou, zaměřuje především na automatické sčítání kolonií v laboratorním prostředí. Automatické zpracování fotografií vzorků zooplanktonu je možné, ale nepatří k nejčastějším řešením. Hlavním požadavkem KBE bylo nahradit současnou aplikaci, která byla doposud používána a která poskytovala asistenci pouze pro manuální zpracování vzorku.

POČÍTAČ PLANKTONU a nebo v zásadě čehokoliv

Původní aplikace, byla vyvinuta pro operační sytém MS-DOS (respektive pro Windows 3.1-98, které umožňují nativně spouštět aplikace pro MS-DOS).



Obr. 2 Původní aplikace

Hlavní součástí této aplikace je počítadlo, které počítá stisky kláves a k nim asociované druhy mikroorganizmů (dále jen manuální rozhraní). Tímto si laborant udržuje přehled o počtech jednotlivých druhů v průběhu laboratorního měření.

Aplikace si interně udržuje seznam druhů mikroorganizmů a již hotových vzorků, které jsou uloženy v aplikaci v databázových souborech (mime type: application/x-dbf).

V této aplikaci je pouze umožněno zobrazení vzorků přímo v aplikaci a jejich export do CSV souborů. Další možnosti zpracování tato aplikace nenabízí.

4 Přehled použitých technologií

Tato kapitola uvádí přehled většiny použitých technologií a důvod jejich použití. V případě JPA a SL4J jsou uvedeny i konkrétní použité implementace těchto frameworků. Tato kapitola zároveň slouží i k vysvětlení některých pojmů, které budou použity dále v textu práce.

4.1 Apache Maven

Apache Maven (dále jen Maven) je nástroj vyvíjený organizací Apache Software Foundation pro správu a řízení vývoje nejenom Java projektů.

Tento nástroj byl vyvinut mimo jiné za účelem, zajistit jednotný proces buildování, sjednocení adresářové struktury projektu, celkového řízení životního cyklu projektu a řízení závislostí [1].

Maven byl využit pro správu závislostí, k vygenerování úvodní struktury aplikace na základě archetypu, a také k vygenerování softwarové licence.

Adresářová struktura aplikace:

`src/main/java`	- adresář se zdrojovými soubory aplikace
`src/main/resources`	- adresář s ostatními soubory(.fxml, properties, atd.)
`src/test/java`	- adresář se zdrojovými soubory testů
`src/test/resources`	- adresář se soubory pro potřeby testování
`pom.xml`	-“Project Object Model“ deskriptor projektu

Descriptor projektu (*pom.xml*) se nachází v top-level adresáři a je automaticky vyhledáván Mavenem. Tento soubor obsahuje veškeré informace o způsobu kompilování, testování a způsobu buildu aplikace. Zároveň obsahuje seznam požadovaných závislostí. Tento soubor je přibalen i do výsledné aplikace. Slouží k celkové identifikaci projektu [2].

4.2 Git

Při vývoji byl použit verzovací nástroj Git. Jedná se o systém pro správu verzí. Původně vytvořený Linusem Torvaldsem, který je šířen pod GPL v.2. Díky použití tohoto nástroje bude zjednodušen i další vývoj a distribuce vytvořené aplikace [3].

4.3 Java

Jako programovací jazyk a platforma pro novou aplikaci byla vybrána Java.

Jedná se o objektově orientovaný programovací jazyk, jehož platforma je nezávislá na operačním systému a je tedy vhodný pro přenositelné aplikace [4].

4.3.1 JavaFX

JavaFX je soubor grafických a mediálních knihoven vytvořený speciálně pro tvorbu bohatých klientských aplikací (Rich Client Applications). Od verze Javy 7u6 je přímo součástí JDK a běhového prostředí (JRE). Umožňuje definovat layout aplikace jak přímo v kódu třídy pomocí Java API, tak i deklarací ve FXML dokumentech, ze kterých je následně načten celý layout [5].

Použité prvky v aplikaci vyžadují nainstalovanou verzi JavaFX 8u60 a vyšší.

FXML

Jedná se o soubory ve formátu XML, který obsahují deklaraci layoutu pro JavuFX. Celý layout lze pak načíst přímo z těchto souborů.

```
<GridPane xmlns:fx="http://javafx.com/fxml/1"
    xmlns="http://javafx.com/javafx/8"fx:controller="my.form.Controller" >
    <children>
        <Label fx:id="labelName" text="%name" />
            GridPane.rowIndex="1" />
        <TextField fx:id="nameInput" GridPane.columnIndex="1" />
    </children>
<! ...
```

Příklad 1 Příklad deklarace formuláře ve FXML

Vytváření vlastních komponent

Oba přístupy definování layoutu pomocí FXML a Java API lze kombinovat při vytváření vlastních komponent (custom component), kdy je layout definován pomocí FXML, který je následně načten v konstruktoru objektu, jehož layout deklaruje.

Pro vytvoření takové komponenty je nejprve nutné v FXML souboru použít kořenový element s tagem *fx:root* (viz. Příklad 2).

```
<fx:root type="javafx.scene.layout.Vbox" xmlns:fx="http://javafx.com/fxml">
    <Label fx:id="name"/>
    <Button text="%click" onAction="#clickAction"/>
</fx:root>
```

Příklad 2 Příklad deklarace s tagem fx:root

Následně je layout načten v konstruktoru třídy (viz. Příklad 3).

```
public class CustomComponent() extends Vbox{

    public CustomComponent(){
        FXMLLoader fxmlLoader =
           new FXMLLoader(getClass().getResource("/fxml/..."));
        fxmlLoader.setRoot(this);
        fxmlLoader.setController(this);
        try {
            fxmlLoader.load();
    //...
```

Příklad 3 Ukázka načtení layoutu z FXML

Následně můžeme pracovat s takovým objektem stejně, jako s kterýmkoli nativním objektem poskytovaným přímo JavouFX.

JavaFX Properties

Obvykle se v Javě k hodnotám objektů přistupuje přes sadu známých metod (tzv. JavaBean). V JavěFX přibyla možnost provázat hodnoty objektů s GUI přímo, použitím objektů z balíčku *javafx.beans.property.** (dále jen properties). Tyto objekty umožňují reagovat na libovolnou změnu pomocí bindování a nebo listenerů.

4.3.2 Java Persistence API (JPA)

JPA je framework pro objektově relační mapování (dále jen ORM). Je součástí specifikace JavaEE. Tento framework je ale plně použitelný i pro aplikace psané pro JavuSE.

Mezi nejznámější implementace patří:

- Hibernate (RedHat)
- EclipseLink (The Eclipse Foundation)
- TopLink (Oracle)

EclipseLink

Pro implementaci aplikace byl použit EclipseLink vyvíjený Eclipse Fundation. V roce 2006 nahradil TopLink jako referenční implementaci a je používán například v aplikačním serveru GlassFish.

```
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>${eclipselink.version}</version>
</dependency>
```

Příklad 4 Deklarace Maven závislosti na EclipseLink

4.3.3 Simple Logging Facade for Java (SLF4J)

SL4J je framework, poskytující jednotné rozhraní, pro většinu používaných logovacích frameworků, jako je například Java Util Logging, Log4J, Logback, atd [6].

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
```

Příklad 5 Deklarace Maven závislosti na SLF4J

Díky přístupu pouze přes rozhraní SL4J, je snadné změnit konkrétní použitý framework. SLF4J nijak nezasahuje do nastavení konkrétního frameworku. V případě, že konkrétní framework není sám kompatibilní přímo se SLF4J, je nutné poskytnout vhodný adapter [9].

```
private static final Logger LOGGER = LoggerFactory.getLogger(HelloWorld.class);
//...
LOGGER.info("Hello {}", /*...*/);
```

Příklad 6 Ukázka logování s použitím SLF4J

Logback

Pro samotné logování byl použit framework Logback [7]. Tento framewok sám o sobě nativně implementuje potřebné třídy pro SLF4J, které přímo používá jako své vlastní rozhraní a není tedy nutné přidávat závislosti na další knihovny.

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
```

Příklad 7 Deklarace Maven závislosti na frameworku Logback

Logback je pokračovatelem frameworku Log4J. Jeho konfigurace je vždy uložena v souboru *logback.xml*.

4.4 Použitá databáze H2

Jako databáze pro aplikaci byla použita relační databáze H2, která je plně implementována v Javě. H2 byla použita především díky možnosti pracovat plně v embedded režimu ([8]), což byl jeden z hlavních požadavků pro vytvářenou aplikaci.

Díky použití JPA aplikace nebude vázána na tento konkrétní typ databáze.

4.5 Vývojové prostředí

Jedním z hlavních důvodů pro použití Mavenu pro vyvíjenou aplikaci bylo docílit co největší nezávislosti na použitém vývojovém prostředí a dalších nástrojů specifických pro tyto aplikace.

Zdrojový kód aplikace by měl tedy být plně přenositelný i mezi vývojovými prostředími.

NetBeans IDE 8.0.2

Pro tvorbu aplikace bylo použito NetBeans IDE 8.0.2 (osobní preference autora). V této práci bylo použito barevné schéma zdrojového kódu, převzaté z uvedené aplikace.

JavaFX Scene Builder 2.0

JavaFX Scene Builder je vizuální nástroj pro snadné a rychlé generování layoutu ve formě FXML dokumentů [5].

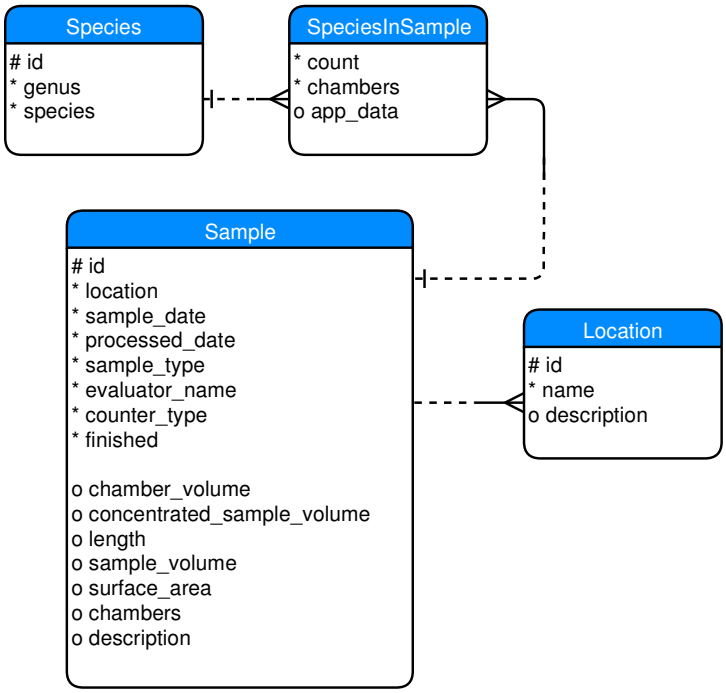
draw.io

Pro generování grafů a diagramů v této práci byl použit on-line nástroj, který lze nalézt na stránkách https://www.draw.io/.

5 Návrh řešení

Tato kapitola popisuje architekturu aplikace, některé klíčové komponenty a GUI.

5.1 Entitně relační model



Obr. 3 Entitně relační model

Samotný datový model, který popisuje data generovaná v průběhu počítání se skládá pouze ze čtyř entit (viz. Obr. 3). Díky JPA mapování na objekty mohou být použity v celé aplikaci přímo, jako jejich objektové varianty.

Druh (Species)

Druh (Species) představuje seznam jednotlivých organizmů, které může laborant pozorovat v průběhu měření. Samotná entita neobsahuje nic víc než rodové a druhové jméno. V budoucnu je počítáno s jejím rozšířením o popis.

Lokality (Location)

Lokality (Location) představuje jednotlivé místa odběru vzorku, případně může oddělovat kontrolované skupiny.

Vzorek (Sample)

Vzorek (Sample) představuje samotný záznam o měření. Kromě povinných parametrů, které zaznamenávají kdy a kde byl vzorek odebrán a záznamu o jeho samotném zpracování, obsahuje i hodnoty potřebné pro další výpočty (např. objem odebraného vzorku).

Některé hodnoty nejsou záměrně označeny v modelu jako povinné. Nutnost jejich vyplnění závisí převážně na typu vzorku (způsobu odběru), a následném způsobu vyhodnocení. Vyplnění těchto hodnot musí být zajištěno na aplikační úrovni.

Seznam druhů (SpeciesInSample)

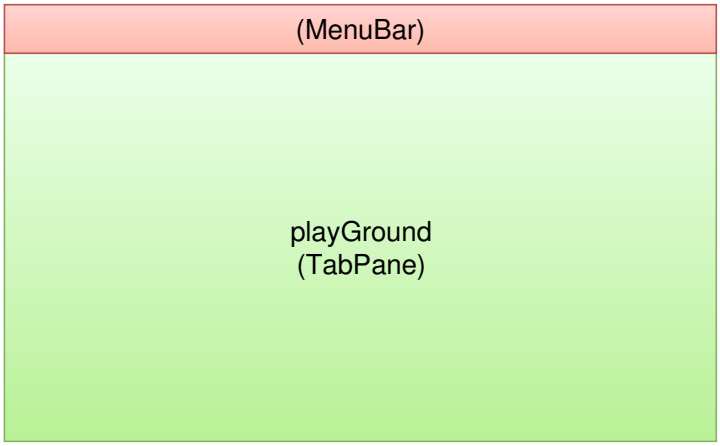
Seznam druhů a jejich četností v rámci jednotlivých vzorků (*SpeciesInSample*) představuje naměřené výsledky. Obsahuje naměřený počet komůrek (vychází ze způsobu měření). Z těchto hodnot jsou následně počítány další hodnoty, jako je průměrné zastoupení a hustotu jednotlivých druhů.

Samotný průběh měření bude vyžadovat další nastavení a hodnoty (dále jen aplikační data). Tyto data obecně nepředstavují výsledek a v případě rozšíření aplikace o zcela jiný druh rozhraní počítadla se mohou navzájem lišit.

Uložení aplikačních dat je třeba i v případě přerušení měření a jeho následném navázání. Proto tyto důvody, jsou alespoň uloženy v serializované podobě (atribut app_data).

Aplikační data mohou zároveň spolu s ostatními daty sloužit k rychlé konfiguraci nového měření.

5.2 Návrh a rozložení grafického rozhraní



Obr. 4 Grafický návrh hlavního okna aplikace

Grafické rozhraní (dále jen GUI) aplikace se bude skládat pouze z menu (*MenuBar*), které umožní uživateli ovládat aplikaci. Zbytek okna bude zabírat jediný prvek a tím je *javafx.scene.control.TabPane*.

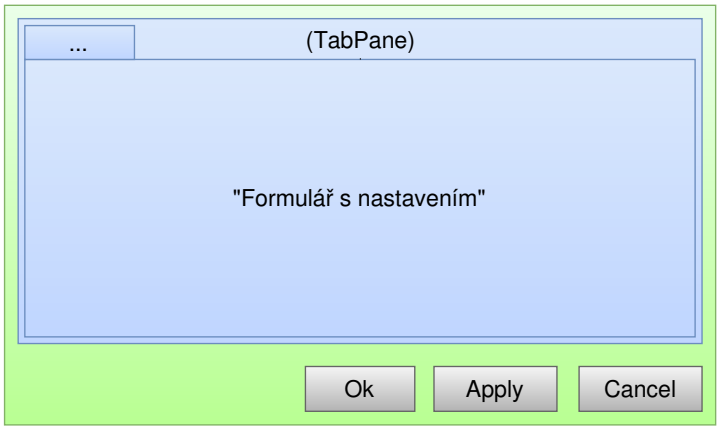
TabPane umožňuje otevírání více na sobě nezávislých záložek zároveň a jejich snadné přepínání. Zbytek aplikace bude tedy především využívat právě této možnosti a zbylé hlavní funkční celky budou implementovány pomocí vlastních záložek.

Přehled dat

Aplikace bude obsahovat samostatné záložky pro správu dat, jejichž cílem je nahradit nutnost použití dalších nástrojů pro správu lokalit, druhů a již vytvořených vzorků. Bude se jednat o jakési administrativní rozhraní aplikace.

Nastavení

Aplikace bude obsahovat samostatnou záložku s uživatelským nastavením.



Obr. 5 Návrh GUI pro nastavení aplikace

Z důvodu, že se předpokládá růst možností nastavení, samotný celek bude opětovně implementován pomocí *TabPane*, kde jednotlivé záložky budou obsahovat nastavení pro jednotlivé části aplikace.

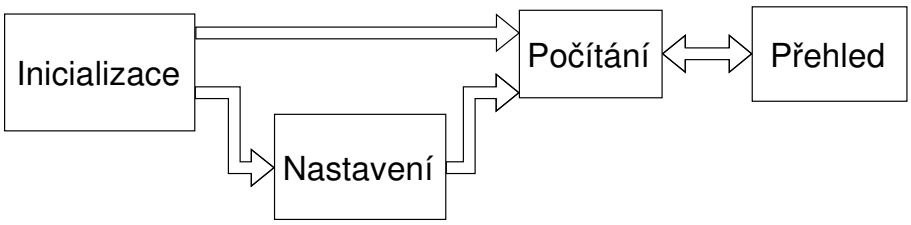
V dolní části se budou nacházet společná tlačítka pro ukládání změn.

5.2.1 Manuální počítadlo

Samotné manuální počítadlo nahrazuje funkcionalitu původní aplikace. Umožňuje tedy laborantovi vytvářet nové záznamy o jednotlivých měření a udržovat aktuální hodnoty během měření samotného.

Samotné měření provádí laborant pomocí mikroskopu a aplikace slouží k udržování aktuálního naměřeného počtu mikroorganizmů pomocí klávesových zkratek.

Proces měření bude koncipován jako průchod wizardem. Samotný průběh je rozdělen do čtyř fází.

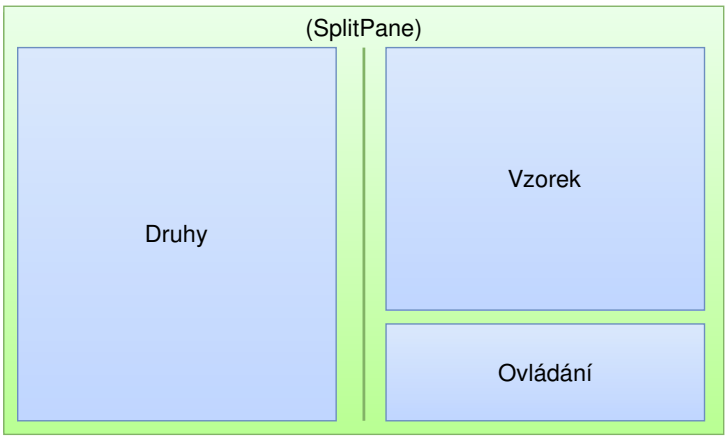


Obr. 6 Průběh průchodu manuálním počítadlem

Mezi posledními dvěmi fázemi by mělo být umožněno libovolně přepínat. Jakmile, ale bude měření v poslední fázi řádně ukončeno, měl by být vzorek označen jako dokončený a nemělo by být umožněno v něm pokračovat.

Uživateli by mělo být umožněno kdykoliv přerušit měření a ukončit aplikaci bez dokončení vzorku a uložit rozpracovaný vzorek do databáze tak, aby bylo možné v něm pokračovat později. Může být umožněno i samotné ukládání v průběhu měření (manuální nebo automatické), aby bylo zabráněno ztrátě dat v případě pádu nebo násilného ukončení aplikace.

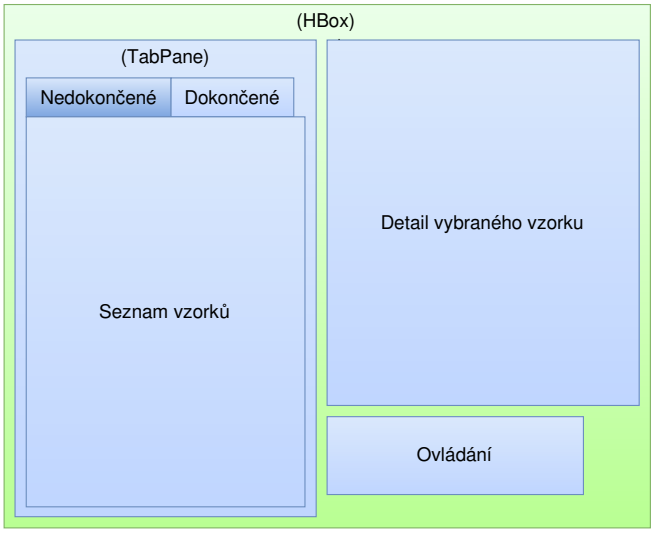
Kromě první fáze bude obrazovka vždy rozdělena do dvou částí. Levá strana bude vždy obsahovat zobrazení měřených druhů, pravá bude věnována informacím o aktuálním vzorku a ovládacím prvkům.



Obr. 7 Návrh grafické rozdělení počítadla

Inicializace

První krok slouží především jako přehled starších hotových či nedokončených měření. Z těchto měření bude moci laborant libovolně pokračovat, nebo zkopírovat nastavení pro nové měření.



Obr. 8 Návrh GUI prvního kroku manuálního počítadla

Místo prvků zobrazujících hodnoty druhů, jak bylo naznačeno v základním rozložení (viz. Obr. 7), obsahuje v levé části přepínatelné seznamy hotových a rozpracovaných vzorků. V těchto vzorcích bude moci laborant libovolně pokračovat či je použít jako předlohu pro nové měření.

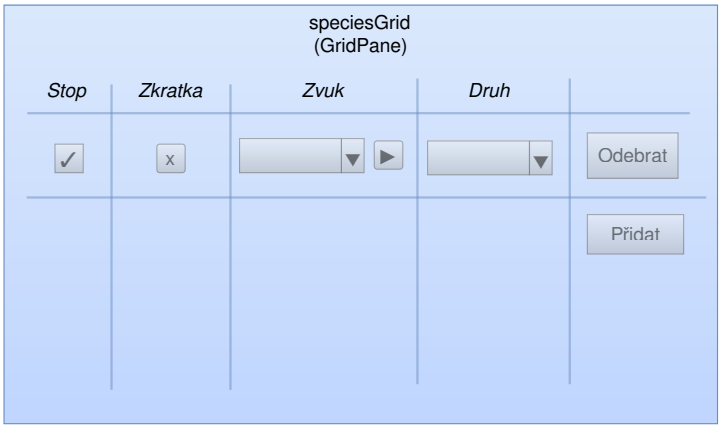
Kromě samotných seznamů vzorků a ovládacích prvků jsou zde zobrazeny i obecné informace o aktuálně vybraném vzorku. V případně nedokončených vzorků může být dovolena i jejich editace, ale v tomto případě se jedná spíše o doplňkovou funkcionalitu. Samotná editace vzorků by měla být řešena samostatně, případně pouze v průběhu měření.

Nastavení měřeného vzorku

V druhém kroku budou nastavovány hodnoty nového vzorku. Nastaveny boudou základní hodnoty popisující vzorek a bude připraven seznam druhů, které má laborant zájem počítat nebo předpokládá jejich výskyt.

V případě, že požadovaný druh, nebo lokality nebyly do aplikace doposud zaneseny, mělo by být uživateli umožněno je doplnit.

Nebudou-li vyplněny všechny potřebné hodnoty, měl by být uživatel upozorněn. Zároveň by mu mělo být zabráněno zadání více stejných druhů mikroorganizmů a použití stejných klávesových zkratek.



Obr. 9 Návrh rozhraní pro nastavení druhů

Nastavení jednotlivých druhů v levé části záložky (viz. Obr. 9) kromě výběru samotného druhu bude obsahovat nastavení klávesové zkratky, pod kterou bude druh uložen, a zvukový signál, který bude přehrán vždy při stisku přiřazené klávesové zkratky. Tento krok zároveň umožňuje předem vyřadit konkrétní druh z počítání (klávesová zkratka bude vyřazena z provozu a nebude zvyšován počet prozkoumaných komůrek).

Počítání

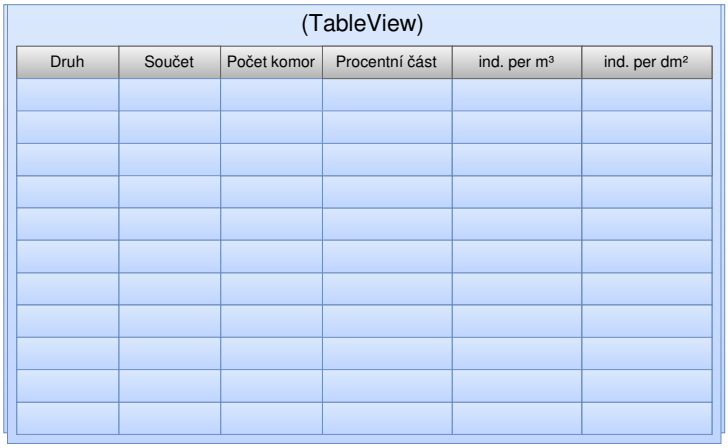
Hlavní funkcionalitou třetího kroku je zaznamenávání počtu stisků kláves, a tedy i počet organizmů přiřazeným k těmto zkratkám. V pravé části bude opět zobrazen přehled obecných informací o vzorku.

Obr. 10 Zobrazení druhů v průběhu počítání

V levé části se tentokrát bude nalézat seznam měřených druhů s jejich naměřenými počty (viz. Obr. 10). Stejně jako v předešlé fázi je u každého druhu umožněno vyřadit ho z průběhu počítání a dočasně tak přerušit jeho počítání.

Přehled

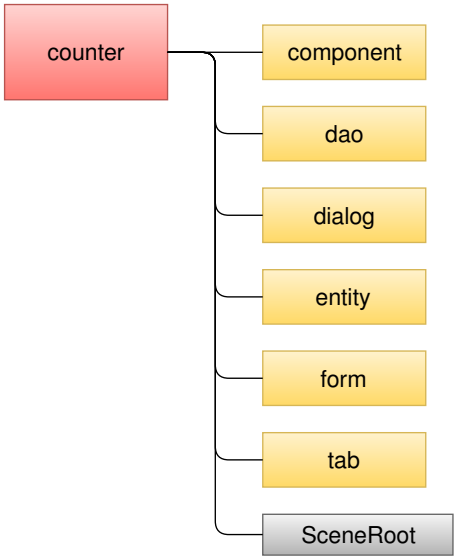
V poslední fázi budou zobrazeny výsledné vypočítané hodnoty v tabulce (viz. Obr. 11). Takové zobrazení vypočítaných hodnot vzorku bude použito například i v exportu nebo detailech již hotových vzorů.



Obr. 11 Návrh zobrazení vypočítaných hodnot

6 Implementace

Tato kapitola obsahuje základní nebo nejdůležitější implementované prvky samotné aplikace.



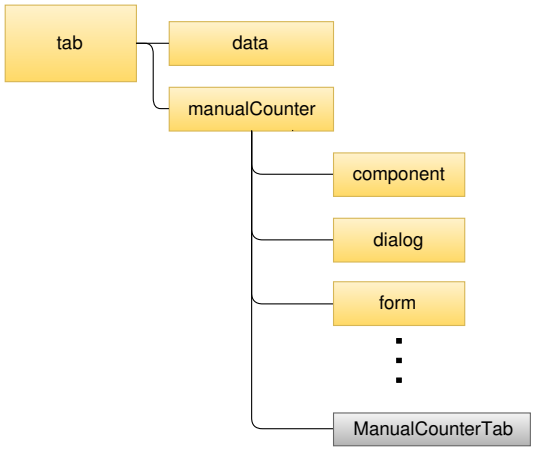
Obr. 12 Základní rozdělení aplikace

Aplikace je rozdělena na několik částí, které kopíruje i samotná struktura zdrojového kódu (viz. Obr. 12).

Základní komponenty jako jsou entitní třídy (*cz.jcu.prf.counter.entity*), dialogy, formuláře a objekty pro přístup do databáze (Data access object dále jen DAO objekty) se nacházejí v samostatných balíčcích.

Zbylé komponenty nebo grafický layout jednotlivých formulářů lze nalézt v balíčku *cz.jcu.prf.counter.component*. U těchto komponent se počítá s opakovaným použitím v různých částech aplikace. Jedná se především o objekty, které například pouze zdržují grafické komponenty, ale jejich funkcionalita závisí především na konkrétním místě použití.

Hlavní prvkem okna aplikace je *TabPane*, a tedy základními funkčními celky jsou samostatné záložky, které lze nalézt v samostatných balíčích v *cz.jcu.prf.counter.tab*. S těmito záložkami je počítáno jako s prostorem pro další rozšiřování funkcionality aplikace.



Obr. 13 Rozdělení tříd pro jednotlivé záložky

Implementace každé záložky by měla kopírovat strukturu základní aplikace (viz Obr. 12), od přidávání dalších formulářů po přidávání dialogů, atd.

Všechny záložky by měly být na sobě navzájem nezávislé, případně závislé pouze na prvcích ze základní části aplikace.

6.1 Pomocné třídy

Pro snadný přístup k nastavení nebo k překladům byly v aplikaci vytvořeny tři třídy implementující návrhový vzor Singleton (Jedináček), které udržují v paměti načtená nastavení, ke kterým umožňují snadný přístup.

Settings

Třída *cz.jcu.prf.counter.Settings* udržuje interně načtenou konfiguraci aplikace a poskytuje metody pro získání konkrétního nastavení. Používání těchto metod značně usnadňuje případnou modifikaci nastavení.

```
public class Settings {

    //...

    private Settings() {
        try{
            //...
            this.settings = new Properties();
            settings.load(getClass()
            .getResourceAsStream("/properties/config.properties"));
        //...
    }

    public boolean isMute() {
        return Boolean.parseBoolean(this.settings.getProperty("jukebox.mute"));
    }

    public Object setMute(boolean mute) {
        return this.settings.setProperty("jukebox.mute", String.valueOf(mute));
    }
    //...
```

Příklad 8 Ukázka implementace třídy Settings

Samotné použití této třídy je pak následně velmi snadné.

```
if (!Settings.getInstance().isMute()){
//...
```

Příklad 9 Ukázka použití třídy Settings

Translator

Třída *cz.jcu.prf.counter.Translator* zastává funkci interního překladače a obdobně jako *Settings*, udržuje interně načtené překlady. Tato třída na rozdíl od Settings je navržena spíše pro předávání celých překladů pro *FXMLLoader* nebo pro získávání specifických překladový hlášek.

```
public class Translator {

    private static Translator instance;
    private ObjectProperty<ResourceBundle> resources;
    //...

    private Translator() {
        //...
        this.resources = new SimpleObjectProperty<>(ResourceBundle
            .getBundle("translations.lang"));
    }

    public final ResourceBundle getResources() {
        return this.resources.get();
    }

    public String getString(String key) {
        try {
            return this.resources.get().getString(key);
        } catch (... exception) {
            LOGGER.warn("String '{}' not found.", key, exception);
            return "%" + key;
        }
    }
    //...
```

Příklad 10 Ukázka implementace třídy Translator

Jukebox

Jukebox (*cz.jcu.prf.counter.Jukebox*) je výčtovou třídou (enum), které interně udržuje ve svých instancích načtené audio klipy a umožňuje tak přehrát libovolný tón kdekoli v aplikaci.

Při přehrání klipu se automaticky kontroluje ztlumení aplikace poskytované globálním nastavením.

```
//...
private AudioClip clip;

public void play() {
    if (!Settings.getInstance().isMute()) {
        if (this.clip == null) {
            this.clip = new AudioClip(/*...*/);
        }
        this.clip.play();
    }
}
```

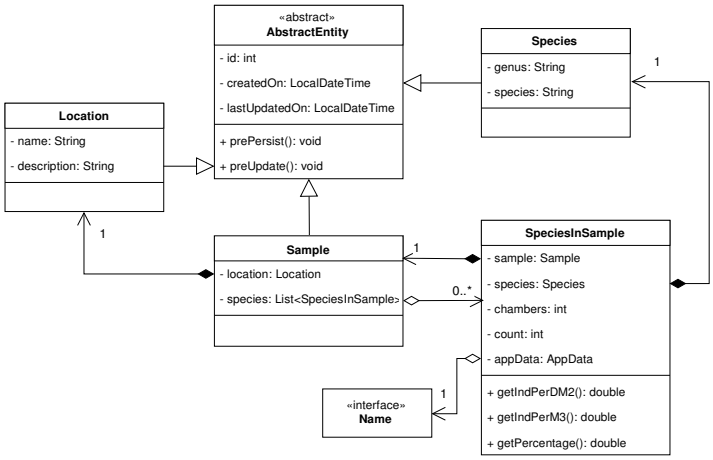
Příklad 11 Ukázka implementace třídy Jukebox

Většina klipů byla přidána pro vlastní manuální počítadlo, ale obsahuje i čistě aplikační zvuky a tóny, které jsou využívány v celé aplikaci.

Veškeré zvukové klipy byly získány z freesound.org.

6.2 Entitní třídy

Veškeré datové třídy lze nalézt v balíčku *cz.jcu.prf.counter.entity*. Tento balíček zároveň obsahuje i další třídy a rozhraní, které slouží především ke zpracování dat pomocí JPA nebo pro univerzálnější deklaraci formulářů.



Obr. 14 UML diagram entitních tříd

AbstractEntity

Jako základní kámen byla vytvořena abstraktní entitní třída (*AbstractEntity*), která sjednocuje definici spíše technických atributů jako je id a záznamy o vytvoření a modifikaci.

```
@MappedSuperclass
public abstract class AbstractEntity implements Serializable {
    @Id
    @GeneratedValue
    @Column(name = "id")
    private int id;

    @Column(name = "created_on", nullable = false, updatable = false)
    private LocalDateTime createdOn;

    @Column(name = "last_updated_on", nullable = false)
    private LocalDateTime lastUpdatedOn;

    //...
```

Příklad 12 Ukázka implementace AbstractEntity

Samotné generování časových známek je řešeno pomocí implementace PrePersist a PreUpdate metod poskytovaných přímo JPA, které automaticky nastaví aktuální hodnoty při zpracování pomocí EntityManageru.

```
@PrePersist
private void prePersist() {
    this.createdOn = this.lastUpdatedOn = LocalDateTime.now();
}

@PreUpdate
private void preUpdate() {
    this.lastUpdatedOn = LocalDateTime.now();
}
```

Příklad 13 Generování časových známek

Tato třída zároveň slouží při obecné deklaraci DAO objektů a tříd, které pracují s těmito entitami.

SpeciesInSample

SpeciesInSample slouží především k asociaci vzorku s druhy, které byly naměřeny.

```
@Entity(name = "species_sample")
@IdClass(value = SpeciesInSamplePK.class)
public class SpeciesInSample implements Serializable {

    @Id
    @ManyToOne(targetEntity = Species.class)
    @JoinColumn(name = "species_id")
    private Species species;

    @Id
    @ManyToOne(targetEntity = Sample.class)
    @JoinColumn(name = "sample_id")
    private Sample sample;

    //...
```

Příklad 14 Ukázka implementace SpeciesInSample

Objekty této třídy mají přístup ke všem hodnotám potřebným ke generování samotných výsledků měření (hustota výskytu, atd.). Z tohoto důvodu u ní byli implementovány metody pro jejich samotný výpočet jako dodatečné gettery, což usnadňuje jejich použití přímo v aplikaci.

Tento způsob implementace, především díky jeho snadnému použití, patří spíše k praktičtějším, ale nemusí se vždy jednat o nejefektivnější řešením těchto výpočtů (z pohledu výkonu).

```
public double getIndPerDM2() {
    if (this.sample != null) {
        return ((this.sample.getConcentratedSampleVolume() /
            (this.sample.getChamberVolume() * this.chambers))
            * this.count) / this.sample.getSurfaceArea();
    }
    return -1;
}
```

Příklad 15 Výpočet hustoty organizmu na dm²

```
public double getIndPerM3() {
    if (this.sample != null) {
        return ((this.sample.getConcentratedSampleVolume() /
            (this.sample.getChamberVolume() * this.chambers))
            * this.count)
            / (this.sample.getSurfaceArea() / 100 *
            this.sample.getLength());
    }
    return -1;
}
```

Příklad 16 Výpočet hustoty organizmu na m³

```
public double getPercentage() {
    if (this.sample != null) {
        return (this.getIndPerM3() / this.sample.getSpecies()
        .parallelStream().mapToDouble(s -> s.getIndPerM3()).sum())
        * 100;
    }
    return -1;
}
```

Příklad 17 Výpočet procentního zastoupení organizmu

Další parametry používané v počítadle jsou přidávány pomocí modelových obalových tříd.

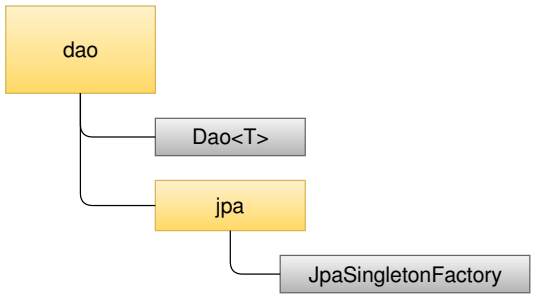
AppData

V průběhu vývoje rozhraní pro manuální počítání vznikl požadavek na možnost přerušit laboratorní činnost a následně na ni později navázat, případně zkopírovat celé nastavení pro nové měření. Z toho to důvodu bylo přidáno do *SpeciesInSample* další pole (*app_data*), které má za účel uchovávat hodnoty z obalových tříd.

Protože by se samotná ukládaná data mohla lišit pro případná nově implementovaná rozhraní, tento atribut slouží k ukládání libovolného serializovatelného objektu, který je uložen do databáze ve formě binárního blobu (Binary Large Object).

6.3 Data access objecs (DAO)

Komunikace s databází v aplikaci byla zprostředkována pomocí samostatných DAO objektů. Tyto objekty sdružují veškeré funkce potřebné pro práci s entitními třídami a databází.



Obr. 15 Rozložení tříd DAO objektů

Pro definici těchto objektů bylo vytvořeno generické rozhraní (*Dao<T>*), které obsahuje pouze nejnutnější hlavičky metod, které umožňují manipulaci s jednotlivými třídami.

```
public interface Dao<T> {
    //...
}
```

Příklad 18 Rozhraní pro komunikaci s databází

Z hlavičky lze poznat, že není kladeno žádné omezení pro kterou třídu toto rozhraní může být použito, ale základním předpokladem je, že se bude jednat o potomka *AbstractEntity*.

Pro vyhledání specifického objektu slouží metoda *getSingle()*, která vyhledá specifický objekt dle jeho id.

```
public T getSingle(int id);
```

Příklad 19 Deklarace metody pro získání konkrétního objektu

Alternativou je metoda, která má za úkol vyhledat seznam objektů dle zadaných kritérií.

```
public List<T> findAll(Map<String, Object> searchCriteria);
```

Příklad 20 Deklarace metody pro získání seznamu objektů

Tato metoda by měla být časem přetížena tak, aby nabízela možnost přidat i parametry pro řazení objektů získaných z databáze (konkrétní ukázka implementace této metody je uvedena na str. 32).

Kromě metod pro uložení a update hodnot v databázi byla po vzoru Hibernate přidána i univerzální metoda, která sama rozhoduje, zda má dojít k uložení nové entity, nebo pouze k uložení změn. Tato metoda by měl být plně zodpovědná za toto rozhodnutí a zjednodušit tak samotné použití DAO objektu.

```
/**
 * Update/persist entity into database
 *
 * @param t
 * @return persisted object
 */
public T saveOrUpdate(T t);
```

Příklad 21 Deklarace metody pro ukládání dat

6.3.1 JPA

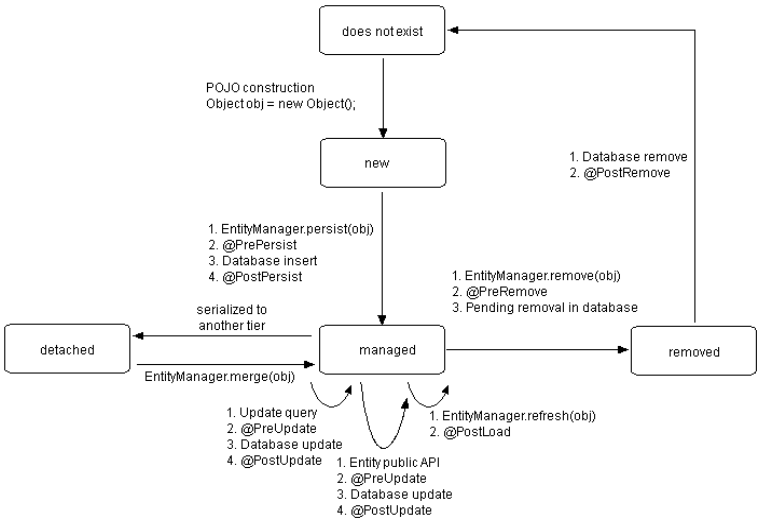
EntityManagerFactory a samotné vytvoření kontextu pro JPA patří k nejnáročnějším činnostem, které musí být provedeny před samotným použitím JPA. Z tohoto důvodu byla vytvořena statická tovární třída (*cz.jcu.prf.counter.dao.jpa.JpaFactory*), která udržuje společný objekt *EntityManagerFactory* a generuje potřebné DAO objekty, kterým je tento kontext předán.

Samotné vygenerované *EntityManagery* nejsou thread-safe a není je možné využít k provádění více paralelních transakcí. Tato skutečnost bohužel brání jejich sdílení v prostředí více vláken.

Naštěstí je *EntityManager* koncipován spíše jako jednoduchý jednorázový objekt a může tak být snadno generován zvlášť pro každý dotaz do databáze.

Bohužel tímto se vytrácí možnost použít stejný *EntityManager* pro komitování změn zpět do databáze nebo určení, jestli se jedná o již načtený objekt.

Z tohoto důvodu a relativně snadné dostupnosti EntityManager objektů byl opuštěn koncept práce s „*managed*“ objekty a všechny načtené objekty jsou automaticky detachovány od databáze.



Obr. 16 Životní cyklus JPA entity [10]

Samotná *EntityManagerFactory* je implementována, jako thred-safe a může tedy být bezpečně předávána přímo pro použití konkrétním DAO objektům.

```
public class JpaFactory {
    private static final EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("persistenceUnit",
        Settings.getInstance().getDatabaseProperties());

    private static Dao<Location> locationDao;

    public static Dao<Location> getLocationDao() {
        if (this.locationDao == null) {
            this.locationDao = new LocationJPADao(this.factory);
        }
        return this.locationDao;
    }

    //...
```

Příklad 22 Ukázka implementace JpaFactory

Samotný DAO objekt následně využívá předanou *EntityManagerFactory* ke generování vlastních *EntityManagerů* pro použití v jednotlivých metodách.

```
@Override
public Location getSingle(int id) {
    EntityManager manager = factory.createEntityManager();
    EntityTransaction tx = manager.getTransaction();
    Location result;
    tx.begin();
    try {
        result = manager.find(Location.class, id);
        tx.commit();
        manager.close();
    } catch (Exception e) {
        //...
     }
    return result;
}
//...
```

Příklad 23 Ukázka použití EntityManageru

Metoda pro vyhledání seznamu objektů

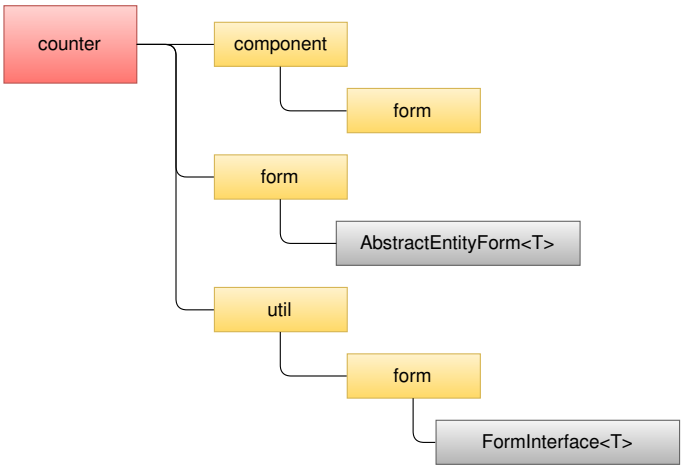
Metoda pro vyhledávání dle seznamu parametrů využívá CriteriaQuery, která je automaticky vytvořena z předaných parametrů metody.

```
@Override
public List<Sample> findAll(Map<String, Object> searchCriteria) {
    //...
    EntityManager manager = this.factory.createEntityManager();
    CriteriaBuilder criteriaBuilder = manager.getCriteriaBuilder();
    CriteriaQuery<Sample> query = criteriaBuilder
        .createQuery(Sample.class);

    //...
    Root<Sample> root = query.from(Sample.class);
    query.select(root);
    List<Predicate> restrictions = new ArrayList<>();
    searchCriteria.forEach((key, value) -> {
        switch (key) {
            case "sampleDateFrom":
                 restrictions.add(criteriaBuilder.gt(
                     root.get("sampleDate"), (LocalDateTime) value)
                    );
            //...
            }
        });
    //...
    query.where(
        criteriaBuilder.and(restrictions.toArray(
        new Predicate[restrictions.size()])));

    TypedQuery<Sample> typedQuery = manager.createQuery(query);

    return typedQuery.getResultList()
}
```

Příklad 24 Ukázka vyhledávání pomocí zadaných parametrů

Tímto způsobem byla vytvořena metoda, která umožňuje snadno získat předem vyfiltrovaný výčet objektů z databáze.

6.4 Formuláře



Obr. 17 Rozložení tříd formulářů

Všechny formuláře rozšiřují základní rozhraní (*FormInterface<T>*), které zajišťuje základní ovládání těchto formulářů (například kontrola validity).

Před získáním hodnot z těchto formulářů by mělo dojít ke kontrole, zda byl formulář vyplněn správně a případně upozornit uživatele na konkrétní chybu pří vyplňování formuláře.

Díky implementaci pomocí prvků z JavaFX, formuláře mohou sami automaticky reagovat na změny hodnot a okamžitě tak upozornit uživatele na nesmyslný nebo chybný vstup nebo zvýraznit, že nějaká hodnota byla změněna.

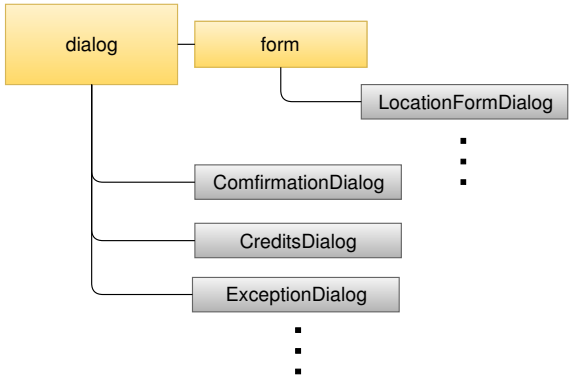
```
input.getTextProperty().addListener((observable, oldValue, newValue) -> {

    if (!input.getStyleClass().contains("changed")) {
        input.getStyleClass().add("changed");
    }
    try {
        NumberFormat.getInstance().parse(newValue);
        input.getStyleClass().remove("badInput");
    } catch (ParseException exception) {
        if (!input .getStyleClass().contains("badInput")) {
            input.getStyleClass().add("badInput");
        }
    }
//..
```

Příklad 25 Testování validace vstupu pomocí listeneru

Pro všechny entitní třídy byly vytvořeny konkrétní formuláře, které lze najít v příslušném balíčku (*cz.jcu.prf.counter.form*) a jsou poskytovány pro celou aplikaci.

V průběhu vývoje vznikla potřeba použití formuláře pro editaci vzorku se stejným layoutem, ale jiným ovládáním a datovým modelem. Jako řešení této situace byly formuláře rozděleny. Formulář nyní tvoří pouze obalovací třída (*cz.jcu.prf.counter.form.**), která pracuje nad společným layoutem (*cz.jcu.prf.counter.component.form.**).

6.5 Dialogy



Obr. 18 Rozložení tříd aplikačních dialogů

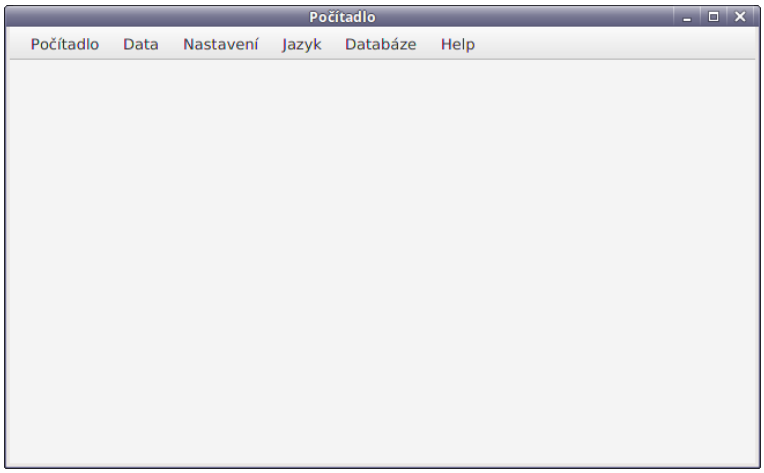
Pro snadnější použití byli předpřipraveny vyskakovací dialogy. Většinou se jedná o potomky třídy *Alert* (JavaFX 8u40), jejichž obsah byl předem připraven pro snadnější a rychlejší použití.

Speciálním případem jsou dialogy, které využívají předem vytvořené formuláře a umožňují je tak použít téměř kdekoliv v aplikaci, bez nutnosti je zakomponovávat přímo do GUI.

```
public LocationFormDialog() {
    super(Alert.AlertType.NONE, null, ButtonType.OK,
        ButtonType.CANCEL);

    //...
    this.form = new LocationForm();
    this.form.setEditForm(false);
    this.getDialogPane().setContent(this.form );

    this.setOnCloseRequest(event -> {
        ButtonType result = this.getResult();
        if (ButtonType.OK.equals(this.getResult()) && !this.form.isValid()) {
            Jukebox.BAD_BEEP.play();
            event.consume();
        }
    });
    this.setOnShown(e -> {
        Jukebox.THREE_BEEPS.play();
});
    //...
}
```

Příklad 26 Ukázka dialogu s formulářem pro vytváření lokalit

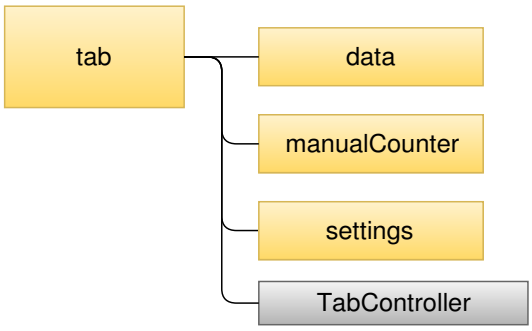
6.6 Grafické rozhraní



Obr. 19 Výsledný vzhled aplikace

Jak bylo již zmíněno, základní rozhraní (*cz.jcu.prf.counter.SceneRoot*) se skládá z menu a prostoru pro jednotlivé záložky. Tento prvek byl, stejně jako většina zbylých částí, implementován jako vlastní komponenta, a díky tomu samotná třída obsahuje pouze inicializaci a metody pro ovládání menu.

Záložky



Obr. 20 Rozložení tříd záložek

Jako základní funkční jednotka rozhraní byla vytvořena třída *TabController*, která má za účel představovat libovolnou záložku, která může být přidávána do GUI. Z důvodu, že je vyžadováno, aby se jednalo o skutečnou záložku (Tab), je *TabController* implementován, jako abstraktní třída, která dědí přímo z *javafx.scene.control.Tab*.

```
public abstract class TabController extends Tab
```

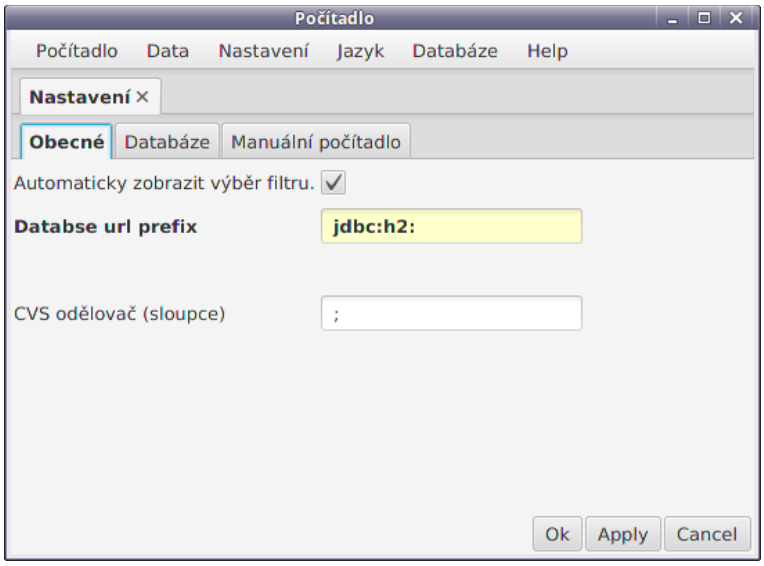
Příklad 27 Abstraktní třída TabController

Prozatím nebyl implementován žádný mechanismus, který by zautomatizoval přidání samostatných záložek do GUI, a je tedy vždy nutné rozšířit základní menu aplikace o nové tlačítko a příslušnou událost, které otevře novou záložku.

```
@FXML
private Menu optionsMenu;

@FXML
public void settingsTabAction(ActionEvent event) {
    Optional<Tab> any = this.playground.getTabs().stream()
        .filter(tab -> tab instanceof SettingsTab).findAny();
    if (any.isPresent()) {
        this.playground.getTabs()
            .getSelectionModel().select(any.get());
    } else {
        TabController settingsTab = new SettingsTab();
        this.playground.getTabs().add(settingsTab);
        this.playground.getSelectionModel().select(settingsTab);
    }
}
```

Příklad 28 Ukázka otevření nové záložky s nastavením

6.7 Nastavení



Obr. 21 Výsledný vzhled záložky nastavení

V aplikaci byla vytvořena záložka *SettingsTab*, která si nese odkazy na jednotlivé záložky a formuláře, které byli umístěny pomocí FXML.

Záložka (*SettingsTab*) pomocí listenerů kontroluje přepínání jednotlivých podzáložek a změny v jejich formulářích, kterými mění stavy (uvolnění) společných tlačítek (Ok, Apply, Cancel).

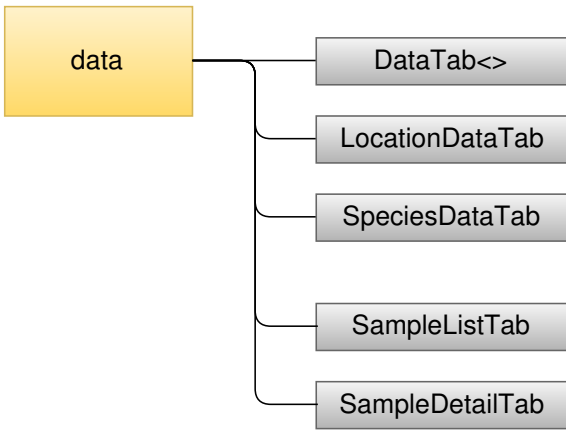
```
this.innerTabPane
        .getSelectionModel()
        .selectedItemProperty()
        .addListener((ObservableValue<? extends Tab> observable, Tab oldValue,
Tab newValue) -> {
    if ((newValue == this.generalTab && this.generalForm.isChanged())
            ||/*...*/) {
                this.applyBtn.setDisable(false);
            } else {
                this.applyBtn.setDisable(true);
            }
        });
```

Příklad 29 Kontrola uvolnění tlačítka pro potvrzení změn

```
this.generalForm.changedProperty()
        .addListener((ObservableValue<? extends Boolean> observable,
                Boolean oldValue, Boolean newValue) -> {
            if (newValue) {
                this.getStyleClass().add("changed");
            } else {
                this.getStyleClass().remove("changed");
            }
        });
```

Příklad 30 Změna CSS záložky při změně dat

6.8 Prohlížení dat

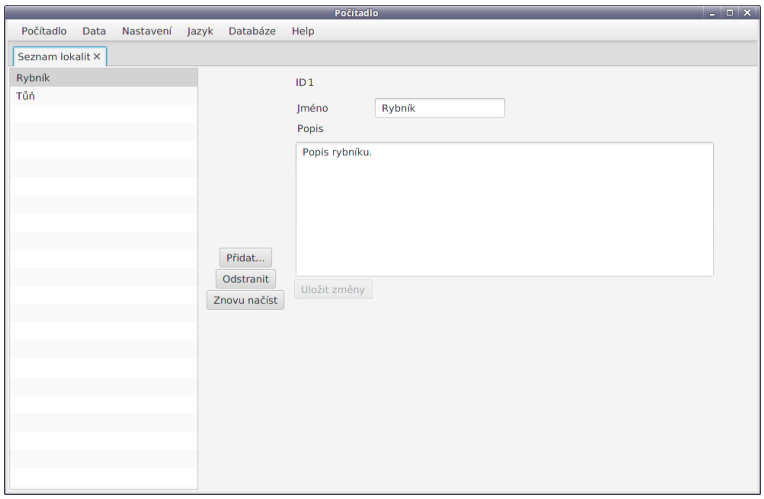


Obr. 22 Záložky pro správu dat

Tato podkapitola slučuje několik záložek, které slouží především k prohlížení dat přímo v aplikaci místo externího databázového rozhraní.

Seznam lokalit a druhů

Základem jsou záložky se seznamy druhů a lokalit, které umožnující zobrazit jejich seznam, jejich editaci a samozřejmě i jejich přidávání.

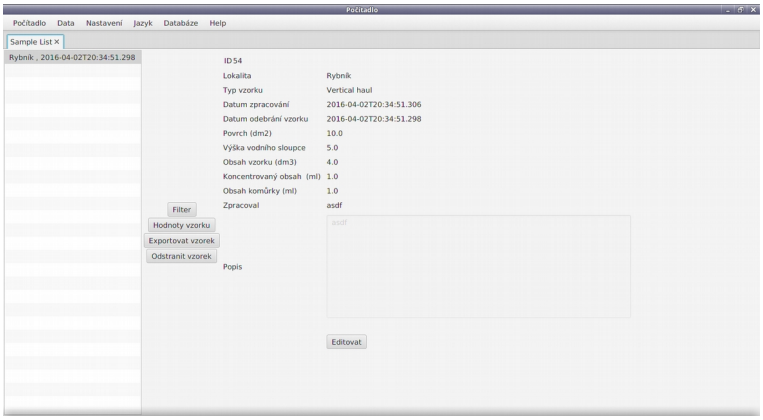


Obr. 23 Seznam lokalit

Obě záložky vycházejí ze stejné abstraktní třídy (*DataTab*), která poskytuje základní layout a implementuje některé akce. Synovské třídy *LocationDataTab* a *SpeciesDataTab* doplňují pouze formulář pro editaci, konkrétní DAO objekt a v době psaní daného textu upravují akce pro přidání a odebrání konkrétních entit.

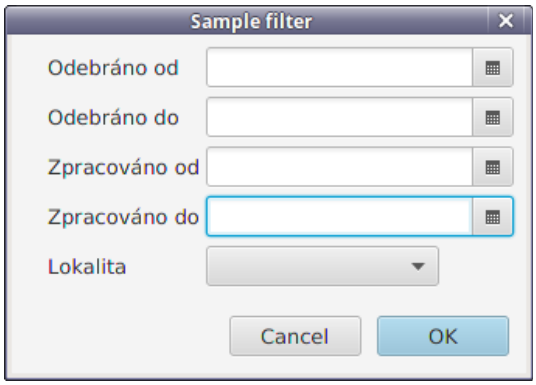
Seznam vzorků

Seznam zpracovaných vzorků, vychází z obdobného layoutu a funkcionality, který byl pouze obohacen o možnost filtrování a exportu dat do souboru. Z důvodu praktičnosti se v této záložce zobrazují pouze obecné hodnoty.



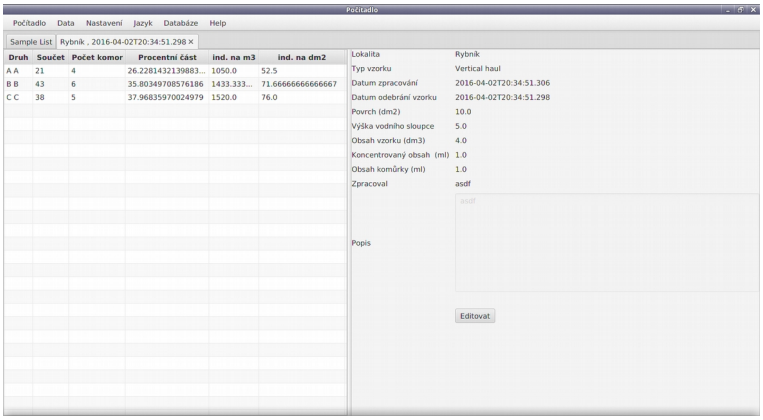
Obr. 24 Přehled vypracovaných vzorků

Pro filter byl připraven samostatný formulář, který je zobrazen pomocí vlastního dialogu (*cz.jcu.prf.counter.dialog.filter.SampleFilterDialog*). V současné době umožňuje pouze filtrování dle lokality, data odběru a zpracování. Může být ale rozšířen o jakýkoli atribut nacházející se na objektu vzorku.



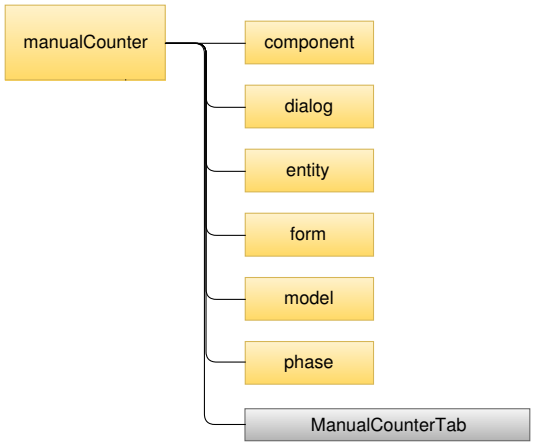
Obr. 25 Dialog pro filtrování vzorků

Samotné naměřené hodnoty a přehled druhů se zobrazují v samostatné záložce (*SampleDetailTab* viz. Obr. 26). Tato záložka vychází z návrhu posledního kroku počítadla. V levé části se zobrazují vypočítané hodnoty z poskytnutých dat.



Obr. 26 Detail konkrétního vzorku

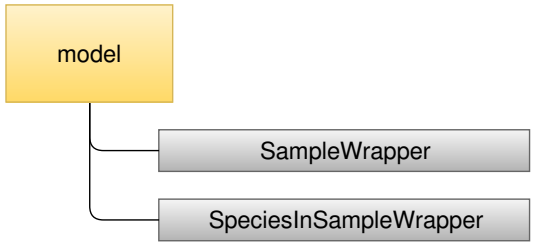
6.9 Manuální počítadlo



Obr. 27 Rozložení tříd manuálního počítadla

Jediným vstupním bodem pro manuální počítadlo je samotná třída *ManualCounterTab*, která představuje záložku počítadla. Jejím hlavním úkolem je zajistit přepínání mezi jednotlivými fázemi wizardu, udržovat modelové proměnné (zpracovávaný vzorek) a předávat je mezi jednotlivými fázemi.

6.9.1 Modelové proměnné



Obr. 28 Modelové proměnné

Pro usnadnění přenášení dat mezi jednotlivými fázemi a hlavně pro možnost editace vzorku na více místech zároveň, byly vytvořeny samostatné modelové obalové třídy, které jsou postaveny na JavaFX Properties a mohou tak být snadno sdíleny mezi různými objekty, které mohou měnit jejich hodnoty.

Samotné počítadlo navíc využívá dalších nastavení pro svou činnost (například klávesové zkratky), které byly do modelových tříd přidány (tyto parametry jsou při ukládání do databáze uloženy v parametru app_data). Použití JavaFX properties navíc umožňuje navázat listenery přímo na modelové objekty a s jejich pomocí zabránit duplicitám jako

je použití druhu nebo klávesové zkratky vícekrát, případně zajištit dalších požadované vlastností.

V průběhu vývoje bylo zváženo i využití *PropertyChangeSupport*, která umožňuje vytvořit JavaFx properties přímo jako nadstavbu nad samostatným POJO (Plain Old Java Object). *PropertyChangeSupport* vyžaduje úpravu samotné modelové třídy, kdy je v settru při nastavování nové hodnoty zavolána příslušná funkce, která zajistí update vygenerovaných properties, které byly vytvořeny nad danou proměnou.

Použití *PropertyChangeSupport* ovšem vyžaduje předem provést některé úpravy v kořenové třídě. Navíc může vyvstat problém v případě použití JPA mapování a práce s objekty ve stavu managed (změny mohou být automaticky propagovány do databáze, bez ručního volání daných metod). Z těchto důvodů nebyl tento koncept použit.

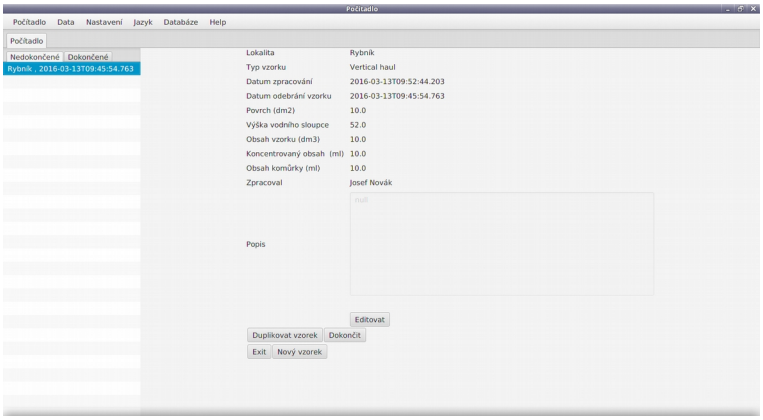
6.9.2 Průchod počítadlem

Jak bylo již zmíněno v úvodu této kapitoly, přecházení mezi jednotlivými fázemi počítadla obstarává třída *ManualCounterTab*, která zároveň představuje samotnou záložku s vyměnitelným obsahem.

```
private void reload(WizardPhases phase) {
    switch (phase) {
        case INITIALIZING:
            this.setContent(new InitPhase(this));
            //...
            break;
        case SETTING:
            this.setContent(new SettingPhase(this,
                this.sampleWrapper));
            //...
            break;
        //...
}
```

Příklad 31 Přepínání mezi jednotlivými fázemi

Inicializace



Obr. 29 Inicializace

V levé části můžeme nalézt seznamy vzorků (*ListView*), které lze přepínat pomocí příslušných záložek. Oba seznamy jsou naplněny pomocí příslušného DAO objektu po načtení záložky počítadla (viz. Příklad 32).

```
new Thread(() -> {
    try {
        HashMap<String, Object> searchCriteria = new HashMap<>();
        searchCriteria.put("finished", true);
        ObservableList<Sample> samples = FXCollections
            .observableArrayList(
                this.sampleDao.findAll(searchCriteria)
            );
        Platform.runLater(() -> {
            this.finishedSamples.setItems(samples);
        });
    } catch (Exception e) {
        LOGGER.warn("Faild to load unfinished samples from database.", e);
        Platform.runLater(() -> {
            new ExceptionDialog(Alert.AlertType.WARNING, e, //...
                ).show();
        });
    }
}).start();
```

Příklad 32 Načtení hotových vzorků

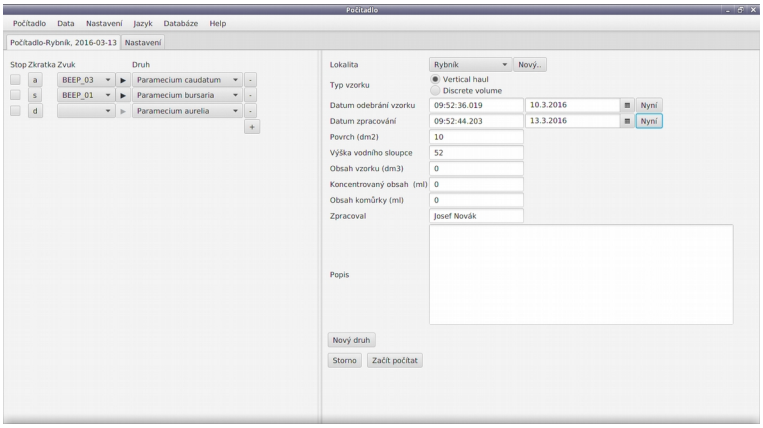
Po vybraní vzorku ze seznamu, je vzorek automaticky zobrazen v pravé části záložky a jsou zpřístupněny povolené akce (pokračování ve zpracovaní, duplikování vzorku, atd.).

```
listView.getSelectionModel()
        .selectedItemProperty()
        .addListener((observable, oldValue, newValue) -> {
            if (newValue != null) {
                this.sample = newValue;
                this.sampleView.setSample(sample);
                this.btnFinish.setDisable(this.sample.isFinished());
                this.btnFork.setDisable(false);
                this.sampleView.setEditable(!this.sample.isFinished());
            }
            this.sampleView.setDisable(newValue == null);
        });
```

Příklad 33 Zobrazení vybraného vzorku

Toto je jediný krok, který nevyužívá obalovacích modelových tříd. Místo toho používá pouze původní entitní třídy a prvky pro ně připravené z hlavní části aplikace.

Nastavení měřeného vzorku



Obr. 30 Nastavení měřeného vzorku

V levé části (Obr. 30) můžeme najít *GridPane*, ve kterém jsou zobrazeny ovládací prvky, jejichž pomocí lze nastavit měřené druhy. Ovládací prvky pro ovládání jsou udržovány pomocí instancí pomocné třídy *SpeciesInSampleWrapperOptionsCrate*, která implementuje návrhový vzor Crate (Messenger). Přidávání či odebírání těchto přepravek je zajištěno pomocí listeneru na seznamu (*ObservableList*) pozorovaných druhů.

```
this.speciesInSampleList.addListener((/*...*/ c) -> {
        while (c.next()) {
            if (c.wasAdded()) {
                SpeciesInSampleWrapper speciesInSampleWrapper =
                    c.getAddedSubList().get(0);
                this.crates.add(new SpeciesWrapperOptionsCrate(
                    this.speciesDao.findAll(),
                     this.crates, foo
                ));
            } else if (c.wasRemoved()) {
                SpeciesInSampleWrapper speciesInSampleWrapper =
                    c.getRemoved().get(0);
                this.crates.removeIf(p -> p.getSpeciesInSampleWrapper() ==
                    speciesInSampleWrapper);
            }
        }
    });
```

Příklad 34 Přidání ovládacích prvků pro druh

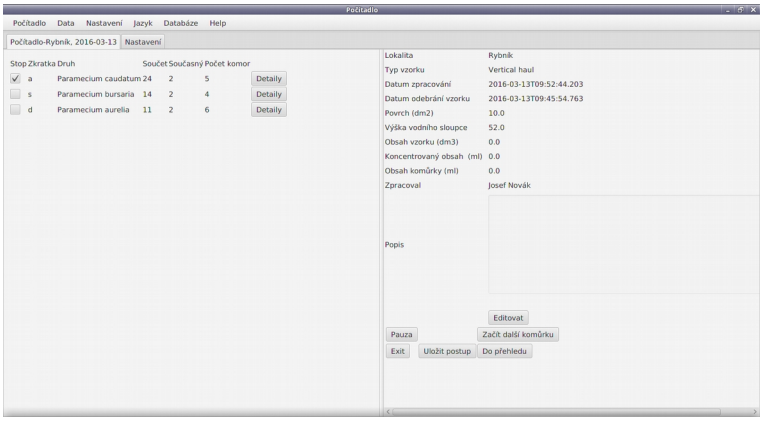
V pravé části se nalézá upravený formulář, jehož pole jsou přímo navázána na jednotlivé properties v modelové třídě, a veškeré změny jsou okamžitě přeneseny do modelu.

Přejít na další krok lze pouze, je-li formulář korektně vyplněn, a veškeré pozorované druhy jsou správně nastaveny (viz. Příklad 35).

```
@FXML
private void nextPhase(ActionEvent event) {
    if (!this.sampleForm.isValid()) {
        Jukebox.BAD_BEEP.play();
        return;
    }

    Optional<SpeciesWrapperOptionsCrate> withoutSpecies
            = this.crates.parallelStream().filter(c -> {
                return c.getSpeciesInSampleWrapper()
                    .speciesProperty().isNull().get();
            }).findFirst();

    if (withoutSpecies.isPresent()) {
        withoutSpecies.get().getSpeciesSelectBox().requestFocus();
        Jukebox.BAD_BEEP.play();
        return;
    }

    //...
```

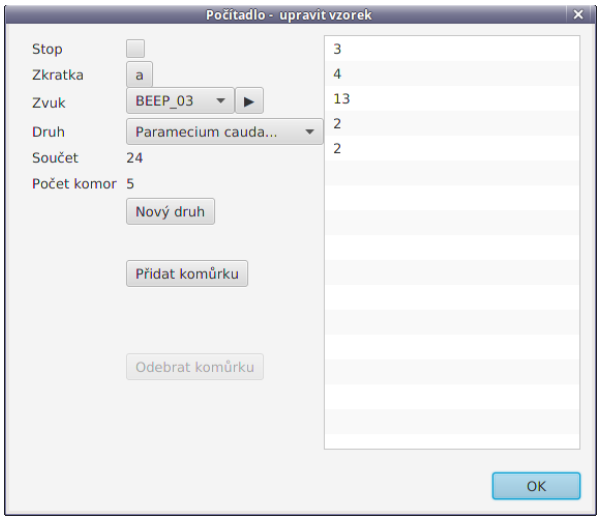
Příklad 35 Přechod na další krok

Počítání



Obr. 31 Výsledný vzhled třetího kroku počítadla

V tomto kroku jsou ovládací prvky druhů (*SpeciesInSampleWrapperOptionsCrate*) vyměněny za přepravky, která udržují obdobným způsobem zobrazovací prvky, ovládání pauzy a tlačítko pro další detaily (*SpeciesInSampleWrapperCrate*).



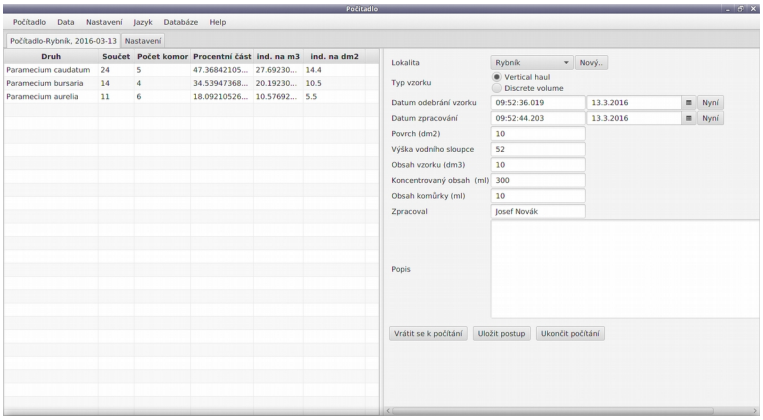
Obr. 32 Editace napočítaných hodnot

V rámci průběhu laboratorní práce je nutné udržovat přehled o počtu jedinců za více komůrek a další potřebná nastavení. Všechny tyto hodnoty musí být upravitelné. Pro tyto účely byl vytvořen vlastní dialog (*SpeciesInSampleWrapperFormDialog viz. Obr. 32*), který dokáže tyto hodnoty kdykoliv v průběhu upravit.

Hodnoty v tomto dialogu jsou provázány s modelem pomocí properties a veškeré změny jsou tedy okamžitě propagovány dále.

Tento dialog může být otevřen v posledních dvou krocích pro každý druh zvlášť. V průběhu počítání může být otevřen právě pomocí tlačítka pro detaily u jednotlivých druhů (Obr. 31 vlevo).

Přehled



Obr. 33 Poslední fáze počítadla

V posledním kroku se v levé části nalézá tabulka (*TableView*) s předvypočítanými výsledky. Jednotlivé sloupce jsou navázány přímo na modelové objekty druhů.

```
//...
this.chambersColumn.setCellValueFactory(new PropertyValueFactory("chambers"));

this.percentageColumn.setCellValueFactory(
    (TableColumn.CellDataFeatures<SpeciesInSampleWrapper, Double> param)
        -> new SimpleObjectProperty<>(param.getValue().getPercentage())
);
//...
```

Příklad 36 Navázání hodnot do tabulky

Při změně hodnot pomocí formuláře vzorku nebo při editaci pomocí dialogu jsou vypočítané hodnoty v tabulce obnoveny automaticky pomocí metody *refresh()* (JavaFX 8u60).

7 Testování

Aplikace byla testována autorem v průběhu celého vývoje. Zároveň byla v pravidelných intervalech předávána Mgr. Michalu Šorfovi Ph.D k pravidelnému uživatelskému testování. Na základě jeho upomínek a konzultací byl vytvořen kompletní návrh a implementace aplikace.

8 Návrhy pro budoucí řešení

Jako autor doufám, že díky výběrů použitých technologií bude umožněno dlouhodobé používání a snadné doplnění další funkcionality.

Zobrazení a statistika

V současné době aplikace umožňuje zobrazení a export pouze po jednom vzorku obdobně jako v původní aplikaci. Tímto způsobem se nezměnily zavedené postupy laboratorní činnosti KBE a je tedy nutné provádět zpracování pomocí dalších nástrojů.

Aplikace může být rozšířena o širší škálu možností pro samotný export naměřených dat (časosběrné záznamy, export více vzorků zároveň).

Další možností je integrovat statistické/analytické nástroje přímo do aplikace.

Databázový server

Díky použití JPA je teoreticky použití vzdáleného databázového serveru pouze otázkou vhodného nastavení. Vytvoření a správa databázového serveru je tedy spíše otázka pro Katedru biologie ekosystémů Přírodovědecké fakulty Jihočeské univerzity, ale v době psaní teoretické části nebyla aplikace pro toto použití testována.

V případě zavedení dedikovaného databázového serveru odpadají některé výhody použít H2 (možnost pracovat s databází v souborovém systému), a tedy existuje reálná možnost, že by mohlo dojít k výměně druhu použité databáze, což díky použít JPA je téměř pouze otázka konfigurace.

Další typy počítadel

V rámci této bakalářské práce byla implementována pouze alternativa k původnímu manuálnímu zpracování v původní aplikaci. Na tomto poli si jistě můžeme představit další typy rozhraní pro analýzu, od manuálního zpracování fotek vzorku, až po plně automatické zpracování, které by výrazně zjednodušilo samotnou laboratorní činnost.

Záložky

Samotné záložky rozšiřují univerzální třídu, ale nebyl implantován, žádný mechanizmus, který by zautomatizoval proces přidání (registrace) těchto prvků do GUI. V současné době přidání další záložky znamená manuální přidání dalšího prvku do hlavního menu a příslušné události.

Nastavení aplikace

Nastavení aplikace by mohlo být zpřehledněno a rozděleno na oddělené části, které by lépe odpovídaly struktuře aplikace (oddělená konfigurace záložek).

Zároveň by měla být přidána i další nastavení, která byla navržena již při samotném vývoji, například jako je použitý formát a zobrazení datumu.

9 Závěr

Hlavním cílem této bakalářské práce bylo vytvořit novou aplikaci pro Katedru biologie ekosystémů Přírodovědecké fakulty Jihočeské univerzity. Tato aplikace byla úspěšně vytvořena pro platformu Java (JavaFX), která díky své nezávislosti na operačním systému zajišťuje maximální přenositelnost.

Výsledná aplikace umožňuje určovat počty jedinců ve studovaných vzorcích obdobným způsobem jako původní aplikace. Veškeré zpracovávané údaje jsou ukládány do souborové databáze H2. Databáze H2, kromě ukládání dat přímo v souborovém systému, umožňuje pracovat proti vlastnímu dedikovanému serveru, který může být zřízen v případě potřeby pro centralizovanou zprávu dat. Použitím JPA není aplikace ani přímo vázána na tento konkrétní typ databáze.

Díky použití Javy jako platformy a samotnému rozdělení hlavních částí aplikace na jednotlivé záložky, je zajištěn prostor pro další snadné rozšiřování o novou funkcionalitu.

Aplikace bude již brzy aktivně nasazena po opravě posledních nedostatků, které ale budou dořešeny až po odevzdání textové části.

Před odevzdáním bakalářské práce vyvstala otázka, jakým způsobem spravovat další vývoj vytvořené aplikace. Jednou z možností je uveřejnění přímo na serveru GitHub. Vyskytla se však otázka, zda by neměl být zřízen přímo účet univerzity. Tento účet by mohl být dále využíván i pro další studijní účely a projekty. Další možností by byl vlastní server spravovaný přímo univerzitou.

Tato otázka bohužel zůstává, v době odevzdání této textové práce, nedořešena. Z tohoto důvodu nebyl zatím zveřejněn zdrojový kód aplikace ve veřejně přístupném repozitáři. V případě nevyřešení této otázky bude aplikace umístěna na serveru GitHub přímo pod účtem autora. Zároveň bude vytvořen subprojekt jehož účelem bude vytvoření a údržba uživatelské příručky pro vytvořenou aplikaci.

10 Seznam použité literatury

[1] Maven's Objectives. Apache Maven Project [online]. 2016 [cit. 2016-04-10]. Dostupné z: https://maven.apache.org/what-is-maven.html

[2] Introduction to the POM. Apache Maven Project [online]. 2016 [cit. 2016-04-10]. Dostupné z: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

[3] CHACON, Scott a Ben STRAUB. *Pro Git* [online]. 2nd Edition. CA: Appress, 2014 [cit. 2016-04-10]. ISBN 978-1-484200-77-3. Dostupné z: https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf

[4] HORSTMANN, Cay S a Gary CORNELL. *Core Java*. 8th ed. Upper Saddle River, NJ: Prentice Hall/Sun Microsystems Press, c2008. ISBN 0132354799.

[5] VOS, Johan. *Pro JavaFX 8: a definitive guide to building desktop, mobile, and embedded Java clients / Johan Vos, Weiqi Gao, Stephen Chin, Dean Iverson, James Weaver*. Berkeley, CA: Apress, 2014. Expert's voice in Java. ISBN 1430265744.

[6] Simple Logging Facade for Java. SLF4J [online]. 2016 [cit. 2016-04-18]. Dostupné z: http://www.slf4j.org/

[7] Logback [online]. 2016 [cit. 2016-04-18]. Dostupné z: http://logback.qos.ch/

[8] H2 Database Engine. H2database [online]. 2014 [cit. 2015-04-17]. Dostupné z: http://www.h2database.com/html/main.html

[9] PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Vyd. 1. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[10] Oracle® Containers for J2EE Enterprise JavaBeans Developer's Guide: Understanding Enterprise JavaBeans. *Oracle Help Center* [online]. 2009 [cit. 2016-04-11]. Dostupné z: http://docs.oracle.com/cd/E16439_01/doc.1013/e13981/toc.htm

[11] Writing JavaBeans Components: Properties. *Oracle Help Center: The Java™ Tutorials* [online]. 2015 [cit. 2016-04-12]. Dostupné z: http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html

11 Přílohy

Seznam obrazových příloh

Seznam příkladů zdrojového kódu

Disk (CD)

Obsah disku:

- Text bakalářské práce (PDF)
- Build aplikace (JAR)
- Zdrojové kódy aplikace (ZIP)
- kopie Git repositáře (ZIP)
- Manuál k původní aplikaci (PDF)
- Původní aplikace (ZIP)
- README.txt (popis struktury disku)