Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Bakalářská práce

# Nástroj pro ruční vytváření komplexních vstupních dat pro testování

Plzeň 2019 Michal Linha

# Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 26.června 2019

Michal Linha

# Abstract

This bachelor thesis' main goal is to create a *Java* library that will be able to create complex input data used for testing. Some existing approaches for random testing data generation and the functionality of Java reflection were described in the first part of this thesis. Second part contains proposion and implementation of the library itself. The library was called *TestingTool*. It is able to create instances of complex data types (if they contain at least one public constructor) and set their attributes using graphical user interface. Fields, `ArrayList` and `HashMap` were types that recieved special support. It is also possible to create multiple instances at once and generate random values for their primitive number attributes.

# Abstrakt

Tato práce se zabývá vytvořením knihovny v *Javě*, která bude vytvářet komplexní vstupní data pro testování. V první části byly popsány některé z existujících postupů generování náhodných dat pro testování a také funkčnost Java reflexe. Ve druhé části je popsán samotný návrh a implementace knihovny *TestingTool*. Knihovna dokáže vytvářet instance komplexních datových typů (v případě, že existuje alespoň jeden veřejný konstruktor) a nastavovat jim atributy pomocí grafického uživatelského rozhraní. Speciální podporu pak obdržely datové typy `ArrayList`, `HashMap` a pole. Také je možné vytvářet více instancí najednou a generovat jim náhodné hodnoty primitivních číselných atributů.

# Obsah

# 1 Úvod

Testování aplikací je nedílnou součástí při vytváření aplikací a jedná se o drahou a důležitou část [1, 2], protože vyžaduje velké množství prostředků. Dělí se na 3 základní typy a to black box testování, white box testování a grey box testování [3].

Při black box testování vůbec neznáme implementaci aplikace, při white box testování ji známe a při grey box testování víme o jejích vnitřních postupech, ale neznáme dostatek informací na to, aby se testování dalo považovat za white box testování [4, 5, 6].

Cílem práce je vytvoření knihovny, jež bude umožňovat vytváření komplexních testovacích dat (objektů s netriviální vnitřní strukturou) ručně, za pomoci grafického uživatelského rozhraní, bez nutnosti dostupnosti zdrojového kódu. Tuto knihovnu bude poté možné využít při white box testování, konkrétně v jednotkových testech.

V teoretické části této práce je popsáno několik postupů při získávání dat pro testování aplikací. Všechny uvedené postupy jsou použity pro automatické generování testovacích dat. V práci jsou popsány dva postupy s použitím pairwise testování, jeden postup založený na genetickém programování, jeden postup založený na optimalizaci hejnem částic a jeden založený na použití meta-dat získaných reflexí.

V praktické části je pak popsán návrh a následná implementace vytvořené knihovny pro ruční vytváření komlexních objektů a nastavování jejich atributů. Dále jsou diskutována její omezení a možná rozšíření.

# 2 Testovací postupy

Jak již bylo zmíněno, testovaní aplikací se dělí do tří základních skupin podle znalosti kódu a to do black box testování, white box testování a grey box testování. Dále je možné je rozdělit podle fáze testování, nebo podle způsobu testování [3].

## 2.1 Black box testování

Při black box testování tester nezná vnitřní implementaci testovaného programu. To znamená, že nemůže testovat celou implementaci programu, ale jen tu část, ke které má přístup, což může být například grafické uživatelské rozhraní. Díky tomu může například otestovat, zda program pracuje správně při zadaných vstupních datech, a tím lze zjistit, zda-li náhodou nedochází k pádům aplikace při určité kombinaci vstupních dat. Také lze otestovat, jestli při zadání mezních hodnot nedochází k neočekávaným chybám, nebo jestli se aplikace po nějaké interakci s uživatelem nachází v očekávaném stavu [4].

Jednou z možností black box testování je rozdělení vstupních dat do skupin, ve kterých jsou si hodnoty podobné, nebo mají nějakou společnou vlastnost. Dojde-li k chybě pro hodnotu z jedné ze skupin, budou všechny hodnoty z též skupiny brány jako hodnoty, které způsobí chybu programu [4].

## 2.2 White box testování

White box testování je druh testování, u kterého tester zná vnitřní implementaci programu a může tedy otestovat jednotlivé komponenty zvlášť. Díky tomu je možné otestovat průběh dat celým programem a tudíž i zlepšit design aplikace, sílu zabezpečení a další faktory [5].

Tester se při testování snaží získat co nejvyšší pokrytí kódu testovacími případy a tím zajistit důkladné otestování všech stavů, do kterých se aplikace může dostat. Po spuštění testů pak zjistí, kterou část kódu může změnami urychlit, v které části jsou bezpečnostní díry nebo dochází ke ztrátě dat, či zda všechny cykly a podmínky provádí správnou činnost. Mezi tyto testy patří například jednotkové testy viz kapitola 2.4 [5].

## 2.3 Grey box testování

Grey box testovaní je testovaní založené na spojení white box a black box testování.Tester tedy zná implementaci testované aplikace, ale pouze částečně. Může tedy otestovat jak vnější část aplikace (např. grafické uživatelské rozhraní), tak vnitřní část aplikace, tedy tu, ke které má přístup. Toho lze využít při penetračních nebo integračních testech [6].

Grey box testovaní umožňuje vytvářet vstupní data jak testerům, tak programátorům a tím zlepšuje kvalitu a sílu testů a tedy i celého programu. Někdy také zkracuje dobu celého testování a dává tak programátorům mnohem více času na opravy chyb [6].

## 2.4 Unit testy

Unit testy, nebo-li jednotkové testy, slouží pro prvotní testování programu a obvykle jsou prováděny programátorem. Díky nim dokáže programátor nalézt chyby ve funkcionalitách tříd, například v případě Javy, a jejich metodách. Programátor tedy vytváří v průběhu své práce testy, ideálně pro každou metodu, kterou napíše. V testech zkoumá, zda metoda provádí očekávanou činnost.

Testovat může například, zda metoda vrací správnou (očekávanou) hodnotu, nebo jestli správně nastaví hodnotu atributu. V případě cyklů může testovat, zda jsou cykly konečné. Dále testuje vstupní parametry metody, zda-li s nimi při jejich validním zadání provede správnou činnost. Také lze takto testovat nevalidní vstupní parametry, nebo atributy třídy či instance a reaguje-li metoda správně například na `null` hodnoty. Při testování Java aplikací je možné pro tyto testy použít například JUnit framework nebo TestNG. Pro C aplikace je to například AceUnit [7].

Podobné unit testům jsou pak modul testy, které se zaměřují na funkcionalitu celých modulů aplikace, jako jsou například knihovny [3].

## 2.5 Integrační testy

Integrační testy už vytváří testeři, jejichž úkolem je otestovat, zda aplikace pracuje správně po připojení všech modulů, ze kterých se skládá, nebo po připojení jednoho nového modulu do již fungující aplikace. Jelikož mohou být moduly aplikace vytvářeny různými programátory, může i po jejich řádném otestování jednotkovými testy docházet k chybám po spojení modulů. Taktéž může nastat případ neotestovaných změn v jednom z modulů a tím

pádem i k výskytu chyb v celém systému. Při integračním testováním obvykle dochází k testování rozhraní mezi jednotlivými moduly a k testování správného pohybu dat mezi nimi. Existuje několik způsobů, jak toto testování provést, například inkrementální způsob nebo způsob velkého třesku [8].

Způsob velkého třesku funguje tak, že se čeká na vytvoření všech modulů aplikace, které se následně složí a společně otestují. To může snižovat efektivitu, protože vytváření některých modulů může trvat velmi dlouhou dobu, a tím pádem musí ostatní moduly čekat na otestování. Také je při použití tohoto způsobu těžší najít chybu, protože se testuje velké množství rozhraní, což může vést i k zapomenutí některého z nich. Nedochází ani k oddělenému testování modulů, u kterých je důležitá vysoká spolehlivost. To může mít za následek, že se u nich při testování přehlédne chyba, která může mít vážné důsledky. Tento způsob integračního testování je tedy vhodný jen pro malé programy [8].

Další možností je použití inkrementálního způsobu. Zde dochází k připojení modulů, které spolu nějak souvisí, a k jejich samostatnému otestování. Pokud testy proběhnou v pořádku, jsou k původním modulům připojeny další, již otestované moduly, a tento celek je opět otestován. Jsou k tomu také použity moduly, které jen simulují funkcionalitu jiného modulu a umožňují tak urychlení testovaní už vytvořených modulů [8].

Tento způsob se dělí na dva typy. Jsou jimi shora dolu a zespoda nahoru. Výhodou typu shora dolu je urychlení hledání problému, možnost vytvoření prototypu systému a také to, že nejdříve dochází k otestování hlavních modulů. Nevýhodou je pak možnost špatného otestování vedlejších modulů. Výhodou typu zespoda nahoru je opět urychlení hledání chyby. Nevýhodou je nemožnost vytvoření prototypu a testování hlavních komponent až na konci testovacího cyklu. Existuje také hybridní typ, který spojuje oba zmíněné typy [8].

## 2.6 Funkční testy

Funkční testy jsou použity pro otestování správné funkcionality celé aplikace podle zadané specifikace. Obvykle jsou prováděny za použití black box testování. Cílem je otestovat, zda každá funkcionalita aplikace provádí s daty očekávané činnosti a vrací očekávané hodnoty. Testují se tedy správně zadané hodnoty, hraniční hodnoty a špatně zadané hodnoty do jednotlivých funkcionalit aplikace. Mezi hlavní testovací nástroje patří Selenium, QTP nebo JUnit [9].

Jedním z typů funkčních testů jsou regresní testy. Regresní testy se provádí po provedení změn či oprav v programu. Protože při údržbě programu dochází ke změnám, jako například opravám, vylepšením, přidáním funkcionalit nebo jejich smazáním, je velká pravděpodobnost, že jedna z výše uvedených změn způsobí neočekávané chování programu. Při regresním testování se tedy provede spuštění již provedených testů funkcionality v předchozí verzi na verzi změněné. Provádění všech testů ale může být velmi časově náročné a nákladné. Proto se někdy provádí výběr jen některých testů, například těch, které dříve vykazovaly velkou chybovost, nebo jen testů určených pro základní otestování [10].

## 2.7 Další testy podle fáze testování

Dalším typem testů jsou systémové testy, které testují celý systém jako celek a zjišťují, zda-li celý systém pracuje podle zadané specifikace. Posledním typem testů jsou testy akceptační, které provádí zákazník a testuje, jestli aplikace pracuje podle jeho představ [3].

## 2.8 Manuální a automatizované testování

Manuální testování je prováděno testery, kteří testy vytváří a následně spouštějí. Pro každou aplikaci musí být nejdříve vytvořeny manuální testy, až poté se testy dají zautomatizovat. Tento způsob testování je velmi důležitý, ale v určitých případech i mnohem dražší než test automatizovaný. Na rozdíl od automatizovaného je ale použitelný u jakýchkoliv aplikací [11].

Automatizované testování vychází z manuálního testování a hodí se nejvíce u stabilních aplikací pro regresní testování. Vytvoření automatizovaného testu je dražší, než vytvoření testu manuálního. Ve většině případech se ale tato investice v čase vrátí, protože automatizovaný test poté může testy provádět 24 hodin denně, zatímco manuální test provádí pracovník ve své pracovní době. U automatizovaných testů ovšem ale záleží na jejich kvalitě, jelikož i přes rychlý průběh nekvalitně napsaného testu, který je rychlejší než manuální, nemusí být chyba nalezena [12].

Existuje také exploratory testování, což je vlastně testování založené na testování aplikace bez daného scénáře. Jejich účelem je nalezení chyb na základě zjišťování funkcionalit aplikace během testování. Tester tedy napíše test a hned ho spustí. Na rozdíl od manuálního testování je tak od testera vyžadováno mnohem většího myšlení při psaní testů [13].

# 3 Generování testovacích dat

## 3.1 Pairwise testování

Pairwise testování je druh black box testování. Cílem je, aby testovací balíček obsahoval všechny možné diskrétní kombinace vstupních parametrů programu a co nejmenší počet testovacích případů. Testovací případy se vytváří tak, že se pro každý pár vstupních parametrů vytvoří všechny možné kombinace jejich hodnot a tyto páry se poté spojí dohromady takovým způsobem, aby každý testovací případ obsahoval jiné kombinace jednotlivých párů než ostatní testovací případy. Bohužel, ne v každé testované aplikaci se povede dosáhnout úplné unikátnosti, ale počet testovacích případů se díky tomuto postupu výrazně zmenší (oproti vytváření všech možných kombinací klasickým kombinatorickým postupem). Výsledky pokusů ukazují, že pairwise testovaní dokáže objevit *50 – 97%* chyb v programu [14, 15].

## 3.2 Efektivní strategie pro generování testovacích dat s pomocí pairwise testování

Jedním z postupů pro generování testovacích dat pomocí pairwise testování je algoritmus pojmenovaný IRPS [15]. Jeho postup bude popsán podrobněji.

### 3.2.1 Popis algoritmu

Před zahájením hledání testovacích případů se nejdříve vygenerují všechny možné páry parametrů a jejich hodnot a ty se uloží do kompaktního spojového seznamu. Kompaktní spojový seznam obsahuje *N - 1* spojových seznamů, kde *N* je počet parametrů programu. Každý spojový seznam obsahuje *M* uzlů označující počet možných hodnot daného parametru. V každém z těchto uzlů je pole spojových seznamů, které reprezentuje pár s každým dalším uzlem z následujících spojových seznamů. Jednou z výhod takového uložení dat je lepší využití paměti. Tento kompaktní spojový seznam tedy zabere mnohem méně místa [15].

V postupu se používá několik výrazů. Váha kandidátního testovacího případu je počet párů, které jsou pokryty tímto testovacím případem. Dále pak maximální váha, která se vypočte jako

$$w_{max} = \frac{N \cdot (N-1)}{2}, \tag{3.1}$$

kde $N$ je počet parametrů. Proměnná *miss* je rozdíl mezi maximální váhou a váhou kandidátního testovacího případu. Další je průnik uzlu v seznamu *i* se seznamem *i+1*, což je průnik mezi uzlem a všemi uzly v *i+1* seznamu. Nakonec je operace odstranění, která odstraní každou proměnnou v každém uzlu [15].

Po vygenerování kompaktního spojového seznamu se spustí proces hledání testovacích případů. Nejdříve je vytvořen dvojitý spojový seznam, který obsahuje *i* uzel a průnik s druhým uzlem z *i+1* seznamu a s dalšími uzly. Pokud je první seznam pole párů prázdný, tak se průnik provede se všemi uzly z dalšího seznamu a hodnota proměnné *miss* bude (pokud bude platit, že *miss* je větší než *0*) snížena o *1*. Jinak bude proces průniku přerušen a běh se posune na další uzel. Kandidátní testovací případ se získá načtením hodnoty uzlu v každém uzlu dvojitého spojového seznamu. Pro poslední uzel vezme kandidátní testovací případ aktuální hodnotu a první prvek seznamu párů. Poté se zjistí, jestli kandidátní testovací případ splňuje zadané kritérium své váhy. V případě, že kritérium nesplňuje, proces odstraní poslední uzel ve dvojitém spojovém seznamu a nahradí ho průnikem s dalším uzlem v seznamu, nebo pokud není v seznamu žádný uzel, odstraní poslední dva uzly a pokračuje v iteraci. Iterace se zastaví, když je kompaktní list prázdný [15].

### 3.2.2 Výsledky

Výsledné testování ukázalo, že ve většině případů vygeneroval výše popsaný algoritmus menší testovací případy než jiné algoritmy založené na pairwise testování. Doba běhu není podle autorů textu změřena správně, ti ale tvrdí, že je akceptovatelná [15].

## 3.3 Generování testovacích dat na základě paralelních stromů

Další algoritmus používá kombinaci pairwise testování, paralelních algoritmů a strategie rozdělování pro vygenerování testovacích případů a následně pro vytvoření testovacích balíčků s nejmenším počtem testovacích případů [14].

Nejznámější strategií paralelních algoritmů je *rozděl a panuj* (*divide-and-conquer*). Touto strategií se řeší například složité výpočty, které se rozdělí

na části, přičemž každá část se poté počítá zvlášť, paralelně s dalšími částmi. Získané výsledky se následně složí do jednoho konečného výsledku [14].

Cílem strategie rozdělování je také rozdělení problému na části, tyto části ale musí být navzájem nezávislé. Díky tomu je možné výsledky podproblémů jednodušeji spojit [14].

### 3.3.1 Popis algoritmu

Algoritmus pro generování testovacích případů pracuje tak, že vytvoří strom, kde v jeho listech jsou uloženy všechny testovací případy, které jsou základem pro nalezení nejmenší množiny testovacích případů. Z kořenového uzlu vychází takový počet hran, jako je počet hodnot prvního parametru programu. Tyto hodnoty jsou uloženy v následujících uzlech. Tuto akci provádí hlavní vlákno. Dále je vytvořeno tolik vláken, jako je počet hodnot prvního parametru, a každému vláknu je přiřazena jedna větev. Vlákna následně procházejí větve do hloubky *N*, kde *N* je počet parametrů. Následující uzel je vždy vytvořen tak, že aktuálně procházený uzel je vždy *(M – 1)*krát replikován, kde *M* je počet hodnot *(N + 1)tého* parametru. Díky paralelismu se zredukuje doba běhu algoritmu [14].

Výše popsaná strategie minimalizuje počet uzlů tak, že během jednotlivých kroků dává důležitost jen aktuálním listům stromu. Pro každý list stromu vždy spočte, kolikrát by měl být replikován pomocí vzorce

$$r = p_j - 1, \tag{3.2}$$

kde $p_j$ je počet hodnot $j$ parametru. Do replik listů jsou pak vloženy jednotlivé hodnoty $j$ parametru a tyto repliky jsou následně uloženy do seznamu listů stromu za původní listy [14].

Po vytvoření stromu se provádí hledání nejmenší množiny testovacích případů. Nejdříve se vytvoří pole pokrytí, ve kterém jsou uloženy všechny možné páry parametrů. Dále hlavní vlákno uloží do výsledných testovacích případů všechny základní testovací případy ze seznamů větví stromu a smaže všechny pokryté páry z pole pokrytí, stejně tak smaže tyto testovací případy ze seznamů větví stromu. Váha testovacího případu se počítá jako počet párů, které jsou testovacím případem pokryty. Po dokončení této části se v hlavním vláknu vytvoří tolik vláken, kolik je počet hodnot prvního parametru. Každé vlákno obdrží jednu větev stromu, jako tomu bylo u výše popsaného postupu. V každém vláknu je uložen seznam testovacích případů s váhou $W_{max}$. Poté, co jedno z vláken dokončí výpočet vah testovacích případů své větve, uzamkne toto vlákno pole pokrytí a poté ze svého seznamu testovacích případů uloží do testovacího balíčku takové testovací

případy, které mají po ověření váhu $W_{max}$. Takto vložené testovací případy jsou poté smazány ze seznamu větví stromu a seznamu testovacích případů vlákna. Páry, které jsou těmito testovacími příklady pokryty jsou smazány z pole pokrytí [14].

Zbylá vlákna počkají ve frontě, dokud první vlákno nedokončí operaci a poté postupně, podle toho, jak byla do fronty vložena, provedou stejnou operaci jako vlákno první, se svými seznamy testovacích případů [14].

Poté, co všechna vlákna provedou své operace, je váha, tedy hodnota $W_{max}$, snížena o *1* a vlákna jsou opět paralelně spuštěna pro svojí větev stromu. Tímto by mělo být zajištěno nalezení minimální velikosti testovacího balíčku a pokrytí všech možných párů hodnot [14].

### 3.3.2 Výsledky

Z testů vyšlo najevo, že tento postup nedává vždy stejně velké výsledné testovací balíčky. Velikost výsledného testovacího balíčku závisí na době běhu jednotlivých vláken [14].

Dále bylo zjištěno, že v porovnání s dalšími strategiemi, je tento postup lepší než jiné strategie při určitých konfiguracích, ale je i horší při jiných konfiguracích [14].

## 3.4 Použití genetického algoritmu pro generování dat sledující více cílů v testování založeném na prohledávání

Cílem generování dat sledující více cílů není jen vygenerování takových testovacích dat, která budou mít vysoké pokrytí kódu, ale také například dobrá doba běhu generování, nebo dobré využití paměti zároveň. V tomto případě se autor snaží získat maximální pokrytí a minimální velikost testovacího balíčku [16].

Při testování založeném na prohledávání se nejprve problém vymodeluje jako problém optimalizace numerické funkce. Tento problém se poté řeší pomocí heuristických funkcí [16].

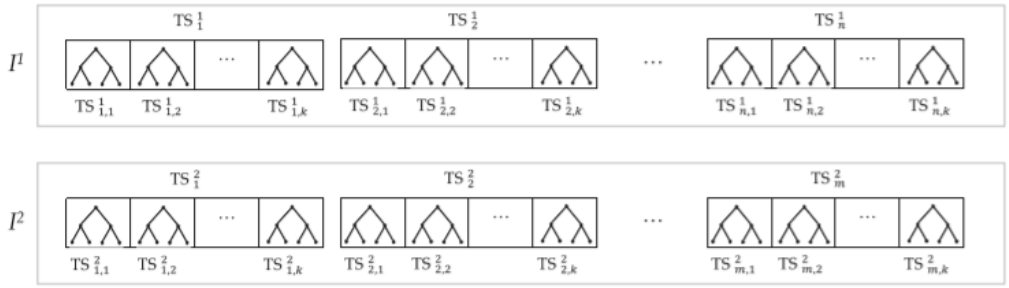
U genetického programování je využito principů podobných biologické evoluci. V podstatě jde tedy o vytvoření základní populace a její modifikaci křížením a mutací. Následuje vyloučení „slabých“ jedinců, z čehož vznikne nová populace, blížící se zadanému řešení. Tento proces se opakuje, dokud není dosažen zadaný počet generací, přičemž z poslední populace se získá přibližný výsledek, nebo dokud není nalezen přesný výsledek [17].

Prvky genetického programování jsou reprezentovány jako stromy, kde ve vnitřních uzlech stromů jsou uloženy funkce a operátory a v listech stromů jsou uložena náhodná čísla. Každý problém obsahuje populaci těchto stromů. Každý strom v populaci reprezentuje jedno možné řešení problému. [17]

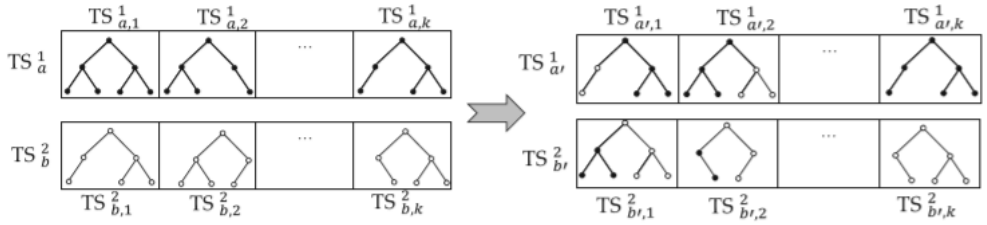
### 3.4.1 Popis algoritmu

Tento postup se zabývá vylepšením reprezentace, křížení a mutace [16].

Reprezentace dat je rozdělena do 3 vrstev. Ve třetí vrstvě je uložen celý testovací balíček, jehož velikost je proměnná z důvodu, že se při aplikaci mutace a křížení může počet testovacích případů měnit. Ve druhé vrstvě jsou uloženy testovací případy. Jedná se o seznam stromů, jehož velikost je daná počtem argumentů programu, který je testován. Nakonec první vrstva reprezentuje jednotlivé stromy pro daný argument. Uzly a listy stromů jsou nakonfigurovány podle typu argumentů [16].



Obrázek 3.1: Struktura vybraných prvků $I^1$ a $I^2$ [16].



Obrázek 3.2: Stav před a po provedení křížení pro první a druhý strom [16].

Pro křížení se nejprve vyberou 2 prvky (testovací balíčky) z populace pomocí turnajového schématu. Poté se náhodně vybere seznam stromů z obou vybraných prvků. Na tyto seznamy se pak použije operace křížení a to tak, že každý strom z prvního prvku se zkříží se stromem na stejné pozici ve druhém prvku. Při křížení se náhodně v každém ze stromů zvolí uzel a tyto

uzly vytvoří kořeny křížených podstromů, které se prohodí. Počet opakování tohoto postupu se rovná velikosti menšího z vybraných prvků. Díky tomuto postupu je možné křížit prvky s různou délkou [16].

Strukturu dvou vybraných prvků a následné provedení křížení a mutací lze vidět na obrázcích 3.1 a 3.2.

Při mutaci se mění velikost vybraných prvků – buď se do vybraného prvku přidá několik testovacích případů, nebo se jeden odebere a nebo zůstane stejný [16].

### 3.4.2 Výsledky

Výše popsaný algoritmus byl implementován v několika optimalizačních frameworcích. Oproti náhodnému hledání a dvěma postupům založeným na genetických algoritmech, měl tento postup lepší výsledky [16].

## 3.5 Automatické generování testovacích dat pro data flow testování s použitím optimalizace hejnem částic

*Data flow* testování generuje graf programu. V každém uzlu vygenerovaného grafu je uložen blok s tvrzeními a každá hrana představuje přesun kontroly programu mezi jednotlivými bloky. Pokud první uzel cesty v grafu je vstupní a poslední je výstupní, nazývá se tato cesta kompletní cesta. *Def-clear* cesta pro proměnnou je cesta neobsahující žádnou novou definici proměnné [18].

Cílem data flow testování je zjistit všechny možné definice proměnné a všechna její použití v programu. Autoři tohoto postupu se rozhodli, že hlavním kritériem je nalezení všech použití proměnné. Pro toto je nutné projít celou def-clear cestu od každé definice proměnné do každého jejího použití [19].

*Optimalizace hejnem částic*, *Particle Swarm Optimization* (PSO), je metaheuristická metoda používaná v oblasti umělé inteligence. Je založena na pohybu hejna ptáků v reálném světě, kde každý pták (částice) má svojí rychlost, polohu a paměť předchozích úspěchů při hledání. Částice, které mají větší úspěchy při hledání, ovlivňují ostatní částice. V každém kroku algoritmu jsou upraveny hodnoty částic a hejno se tak pohybuje k cíli, tedy k řešení [20].

### 3.5.1 Popis algoritmu

Na začátku generování dat pomocí PSO je nutné vygenerovat počáteční generaci částic a jejich rychlosti. Poté se spočte ohodnocující funkce pro každou částici a pokud výsledky nesplňují zadaná omezení, je nutné celý proces opakovat. Pokud výsledky omezení splňují, nastaví se jednotlivé částice jako nejlepší částice aktuální iterace a částici s nejlepším výsledkem uloží jako nejlepší globální částici [19].

Následně se zvýší čítač iterací o *1* a podle stanovených vzorců se upraví pozice a rychlosti částic. Dále se opět spočte ohodnocující funkce pro každou částici a provede se úprava nejlepších pozic a to tak, že pokud je nová hodnota $i$ částice větší než předchozí, je $i$ nejlepší částice aktuální iterace nastavena na $i$ částici, jinak je nastavena na $i$ nejlepší částici z předchozí iterace. Podobně je upravena nejlepší globální částice. Celý postup se opakuje, dokud není dosaženo *100%* pokrytí, nebo stanoveného maximálního počtu iterací [19].

### 3.5.2 Výsledky

Algoritmus byl porovnán s častěji používaným genetickým algoritmem ve 14 různých programech a ve všech jej překonal [19].

PSO byl použit v implementacích jiných autorů, kteří používali jiné postupy a jiné ohodnocující funkce nebo funkce pro změny rychlostí a pozic částic, s těmi ale není porovnáván v programech [19].

## 3.6 Komponentově založené black box testování s použitím meta-dat

Vytvářené aplikace se skládají z různých komponent. Tyto komponenty mohou být dodávány třetími stranami a jejich uživatel ani nemusí znát jejich vnitřní implementaci. Po jejich vložení do aplikace je ale nutné provést testování a to může být obtížný úkol [21].

Meta-data jsou data, která jsou přikládána například ke komponentě aplikace jejím autorem, nebo dalšími uživateli. Jsou to vlastně data o datech. Následující text využívá meta-dat použitím reflexe [21].

Framework *.NET* má od svých autorů implementovány knihovny, které lze využít pro zjištění struktury přeložené aplikace, jelikož u něj nedochází k překladu kódu rovnou do strojového kódu, ale do byte kódu, který lze přečíst a jednodušeji přeložit zpět. Knihovna, která tohoto využívá se jmenuje *.NET* reflexe. Pomocí reflexe je tak možné získat informace o třídách

komponent, metodách či funkcích a dalších typech proměnných. Metody je také možné zavolat [21].

### 3.6.1 Popis algoritmu

Jelikož díky reflexi získáme veškeré informace o komponentě, je možné vytvářet testovací uživatelská rozhraní, přes které poté tester může vložit hodnoty atributů a spustit testování. Pro vytvoření takového uživatelského rozhraní mu jen stačí vložit `.dll` soubor do navržené aplikace, která v něm pomocí reflexe nalezne všechny třídy. Po zvolení nějaké třídy se na tuto třídu opět použije reflexe a aplikace vytvoří grafické uživatelské rozhraní s poli pro atributy třídy. Aplikace také dokáže zobrazit rozsahy jednotlivých atributů a to pokud autor třídy napsal tyto rozsahy do komentářů, jelikož aplikace ukládá komentáře do `.xml` souboru [21].

Po vyplnění všech hodnot je možné spustit test s nastavenými hodnotami. Výsledek testu je buď `true` nebo `false` a také dokáže zobrazit možné výjimky. Výsledky testů lze uložit do dokumentu. Pokud autor potřebuje, může spustit automatické testovaní, které do parametrů dosadí náhodné hodnoty a automaticky test vyhodnotí [21].

### 3.6.2 Výsledky

Při testování bylo zjištěno, že aplikace pracuje správně jen pro třídy nebo funkce, které mají jednoduché typy atributů, jako jsou `int`, `double` a další. Pro jednoduché typy ale aplikace pracuje správně a usnadňuje testerům práci. Aplikaci lze také použít pro white box testování [21].

# 4 Reflexe v Javě

Reflexe v Javě umožňuje zjišťování informací o programu a jeho třídách během běhu programu. Díky ní lze zjistit, jaké atributy má zvolená třída, jaké konstruktory obsahuje a také jaké obsahuje metody. Nalezené atributy lze nastavovat, konstruktory a metody lze volat a provádět tak akce jimi určené. Vše potřebné k práci s Java reflexí je uloženo v jednom ze základních balíků Javy (`java.lang.reflect`) [22].

## 4.1 Třída `Class`

Základním prvkem Java reflexe je třída `Class`, která reprezentuje třídu v programu, tedy její kompilovaný `class` soubor. Instanci této třídy lze získat několika způsoby, například metodou `forName()`, která vrací instanci požadované třídy podle jména, pokud ji nalezne, nebo přidáním přípony `.class` za název požadované třídy, například `Class.class`. Další možností jsou pak metody `getType()` z tříd Java reflexe. Z jednotlivých instancí třídy `Class` je poté možné získávat další potřebné informace [23].

Z tříd lze získat jejich modifikátory, rozhraní, která implementují, anotace, atributy nazývané *fields*, konstruktory, metody, jména tříd, balíky do kterých patří, nebo jejich rodičovskou třídu [23].

## 4.2 Třída `Field`

Atributy tříd jsou reprezentovány třídou `Field`. Po získání instance atributu je možné zjistit jeho jméno, jeho typ a další informace. Dále je možné ho nastavit na určitou hodnotu, ovšem pro toto je nutné mít už vytvořenou instanci třídy, které atribut patří. Atributy mají zvláštní metody pro nastavování hodnot primitivních typů a jednu metodu pro nastavování hodnot obecných objektů. Pomocí reflexe je možné zjistit i privátní atributy, které nejsou nijak zvlášť odlišeny od atributů neprivátních, ale pro jejich nastavení je nutné je metodou `setAccessible()` nastavit na použitelné [24]. Podobně jako u třídy `Class` je možné u každého atributu zjistit jeho další modifikátory a další informace [25].

## 4.3 Třída `Constructor`

Konstruktory jednotlivých tříd jsou reprezentovány třídou `Constructor`. Získaným konstruktorem je možné vytvořit instanci třídy, kterou reprezentuje třída `Class`, pomocí metody `newInstance()`. Konstruktory s argumenty ale k vytvoření objektu potřebují zadané hodnoty obdržet. Tyto hodnoty lze uložit do pole typu `Object` a poté je předat do metody vytvářející instanci požadované třídy. V případě konstruktorů vnitřních tříd a privátních konstruktorů je opět nutné je metodou `setAccessible()` nastavit na použitelné [24]. Modifikátory a další informace lze zjistit i u konstruktorů [26].

## 4.4 Třída `Method`

Další třídou je třída `Method`, která reprezentuje jednotlivé metody programu. Tyto metody je poté možné volat a provádět jimi určené akce, ovšem pro některé metody je nutné mít vytvořenou instanci objektu, nad kterou chceme metodu zavolat. Pro privátní metody lze opět použít metodu `setAccessible()` a tím povolit jejich používání zvenčí [24]. U metod s parametry je jako u konstruktorů opět nutné dodat potřebné parametry. Pokud uživatel reflexe potřebuje zjistit, zda-li je metoda *getter* nebo *setter*, je možné zjistit z jejího jména jestli nezačíná na `"get‘‘` nebo `"set‘‘` a má-li správný počet parametrů [27]. I zde je opět možné zjistit modifikátory metod [28].

## 4.5 Genericita

Další důležitou vlastností Javy je genericita. Pomocí reflexe ji lze v určitých případech (například z atributu reprezentovaném třídou `Field`) získat. Slouží k tomu třída `ParametrizedType`, která v sobě uchovává všechny informace o použité genericitě, pokud je typ, tedy třída, generický. Tato informace je uložena v poli, které se získá pomocí metody `getActualTypeArguments()` pro třídu `ParametrizedType`. Pokud tedy například zjišťujeme genericitu pro atribut, který představuje `List<T> list`, musíme nejprve zavolat nad jeho instancí `Field` metodu `getGenericType()`, která vrací instanci třídy `Type`. Zajímavé je, že metoda `getType()` vrací instanci třídy `Class`. Poté musíme získanou instanci přetypovat na typ `ParametrizedType` a až poté lze získat metodou `getActualTypeArguments()` pole s generickými typy. Typ `<T>` tedy bude na pozici *0*. Pokud v aplikaci existuje pole s generickými objekty, je možné získat generický typ pomocí třídy `GenericArrayType` a

pomocí její metody `getGenericComponentType()`. Pokud má pole více dimenzí, je nutné tuto metodu zavolat nad instancí třídy `GenericArrayType` vícekrát a po nalezení konečného generického typu je nutné výstup z této metody přetypovat na typ `ParametrizedType`. Dále už lze s tímto typem provádět klasické operace [29].

## 4.6 Pole

Pro práci s poli slouží v reflexi třída `Array`. Pomocí ní lze vytvořit pole požadovaného typu, dimenzí a velikosti. Při vyšším počtu dimenzí lze jejich jednotlivé velikosti uložit do pole typu `int` a toto pole poté předat do vytvářející metody `newInstance()`. Pro zjištění typů hodnot uložených v poli slouží metoda `getComponentType()` třídy `Class`, která vrací typ hodnoty formou instance třídy `Class`. Jelikož vícerozměrná pole jsou v Javě reprezentována jako pole polí, vrátila by tato metoda pro vícerozměrné pole jen typ vnitřního pole. Proto je nutné zavolat tuto metodu nad vráceným typem několikrát, dokud nenarazíme na konečnou hodnotu, tedy takovou, která už nebude reprezentovat instanci pole. Pro nastavování hodnot v poli slouží několik metod podobných nastavování hodnot atributů. Existují tedy metody pro nastavení primitivních hodnot a metoda pro nastavení libovolného objektu. Každá metoda navíc vyžaduje ještě parametr s indexem v poli, do kterého chceme hodnotu uložit [30].

## 4.7 Další možnosti

Pomocí reflexe lze za běhu aplikace také získat anotace v ní použité. Tyto anotace poté reprezentuje třída `Annotation`. Anotace lze získat ze tříd, jejich atributů, metod, parametrů metod a konstruktorů. Z instancí anotací potom lze získat další potřebné informace [31].

Dále je možné pomocí Java reflexe vytvářet proxy objekty, které mohou sloužit pro řízení transakcí, správu databáze, nebo mohou být použity v jednotkovém testování jako *mock* objekty [32], což je objekt, který simuluje funkci jiné komponenty, která tak ještě nemusí být vytvořena. Proxy objekty reprezentuje třída `Proxy` a lze je vytvářet prostřednictvím metody `newProxyInstance()` [33].

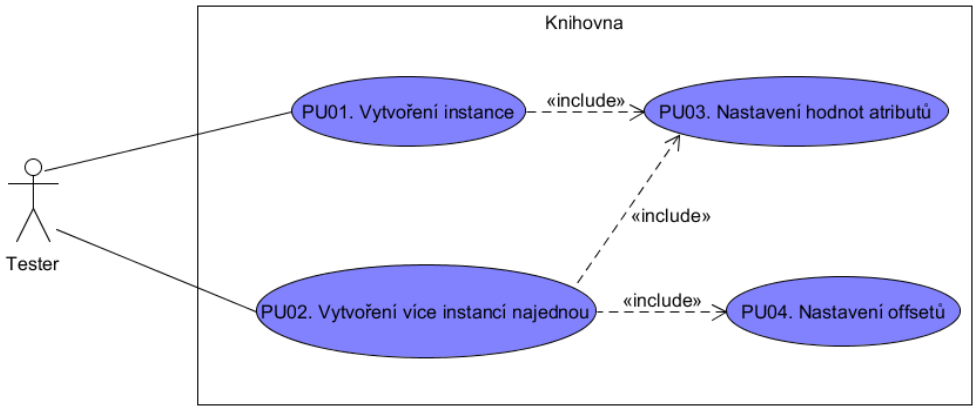
V Javě 9 také umožňuje pracovat s moduly, jež jsou vlastně skupiny balíčků v projektu [34].

# 5 Knihovna a její návrh

Cílem této práce je vytvoření Java knihovny, která bude umožňovat uživateli vytváření komplexních objektů pro testování a nastavování jejich atributů pomocí uživatelského rozhraní a tím mu usnadnit testování aplikací. Vytvořenou knihovnu ale bude možné použít i pro jiné účely, nicméně pro testování aplikací pomocí jednotkových testů bude její použití nejpraktičtější. Hlavní výhodou vytvářené knihovny bude to, že pro její použití bude uživateli stačit tuto knihovnu přidat do projektu, což lze snadno udělat pomocí běžně užívaných vývojových prostředí a poté už jen využívat služeb knihovny. Nebude tedy zapotřebí nějak upravovat zdrojové kódy aplikace, ale bude pouze stačit, když uživatel zavolá metodu vytvořené knihovny ve třídách s testy.

## 5.1 Případy užití

Jednotlivé případy užití pro vytvářenou knihovnu jsou zobrazeny na obrázku 5.1.




Obrázek 5.1: Diagram případů užití knihovny.

### 5.1.1 Vytváření instancí

Uživatel bude moci po zavolání jedné z metod knihovny vytvořit instanci požadovaného komplexního typu. Tuto instanci bude moci vytvořit pomocí zadání hodnot jednotlivých parametrů konstruktoru. Pokud bude parametr komplexního typu, bude uživatel moci vytvořit instanci pro tento parametr.

### 5.1.2 Vytváření více instancí najednou

Uživatel bude moci po zavolání jedné z metod knihovny vytvářet více instancí požadovaného komplexního typu najednou. Stejně jako v předchozím případě bude moci instance vytvářet pomocí zadání hodnot jednotlivých parametrů konstruktoru. Pro komplexní typy parametrů bude uživatel moci opět vytvořit instance pro tento parametr.

### 5.1.3 Nastavování hodnot atributů

Uživatel bude moci nastavovat hodnoty jednotlivých atributů vytvořené nebo vytvořených instancí. Tyto atributy bude moci nastavovat pomocí zadávání hodnot jednotlivých atributů. Pokud budou atributy komplexního datového typu, bude uživatel moci vytvářet i objekty pro tyto atributy.

### 5.1.4 Nastavování offsetů

Uživatel bude moci nastavit jednotlivé intervaly, o které se budou měnit atributy nebo parametry primitivních číselných typů v případě vytváření více instancí najednou a také v případě nastavování hodnot atributů pro více instancí najednou.

## 5.2 Zjišťování struktury knihovny

Pro implementaci knihovny byl zvolen podobný přístup jako v postupu popsaném v kapitole 3.6. Použita tedy bude reflexe, konkrétně Java reflexe, jež je součástí Javy. Důležitou výhodou Java reflexe, ale i reflexe v jiných jazycích, je možnost získávání informací o třídách a jejich atributech během běhu programu. Toto lze využít právě v jednotkových testech, které jsou součástí testovaného projektu a tak je jednoduché používat třídy do projektu patřící, které pravděpodobně chce uživatel testovat. Samozřejmě je také možné získávat informace o třídách a jejich atributech i když do projektu nepatří.

Knihovna by měla po zavolání její metody otevřít okno, které by umožňovalo vytváření a nastavování atributů jakéhokoliv objektu a pokud některé atributy nebudou primitivního typu (tj. bude se jednat o referenční proměnné ukazující na objekty), tak i možnost vytváření a nastavování atributů těchto datových typů. Tento proces je tedy rekurzivní a může probíhat do nekonečna i pro další atributy neprimitivního typu.

Uživatel by také měl mít možnost vytvoření více objektů najednou s tím, že by měl mít možnost pomocí posuvníků zvolit intervaly pro náhodné změny

hodnot pro primitivní číselné parametry nebo atributy.

## 5.3 Struktura knihovny

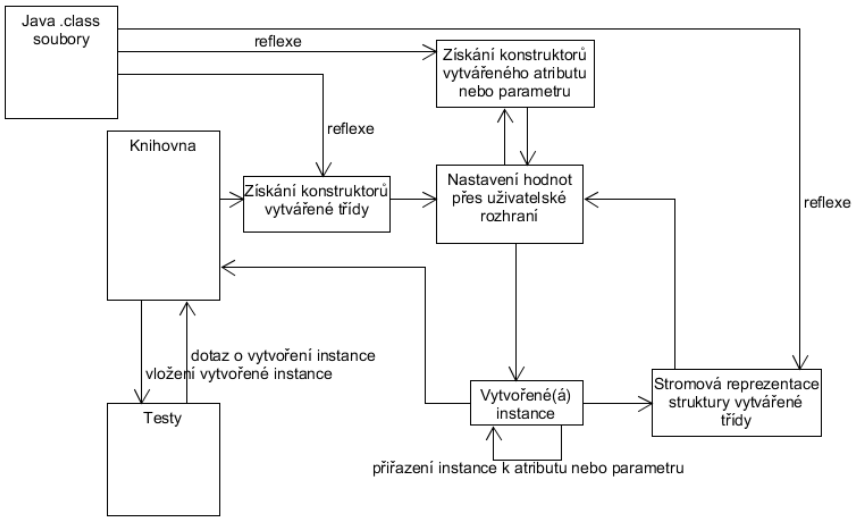
Celá knihovna bude rozdělena do tří částí. První se bude starat o práci s třídami, o jejich vyhledávání a prohledávání grafu, kterým lze vlastně strukturu objektu a jeho atributů reprezentovat a dále bude obsahovat také generátory dat a postupy pro přiřazování hodnot. Dále se také bude starat o kódové rozhraní s uživatelem. Obsahovat tedy bude metody, které bude uživatel moci zavolat pro vytvoření objektů. Další částí bude část s grafickým uživatelským rozhraním (dále jen GUI). Tato část bude obsahovat všechny reprezentace použitých oken a další postupy pro práci s uživatelským rozhraním. Třetí část poté bude obsahovat pomocné struktury a konstanty.

## 5.4 Užívání knihovny

V postupu [21] v kapitole 3.6 byla vytvořená aplikace schopna vytvářet objekty a nastavovat hodnoty jen primitivních atributů, což jí velmi limitovalo. Vytvářená knihovna tuto limitaci částečně odstraní. Knihovna bude umožňovat vytváření instancí běžných tříd, které mají veřejný konstruktor. Instance tříd bez veřejného konstruktu nebude možné vytvořit, protože tyto třídy mají obvykle konstruktor nastaven jako privátní z toho důvodu, že jejich autoři nechtějí, aby instance těchto tříd byly vytvářeny. Dále bude možné s určitými omezeními vytvářet instance vnitřních tříd, jak třídních (statických), tak i těch netřídních. Dalším typem tříd jsou třídy využívající genericitu. Tyto třídy vytvořit půjde, ale nebude možné jim nastavit typ genericity. Dále bude knihovna speciálně podporovat třídy `ArrayList` a `HashMap`, protože tyto třídy jsou velmi často používané. Knihovna by měla být snadno rozšiřitelná o speciální podporu dalších tříd. Také bude v knihovně implementována speciální podpora pro pole, jejich vytváření a nastavování jejich hodnot, pro které bude možné vygenerovat náhodná čísla.

Vytvářeno také bude moci být více objektu najednou. Uživatel si bude moci během vytváření zvolit, zda-li budou všechny objekty stejné a nebo bude moci nastavit změnu hodnot primitivních číselných parametrů či atributů pro právě používané okno.

V obrázku 5.2 je zobrazen celkový algoritmus využití vytvářené knihovny. Java `.class` soubory knihovna získá z `build` složky projektu. Dále pak *Knihovna* je vytvářená knihovna, jejíž instance je vytvořena ve třídě pro jednotkové testy.




Obrázek 5.2: Ukázka možného využití třídy.

## 5.5 Vytváření instancí

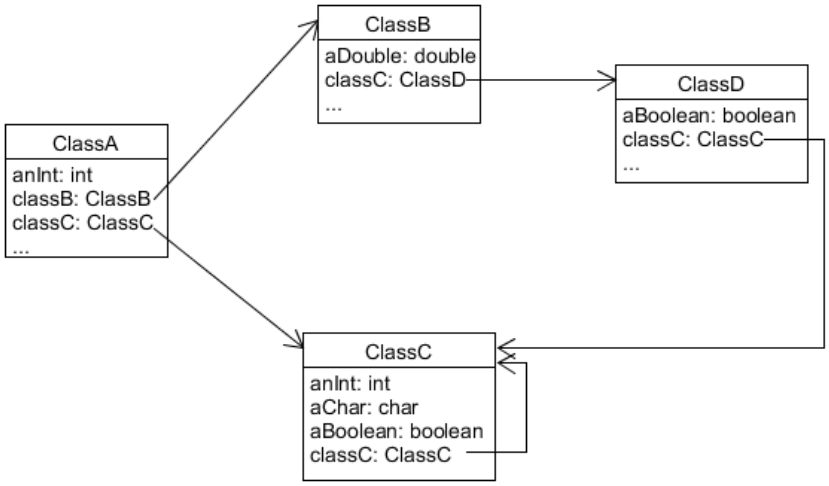
Nejdůležitější částí knihovny bude část provádějící vytváření instancí. To bude knihovna provádět dvěma způsoby. První způsob bude vytváření podle zvoleného konstruktoru a dat parametrů zadaných uživatelem. U třídy, jejíž instanci bude uživatel chtít vytvořit tímto způsobem, budou zjištěny všechny veřejné konstruktory, z nichž si bude uživatel muset jeden vybrat. Podle zvoleného konstruktoru bude muset zadat knihovně data pro parametry zvoleného konstruktoru. Po zadání dat a jejich potvrzení dojde k dosazení těchto dat do konstruktoru a k vytvoření instance pomocí zvoleného konstruktoru.

Druhým způsobem vytváření instancí, který bude možné použít v části starající se o nastavování hodnot atributů, je automatické vytváření instancí pomocí základních hodnot. Tento způsob bude implementován pouze pro usnadnění práce uživatele, kdy nebude muset vytvářet instance atributů manuálně, ale bude mít na výběr mezi manuálním a automatickým vytvářením, například v případě, kdy automatické vytváření selže. Dojde tedy automaticky k vybrání prvního konstruktoru a zjištění jeho parametrů. Dále dojde k automatickému vložení základních hodnot primitivních parametrů a k nastavení ostatních parametrů na hodnotu `null`. Následně dojde k vytvoření objektu s těmito parametry pomocí prvního konstruktoru.

Konstruktory a jejich parametry lze získat pomocí metod implementovaných v Java reflexi, zároveň je pomocí Java reflexe možné do konstruktorů dosadit hodnoty parametrů a konstruktory vyvolat. Vše je popsáno v kapitole 6.2.

## 5.6 Reprezentace závislostí

Třída, jejíž instanci bude chtít uživatel vytvořit a u níž bude chtít nastavit další atributy, může mít jakoukoliv strukturu. Tuto strukturu je možné popsat jako graf, který může obsahovat cykly, kde jednotlivé atributy třídy (i zděděné) představují uzly. Na obrázku 5.3 je zobrazena možná cyklická struktura. Pro procházení této grafové reprezentace byl použit algoritmus *Breadth-first-search*, dále jen BFS, který prohledává graf do šířky, dá se tedy říci, že po vrstvách. Další možností bylo použití algoritmu *Depth-first-search*, dále jen DFS, který prohledává graf do hloubky. Algoritmus DFS ale kvůli jeho průchodu grafem úplně nevyhovuje potřebám vytvářené knihovny. Dále je důležité zmínit, že procházení grafu je vždy nutné nějak omezit, jinak by mohlo dojít k zacyklení, pokud by prohledávaná struktura obsahovala cykly. Procházení bude tedy omezeno nerozvětvováním již navštívené třídy (tedy vrcholu grafu), což je základní vlastnost algoritmu BFS, ale může být také omezeno například hloubkou procházení, kde by se prohledávání zastavilo po dosažení určité hloubky. Zvolené omezení umožňuje lepší rozvětvení částí, ve kterých se vyskytují třídy, které se v jiných částech nevyskytují.




Obrázek 5.3: Možný graf závislostí čtyř tříd.

## 5.7 Nastavování atributů

Nastavování atributů bude probíhat rekurzivně. Ze získané grafové reprezentace, popsané v předchozí kapitole, se vybere první, startovací, uzel. Dále dojde k postupnému nastavování hodnot atributů a to tak, že pokud bude

atribut primitivního typu, nebo výčtového typu nebo typu `String`, nastaví se mu hodnota zvolená uživatelem, pokud bude typu pole, `ArrayList` nebo `HashMap`, nastaví se na vytvořenou instanci tohoto typu, nebo na hodnotu `null`. Nastavovat nepůjde atributy, které mají současně modifikátory static a final, protože to jsou konstanty.

Poslední možností jsou ostatní datové typy, u kterých dojde k nastavení na hodnotu vytvořené instance uživatelem, nebo k automatickému vytvoření instance, jež bylo zmíněno výše, a následnému nastavení atributu na tuto hodnotu. Do nastavování atributů bude moci uživatel zasáhnout tak, že si bude moci vybrat, zda bude hodnota tohoto atributu `null`, nebo zda se má nebo nemá dále větvit. Pro tuto třídu bude poté rekurzivně použita popisovaná metoda.

## 5.8 Pole, ArrayList a HashMap

Speciálním druhem objektů jsou pole. Pro vytváření instancí polí a nastavování jejich hodnot jsou v Java reflexi ve třídě `Array` implementovány metody, které budou pro práci s poli použity. Tyto metody jsou popsány v kapitole 6.5.

Hlavní problém při nastavování hodnot polí tvoří vícerozměrná pole. Ta jsou v Javě vytvářena jako pole polí. To znamená, že dvourozměrné pole tvoří vlastně pole, které má na svých indexech uložena další pole, která mají na svých indexech uloženy hodnoty. Vícerozměrná pole lze tedy reprezentovat jako stromovou strukturu. Knihovna tedy pomocí algoritmu pro procházení stromů dojde na poslední pole ve stromové struktuře a těm nastaví požadované hodnoty. Pro procházení byl opět zvolen algoritmus BFS.

Dalším datovým typem se speciální podporou je typ `ArrayList`. Typ `ArrayList` ukládá komplexní datové typy, které je nutné vytvořit. Vytváření instancí bude probíhat uživatelem, stejným způsobem jako bylo popsáno v kapitole 5.5. Vytvořené instance poté budou uloženy do `ArrayListu`. Do `ArrayListu` bude také možné přidat více objektů najednou. Důležité je také zmínit, že `ArrayList` je generickým datovým typem, ale na genericitu se nebude brát ohled. Vytvořené instance `ArrayListu` budou vytvářeny bez nastavení genericity a do těchto instancí se budou vkládat instance typu `Object`. O přetypování by se měla starat Java sama.

Posledním speciálním typem je typ `HashMap`. `HashMapa` ukládá dvojici klíč a hodnota. Práce s `HashMapou` tedy bude fungovat na podobném principu jako práce s `ArrayListem`. Uživatel bude muset vytvořit jak klíč, tak hodnotu, které se poté přidají do `HashMapy`. Oproti `ArrayListu` nepůjde vy-

tvořit více klíčů a hodnot najednou. Opět se nebude brát zřetel na genericitu, `HashMapa` tedy bude ukládat instance typu `Object`.

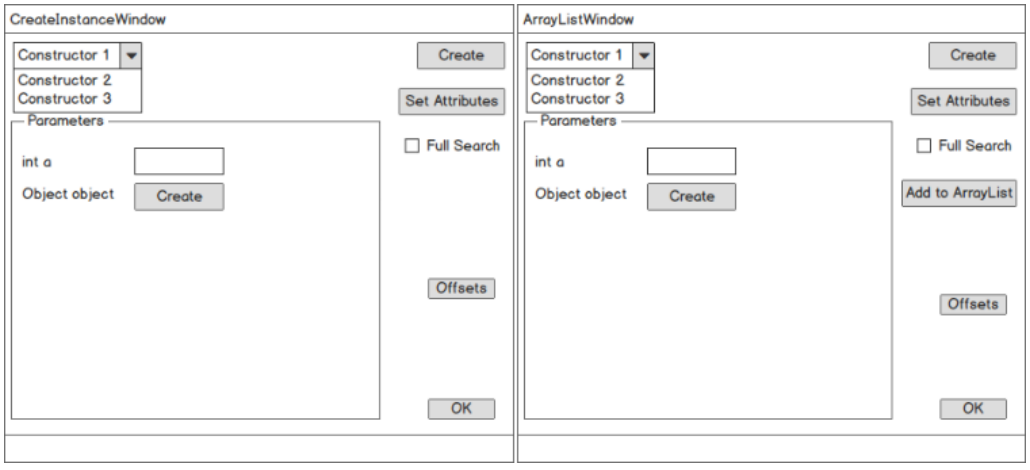
## 5.9 Vytváření více objektů najednou

Knihovna bude umožňovat vytváření více objektů najednou. Pro testovací využití bude dobré, pokud budou mít vytvořené objekty jiné hodnoty primitivních atributů. Toho bude docíleno pomocí nastavení jednotlivých intervalů změn pro primitivní číselné datové typy. Interval bude udávat, o kolik se má číselná hodnota změnit. Změněné číslo pak tedy bude náhodné číslo v tomto intervalu. Vytváření instancí a nastavování hodnot atributů bude probíhat najednou pro všechny instance.

## 5.10 Uživatelské rozhraní

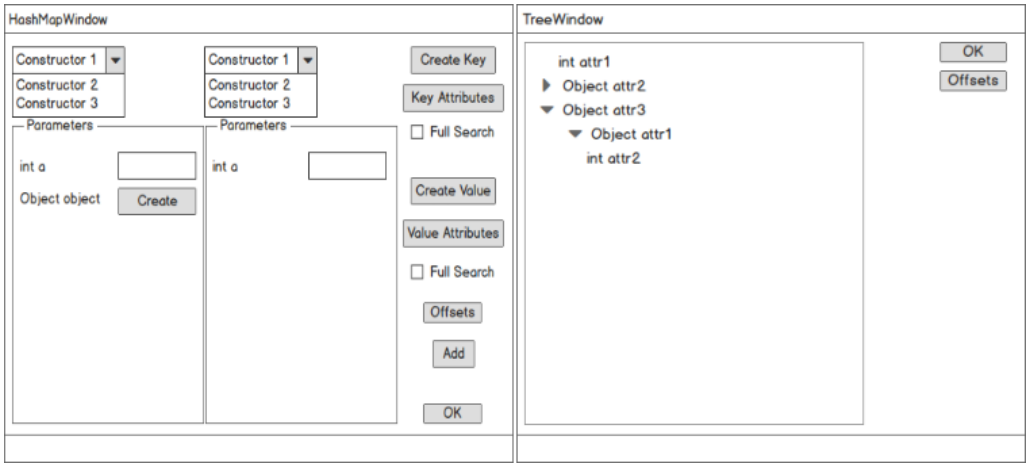
Knihovna bude obsahovat dva typy uživatelského rozhraní a to programové a grafické. Programové uživatelské rozhraní bude uživateli sloužit pro zvolení počtu vytvářených instancí a jejich typu. Bude provedeno zavoláním jedné z metod knihovny.

Pro implementaci grafického uživatelského rozhraní byl použit framework *JavaFX*, jež je součástí instalace Javy. Pro jeho zvolení rozhodovalo jeho snadné použití a hlavně to, že se v dnešní době jedná o standardní způsob vytváření grafického uživatelského rozhraní pro desktopové aplikace, na rozdíl od *Java Swing*. Uživatelské rozhraní bude reprezentováno několika okny, ale celkový počet otevřených oken může být neomezený.

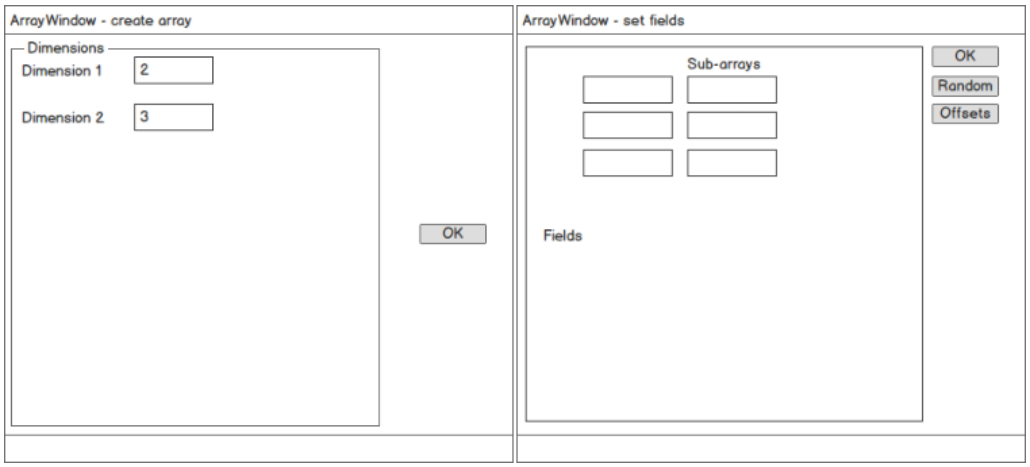



Obrázek 5.4: Návrhy oken pro vytváření instancí a práci s ArrayListem.

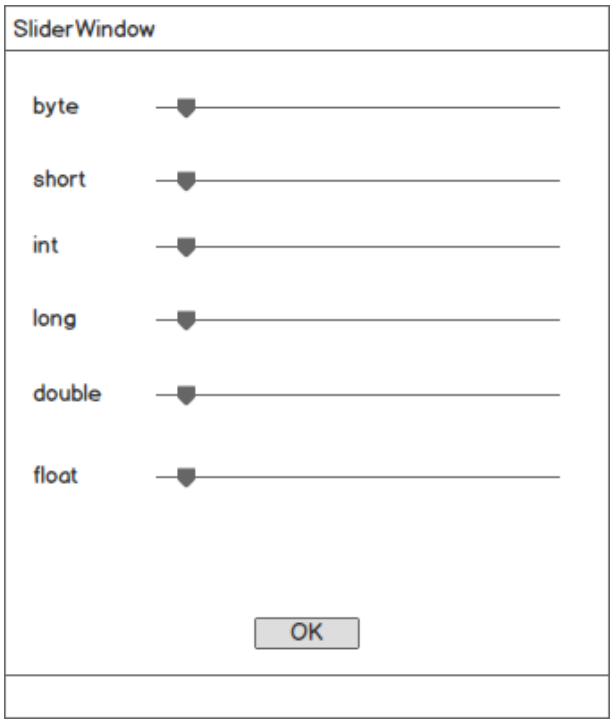
Okno pro vytváření objektů bude obsahovat část s grafickými prvky pro zadávání hodnot parametrů. Dále pak okno pro nastavování atributů bude obsahovat stromový pohled, který bude obsahovat grafické prvky pro nastavování hodnot atributů. Okno pro nastavování hodnot polí bude obsahovat grafické prvky v mřížkovém zobrazení. Okno pro vytváření `ArrayListů` bude využívat podobné zobrazení, jako okno pro vytváření instancí a okno pro `HashMapu` bude rozdělené do dvou částí, jedné pro klíč a jedné pro hodnotu. Posledním oknem bude okno pro nastavování změn primitivních číselných hodnot a bude obsahovat posuvníky pro nastavení hodnot.




Obrázek 5.5: Návrhy oken pro práci s HashMapou a pro nastavování atributů.




Obrázek 5.6: Návrhy oken pro práci s poli. První okno slouží k vytvoření pole, druhé pak k zadávání hodnot do pole.




Obrázek 5.7: Návrh okna pro nastavování offsetů.

Návrhy vzhledů jednotlivých oken použitých ve vytvořené aplikaci jsou zobrazeny na obrázcích 5.4, 5.5, 5.6 a 5.7. Aplikace dále bude obsahovat různá vyskakovací okna pro informování uživatele.

# 6 Implementace knihovny

Knihovna byla implementována bez jakýchkoliv externích knihoven. Její implementace je rozdělena do tří balíků a to do balíku `app`, který obsahuje třídy, které slouží pro práci s třídami, vytváření instancí a práci s nimi a pro interakci s uživatelem pomocí kódu. Dalším balíkem je `gui`, který obsahuje třídy reprezentující jednotlivá okna, obsahující metody pro vytváření GUI. Dále obsahuje pomocné třídy, které obsahují jednoduché metody pro práci s uživatelským rozhraním. Poslední je balík `support`, který obsahuje pomocné třídy. Všechny tři balíky leží v hlavním balíku, který se jmenuje `testingtool`. Dále knihovna obsahuje balík s testy knihovny a se strukturami použitými pro testování. Celá knihovna pak nese název *TestingTool*.

## 6.1 Hlavní třídy knihovny

Knihovna obsahuje celkem 18 tříd, do kterých se nezapočítávají třídy z testování knihovny. Balík `app` obsahuje třídy reprezentující aplikační vrstvu, balík `gui` pak vrstvu s grafickým uživatelským rozhraním. Posledním balíkem je balík `support`, jež obsahuje pomocné třídy a struktury. Strukturu knihovny lze najít v UML diagramu v příloze B.

### 6.1.1 Třídy `TestingTool`, `ClassFinder` a `DataConverter`

Třída `ClassFinder` se stará o nalezení tříd a vytvoření reprezentace struktury projektu pomocí grafu. Patří do balíku `app`. Obsahuje také metodu pro zjištění, zda třída patří do projektu, tedy, jestli v něm byla vytvořena. Pro prohledání projektu je nutné zadat hlavní balík projektu.

Jelikož je vytvořená aplikace knihovna, neobsahuje žádnou hlavní třídu a tedy ani metodu `main()`. Nicméně lze za hlavní třídu považovat třídu `TestingTool`. Tato třída má za úkol inicializovat *JavaFX Toolkit*, a poté spustit GUI a také vytvořit synchronizační primitivum a pozastavit běh hlavního vlákna. Metody, které uživatel zadává pro spuštění uživatelského rozhraní umožňují vytváření jednoho nebo více objektů. Tato třída tedy vlastně slouží jako programové rozhraní mezi uživatelem a samotnou knihovnou. Tato třída také patří do balíku `app`.

Třída `DataConverter` pak slouží k získání dat z GUI a jejich převodu do podoby pro použití v aplikační vrstvě. Jedná se tedy o rozhraní mezi GUI a aplikační vrstvou. Patří do balíku `support`.

### 6.1.2 Třídy `CreateInstanceData`, `ArrayListData` a třída `HashMapData`

Třída `CreateInstanceData` slouží k vytváření instancí. Obsahuje metody pro práci s konstruktory a pro nalezení jejich parametrů. Dále obsahuje metodu pro vytvoření pole se zadanými parametry a jeho použití pro vytvoření instance pomocí zvoleného konstruktoru.

Třída `ArrayListData` je poté použita pro vytváření objektů, které jsou typu `ArrayList`. Obsahuje mimo jiné metody pro přidávání dat do vytvořených `ArrayListů`. Tato třída dědí od třídy `CreateInstanceData`.

Třída `HashMapData` opět dědí od třídy `CreateInstanceData` a jejím účelem je vytváření objektů typu `HashMap` a přidávání dat do nich. Všechny tři třídy patří do balíku `app`.

### 6.1.3 Třídy `ArrayData` a `TreeData`

Třída `ArrayData` obsahuje metody pro vytvoření polí pomocí zadaných parametrů, prohledání stromové reprezentace pole, pokud je vícerozměrné, a pro nastavení hodnot v poli. Při nastavení hodnot komplexních typů se také stará o jejich přetypování.

Další třídou je třída `TreeData` sloužící pro přiřazování hodnot atributům tříd. Obsahuje také metodu pro automatické vytvoření argumentů konstruktoru a vytvoření instance. Obě patří do balíku `app`.

### 6.1.4 Třída `CreateInstanceWindow`

Třída `CreateInstanceWindow` reprezentuje okno, které slouží pro vytváření instancí uživatelem. Obsahuje tedy metody pro vytváření GUI a pro obsluhu tlačítek. Tato třída patří do balíku `gui`.

Ve své podstatě se jedná o hlavní okno GUI, protože je spuštěno jako první. Obsahuje tedy synchronizační primitivum, které pozastavuje činnost hlavního vlákna, tedy například vlákna s jednotkovými testy, ve kterém byla použita vytvářená knihovna a tedy otevřeno toto první okno, dokud neskončí běh jiného vlákna, tedy vytvářené knihovny. Okno může být v průběhu aplikace otevřeno vícekrát, ale synchronizační primitivum obsahuje jen to první, protože toto okno je vždy uzavřeno jako poslední. Při uzavření prvního okna se tedy stará o uvolnění synchronizačního primitiva a tím o pokračování běhu hlavního vlákna. Jako synchronizační primitivum byla použita instance třídy `CountDownLatch`, která dovoluje pozastavit vlákno v té části, ve které je nad instancí zavolána metoda `await()`. Dále musí být

ve vlákně (vláknech), které má běžet dál, v našem případě vlákno grafického uživatelského rozhraní, zavolána metoda `countDown()` a to tolikrát, s jakou hodnotou byla instance `CountDownLatch` inicializována, což je v případě vytvářené knihovny hodnota *1*, jelikož jí potřebujeme uvolnit pouze po uzavření prvního okna. Instance třídy `CountDownLatch` je vytvářena v metodě sloužící jako programové rozhraní s uživatelem a dále je předána prvnímu oknu aplikace. Ostatní okna mají tuto instanci nastavenou na `null` a nic s ní nedělají.

### 6.1.5 Třída `ArrayWindow`, třída `ArrayListWindow` a třída `HashMapWindow`

Třída `ArrayWindow` je použita pro reprezentaci okna pro vytváření polí.

Další třída (`ArrayListWindow`) reprezentuje okno pro práci s typem `ArrayList` a některé postupy dědí od třídy `CreateInstanceWindow`.

Třída `HashMapWindow` opět jako v případě předchozí třídy dědí některé metody od třídy `CreateInstanceWindow`.

Všechny třídy obsahují metody pro vytváření GUI a pro obsluhu jednotlivých použitých tlačítek a patří do balíku `gui`.

### 6.1.6 Třída `TreeWindow`

Třída `TreeWindow` je třída obsahující grafickou stromovou reprezentaci části grafové reprezentace struktury projektu. Umožňuje nastavování hodnot atributů objektů (i zděděných). Opět obsahuje metody pro vytvoření GUI a pro obsluhu tlačítek. Tato třída patří do balíku `gui`.

### 6.1.7 Třídy `SliderWindow`, `Alerts`, `ComponentMethods` a `Constants`

Třída `SliderWindow` je použita pro reprezentaci okna, pomocí kterého uživatel nastavuje hodnoty intervalů, pro primitivní číselné typy, ve kterých se budou generovat hodnoty pro další vytvářené instance (pokud jich je více). Nastavení těchto hodnot není globální, ale platí pro právě používané okno. Její metody slouží pro vytváření GUI, obsluhu tlačítek a pro práci s dalšími grafickými prvky.

`Alerts` a `ComponentMethods` jsou třídy, které obsahují pomocné metody pro práci s GUI. Třída `Alerts` obsahuje různé hlášky a upozornění. Všechny tři třídy patří do balíku `gui`.

Třída `Constants` obsahuje konstanty použité v celé knihovně a také pomocné metody, které nijak nesouvisí s grafickým uživatelským rozhraním. Jedná se o generátory náhodných čísel pro primitivní číselné atributy nebo parametry. Tato třída patří do balíku `support`.

### 6.1.8 Třída `ItemData`

Třída `ItemData` je struktura reprezentující uzel grafové reprezentace struktury projektu, ve kterém je knihovna použita. Obsahuje informace o uzlu a dále pak `ArrayList`, ve kterém jsou uloženy všechny následující uzly, vlastně tedy atributy třídy, kterou uzel reprezentuje. Obsahuje například vytvořené instance třídy (uzlu), pokud není primitivní, nebo různé grafické prvky k němu připojené. Tato třída je použita výhradně v okně reprezentovaném třídou `TreeWindow` a je tedy použita pro nastavování hodnot atributů. Tato třída je z balíku `support`.

## 6.2 Vytváření instancí podle zadaných hodnot parametrů uživatelem

### 6.2.1 Reprezentace parametrů

Každý parametr konstruktoru je reprezentován nějakým grafickým prvkem. Jedná-li se tedy o parametr primitivního číselného typu, typu `char` nebo typu `String`, pak je v knihovně reprezentován jako textové pole. Pokud se jedná o výčtový typ, nebo o typ `boolean`, je tento parametr reprezentován výběrovým boxem. Ve výběrovém boxu jsou pak uloženy všechny možné hodnoty výčtového typu, respektive typu `boolean`, tedy `true` nebo `false`. Posledním typem reprezentace parametru je tlačítko, které je použito pro reprezentaci pole, `ArrayListu`, `HashMapy` a ostatních komplexních typů. Všechny tyto grafické prvky jsou uloženy v seznamu. Dále je použito pole typu `Object`, které bude v následujícím textu nazýváno `objects`. Toto pole obsahuje instance vytvořených komplexních objektů, vyjma typu `String`, tak, jak jdou v konstruktoru popořadě.

### 6.2.2 Vytvoření parametrů

Vytváření parametrů je prováděno v metodě `createArguments()`, která je implementována ve třídě `CreateInstanceData`. Jejím výstupem je pole typu `Object`, pojmenované `args`, které má velikost danou počtem parametrů konstruktoru. Pro svoji funkčnost potřebuje mít k dispozici pole typu `String`

s názvem `strings`, které obsahuje hodnoty jednotlivých grafických prvků pro parametry, nebo hodnoty `null` v případě, že grafický prvek byl tlačítko. Převod dat z grafických prvků do pole `strings` zajišťuje třída `DataConverter`. V algoritmu pak dochází k procházení pole s parametry.

Nejprve dochází ke zjištění, zda-li je parametr výčtový typ. Pokud ano, dojde k získání instance metody s názvem `"valueOf"`, kterou by měl výčtový typ obsahovat. Tato metoda je poté vyvolána metodou `invoke()`, která je součástí třídy `Method`. Jako druhý parametr je dodána hodnota z pole `strings`, která je dána pozicí parametru v poli parametrů. Výstupní hodnota vyvolané metody je poté uložena do pole `args` na příslušnou pozici.

Poté se zjišťuje, jestli je parametr komplexní datový typ. V případě, že ano, dojde ke zjištění, zda-li se nejedná o typ `String`. Pokud ano, je do pole `args` vložen celý řetězec z pole `strings` na příslušné pozici.

Pokud se nejedná o parametr typu `String`, dojde k získání instance z pole `objects` na pozici, kterou udává *index*. Tento *index* je zpočátku nastaven na hodnotu *0*. Je nutné zmínit, že na pozicích pole `objects` jsou uložena další pole objektů, kvůli možnosti vytváření více objektů najednou. To znamená, že se z pole `objects` získá další pole, podle pořadí komplexních parametrů pomocí *indexu*, a z toho se až bere požadovaná instance z pozice dané aktuálně vytvářenou instancí. Tato hodnota je pak vložena do pole `args` na svojí pozici danou pozicí parametru v konstruktoru. Pokud na daném *indexu* v poli `args` není žádné pole, tedy je tam hodnota `null`, je hodnota komplexního parametru nastavena na `null`. Nastavení hodnoty na `null` může způsobit problémy při vytváření instance. Tyto problémy budou popsány v kapitole 8. Po provedení akce dochází ke zvětšení *indexu* pro pole `objects` o *1*.

Poslední případ, který může nastat je, že parametr je primitivního typu. Zde nejprve musí dojít ke zjištění názvu typu parametru, pomocí metody `getName()`, třídy `Type`. Je také možné použít metodu `getSimpleName()` třídy `Type`, na výsledek to nemá vliv. Ze získaného názvu typu parametru se poté provádí akce v závislosti na tomto názvu. Pokud se tedy jedná o typ `char`, dojde k získání textu v poli `strings` a poté znaku na prvním místě z tohoto textu (textová pole, jsou upravena tak, aby přijímala vstup jen takový, jaký je typ parametru, takže zde je uložen pouze řetězec délky 1) a vloží ho do pole `args` na příslušnou pozici. Když se jedná o typ `boolean`, dojde k převedení textové hodnoty z pole `strings` na hodnotu `boolean` pomocí metody pro převod na typ `boolean` a uložení této hodnoty do pole `args`. Dále přicházejí v úvahu už jen primitivní číselné datové typy. U těch jsou řetězce z pole `strings` převedeny příslušnými postupy pro převod na daný číselný datový typ. Tak se děje pokud se jedná o první vytvářenou

instanci. U ostatních dochází ke generování náhodného čísla ze zadaného čísla ve zvoleném intervalu. Výsledná čísla jsou opět uložena do pole `args`.

Celý postup lze popsat pseudokódem v algoritmu 1, který pro zjednodušení nebere v úvahu kontrolu vstupu a možnost vytváření více instancí.

**Algorithm 1** Algoritmus pro vytváření pole argumentů pro konstruktor

```
strings                                          ▷ Pole s daty z GUI uzlů
objects                                          ▷ Pole objektů
parameters                          ▷ Pole parametrů vybraného konstruktoru
procedure CREATEARGUMENTS(strings, object, parameters)
    args[]
    for i = 0; i < parameters.length; i++ do
        if parameter[i].type is Enum then
            method = getMethod("valueOf")
            args[i] = method.invoke(strings[i])
        else if parameter[i].type is not primitive then
            if parameters[i].type is String then
                args[i] = strings[i]
            else
                args[i] = objects[index]
                index++
            end if
        else
            if parameters[i].type is boolean then
                args[i] == parseBoolean(strings[i])
            else
                args[i] = parseNumber(strings[i])
            end if
        end if
    end for
end procedure
```

### 6.2.3 Vytvoření instancí

Při vytváření instancí se nejprve vytvoří pole s argumenty parametrů konstruktoru pomocí výše popsaného algoritmu. Dále dojde k použití metody `newInstance()` třídy `Constructor` volané nad zvoleným konstruktorem, který je uložen jako třídní proměnná. Tato metoda může být zavolána několikrát, v závislosti na zvoleném počtu instancí uživatelem. Výsledné instance se ukládají do pole typu `Object`. Metoda, která toto provádí, se

jmenuje `createInstances()` a její implementaci je možné nalézt ve třídě `CreateInstanceData`. Dále je překryta a použita ve třídě `ArrayListData` a její modifikace jsou také použity ve třídě `HashMapData`.

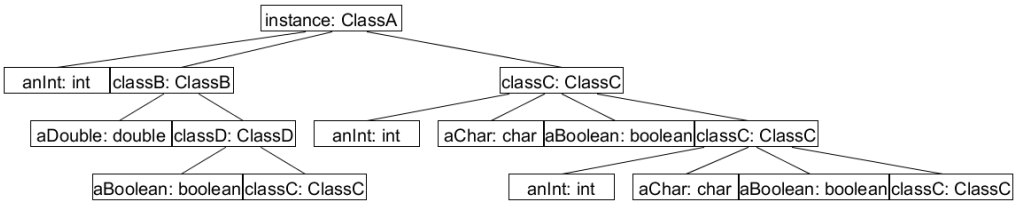
## 6.3 Vytváření stromové reprezentace

Metoda pro vytvoření stromové reprezentace je implementována metodou `search()` a nachází se ve třídě `ClassFinder`. Její návratovou hodnotou je instance třídy `ItemData`, která reprezentuje kořenový uzel stromu. Parametr `sClass` je instance třídy `Class` a udává třídu, která bude kořenem stromu. Parametr `classes` je pak pole, obsahující všechny třídy, které byly vytvořeny v projektu. Posledním parametrem je parametr `isFullSearch`, který udává, zda se mají větvit i třídy, které nebyly vytvořeny v projektu, ale jsou například v použitých knihovnách, nebo mimo hlavní balík aplikace. V následujícím popisu se předpokládá, že hodnota parametru `isFullSearch` bude nastavena na `false`, tudíž nebude docházet k větvení tříd mimo hlavní balík projektu a zároveň se na to, jestli třída patří nebo nepatří do balíku, nebere ohled. Také se nekontroluje, zda-li je atribut konstanta.

Algoritmus je vlastně modifikací algoritmu BFS. Musí tedy obsahovat frontu a seznam navštívených uzlů. Nejprve dojde k vytvoření kořenového uzlu ze zadaného parametru `sClass`. Dále dochází k získání všech atributů třídy, tedy i těch zděděných (ty mohou být i privátní), pomocí metody popsané v kapitole 6.3.1.

Po nalezení všech atributů jsou z nich vytvořeny uzly, které jsou přidány předchozímu uzlu, tedy kořenovému uzlu, do seznamu potomků. To vše je provedeno v metodě `initializeItem()` a následně jsou tyto vytvořené uzly přidány do fronty. Následně dochází k procházení fronty uzlů a postupnému vybírání prvků tak jak je tomu u algoritmu BFS. Zde je to upraveno tak, že se pracuje pouze s typy, které jsou komplexní, nejsou výčtový typ, typ `String`, `ArrayList` nebo `HashMap`, a jsou v projektu (to lze změnit nastavením parametru `isFullSearch` na hodnotu `true`). Dochází tedy k nalezení všech atributů třídy, kterou reprezentuje uzel, vytvoření nových uzlů a jejich přiřazení k předchozímu uzlu a poté, pokud ještě třída nebyla navštívena, k přidání do fronty a do seznamu navštívených tříd se jménem `visited`. Přidávání do tohoto seznamu se děje, až když je algoritmus ve druhé vrstvě stromové reprezentace. Algoritmus končí po vyprázdnění fronty.

Příklad výsledku výše popsaného algoritmu pro graf závislostí z obrázku 5.3 je zobrazen na obrázku 6.1. Pseudokód algoritmu je možné nalézt v algoritmu 2.




Obrázek 6.1: Výsledná struktura stromu pro graf závislostí z obrázku 5.3

**Algorithm 2** Algoritmus pro vytváření stromové reprezentace

```
inheritedIndex  ▷ Index, od kterého začínají zděděné atributy v seznamu
type                                                    ▷ Typ atributu
special                                  ▷ Pokud je typ pole, ArrayList, atd.
procedure SEARCH(class)
    queue
    visited
    node = new Node(class)
    fields
    inheritedIndex = loadAllFields(fields, class)
    for each field in fields do
        newNode = initializeItem(node, inheritedIndex)
        queue.add(newNode)
    end for
    while queue is not empty do
        node = queue.poll()
        if node.type is not primitive and is not special then
            fields.clear()
            class = node.type
            inheritedIndex = loadAllFields(fields, class)
            for each field in fields do
                newNode = initializeItem(node, inheritedIndex)
                if not visited.contains(field.type) then
                    queue.add(newNode)
                    visited.add(field.type)
                end if
            end for
        end if
    end while
end procedure
```

### 6.3.1 Nalezení všech atributů třídy

Nalezení všech atributů třídy je provedeno v metodě `loadAllFields()`. Tato metoda získá všechny deklarované atributy třídy `clazz` a uloží je do seznamu `fields`, poté změní hodnotu `clazz` na její rodičovskou třídu a opět uloží všechny její atributy do seznamu `fields`. Toto pokračuje, dokud není rodičovská třída typu `Object`, nebo nemá hodnotu `null`. Tato metoda vrací index, od kterého začínají v seznamu zděděné atributy. Její pseudokód lze nalézt v algoritmu 3.

**Algorithm 3** Algoritmus pro nalezení všech atributů třídy

```
inheritedIndex  ▷ Index, od kterého začínají zděděné atributy v seznamu
procedure LOADALLFIELDS(class, fields)
    fields.addAll(class.declaredFields)
    inheritedIndex = fields.size
    class = class.superclass
    while class is not null and class is not Object.class do
        fields.addAll(class.declaredFields)
        class = class.superclass
    end while
    return inheritedIndex
end procedure
```

## 6.4 Přiřazování hodnot atributů

K přiřazování hodnot atributů dochází výhradně ve třídě `TreeData`. Pro přiřazení hodnot atributů určité instance musí být tato instance vytvořena. První možností vytvoření instance je její vytvoření uživatelem. Druhou je pak automatické vytvoření instance, které je popsáno v následující podkapitole.

### 6.4.1 Automatické vytvoření instancí

Automatické vytváření instancí probíhá podobným způsobem jako vytváření instancí z hodnot zadaných uživatelem. Nejprve je tedy nutné vytvořit pole s hodnotami parametrů konstruktoru. Toto pole je typu `Object` a nazývá se `args`.

Algoritmus automatického vytváření instancí byl implementován v metodě `createDefaultArguments()` a nachází se ve třídě `TreeWindow`. Jeho

návratovou hodnotou je výše zmíněné pole `args`. Algoritmus obsahuje cyklus, který prochází všechny zadané parametry konstruktoru. Konstruktor, který byl zvolen pro automatické vytvoření instance, je vždy konstruktor nacházející se na prvním místě v poli konstruktorů získaném pomocí metody `getConstructors()` třídy `Class`. V případě, že je aktuální parametr primitivního typu, dochází ke zjištění jména typu parametru. Pokud je typem parametru typ `double`, nebo typ `float`, je hodnota pro tento parametr nastavena na *0*. U dalších primitivních číselných parametrů je hodnota také nastavena na *0*, ale zde musí dojít k přetypování na daný číselný typ. Dalším primitivním typem je typ `char`, jehož hodnota je nastavena na `'a'`. Poslední primitivní typ, `boolean`, je nastaven na hodnotu `true`. Všechny ostatní (komplexní) typy parametrů jsou nastaveny na `null`, což opět může způsobit problémy při vytváření instancí.

Výše popsaný algoritmus je poté použit v metodě pro přiřazování hodnot atributů v případě, že nebyla vytvořena instance daného atributu.

### 6.4.2 Přiřazování

Pro přiřazování hodnot atributů byl použit algoritmus, který byl zmíněn v kapitole 5.7. Implementován je v metodě `assignVariables()` ve třídě `TreeData` s mírnými úpravami pro použití při vytváření více objektů. Ke své funkčnosti potřebuje vytvořenou stromovou strukturu, pomocí které jednotlivé atributy přiřazuje. Stromová struktura je vždy dodána instanci třídy `TreeWindow` při jejím vytvoření, která jí předá instanci třídy `TreeData`. Reprezentuje ji třídní proměnná `startItem` třídy `ItemData`. Tato instance obsahuje informace o třídě, která je kořenovým uzlem, a již vytvořené instance pro tuto třídu (pokud jich je více) a hlavně všechny její atributy, tedy další větve stromu. Parametr `instance` této metody je instance kořenové třídy z instance `startItem`. Algoritmus začne prohledáváním všech uzlů, které vedou z aktuálního uzlu.

Pokud je typ třídy, kterou potomek reprezentuje, primitivní, dojde k získání řetězce textového pole, na které odkazuje uzel potomka a jeho převedení na příslušný typ. I zde dochází ke generování náhodných hodnot primitivních číselných typů při vytváření více instancí, pokud tak uživatel zvolí, ovšem první instance je vždy nastavena podle zadaných hodnot. To samé platí pro atributy výčtového typu, typu `String` a primitivního typu `boolean`. Získané hodnoty jsou poté přiřazeny atributům aktuální instance pomocí toho, že každá instance uzlu v sobě uchovává instanci třídy `Field`, která reprezentuje atribut rodičovského uzlu (třídy). Nad instancí atributu je poté zavolána příslušná metoda (`setInt()`, ...).

V případě atributů komplexních typů dochází k uložení vytvořených instancí přímo do příslušné instance třídy `ItemData` a proto nemusí být použito žádné pole s objekty. Pokud jsou atributy typu pole, `ArrayList` nebo `HashMap`, dojde k přiřazení hodnoty atributu aktuální instance pomocí metody `set()`. Pakliže nebyla vytvořena žádná instance těchto typů (není uložena v uzlu), není atribut nastaven na nic a zůstane tedy `null`.

**Algorithm 4** Algoritmus pro nastavování atributů

```
nextNodes                          ▷ Seznam všech následujících uzlů
type                               ▷ Typ atributu který uzel reprezentuje
instance                           ▷ Instance komplexního typu atributu
field                              ▷ Atribut rodičovské třídy
UINode                             ▷ GUI uzly použité pro zadávání dat uživatelem
procedure ASSIGNVARIABLES(node)
    for nextNode from nextNodes of node do
        if nextNode.type is primitive then
            value = nextNode.UINode.value
            nextNode.field.set(value)
        else if nextNode.type is ArrayList or HashMap or array then
            nextNode.field.set(nextNode.instance)
        else
            if nextNode.instance is not created then
                instance = createInstance()
                nextNode.field.set(instance)
            else
                nextNode.field.set(nextNode.instance)
            end if
            ASSIGNVARIABLES(nextNode)
        end if
    end for
end procedure
```

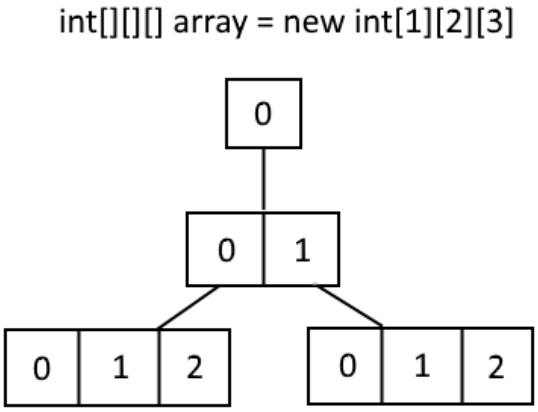
Posledním možným případem jsou ostatní komplexní typy atributů. V následujícím popisu nebudu brát v úvahu možnosti nastavení uživatelem, tedy zákaz větvení, nebo explicitní nastavení na hodnotu `null`. V případě, že instance byla uživatelem vytvořena, dojde k jejímu přiřazení k atributu aktuální instance. Následuje rekurzivní volání popisované metody s tím, že nastavovaná instance bude ta, která se právě přiřadila atributu a a startovní uzel bude ten, do kterého instance patří. Druhou možností je, že uživatel nevytvořil instanci pro přiřazení atributu. Dojde tedy k vyhledání konstruk-

torů, zvolení prvního konstruktoru, a vytvoření pole s hodnotami parametrů způsobem popsaným v předchozí kapitole. Zde se může stát, že nebudou nalezeny žádné veřejné konstruktory a tím pádem nebude možné instanci vytvořit. Po vytvoření instance se opět nastaví tato instance jako hodnota atributu aktuální instance a opět dojde k rekurzivnímu zavolání metody.

Vše lze popsat zjednodušeným pseudokódem v algoritmu 4, který nebere v potaz ovlivňování větvení a nastavení na `null` uživatelem a také možnost vytváření a nastavování více instancí.

## 6.5 Ukládání zadaných hodnot do pole

Pro ukládání zadaných hodnot do pole slouží metoda `setFields()` společně s metodou `assignValues()`. Obě tyto metody jsou implementovány ve třídě `ArrayData`. Jak již bylo dříve zmíněno, vícerozměrná pole lze reprezentovat pomocí stromové struktury. To umožňuje prohledávání tohoto stromu pomocí již několikrát použitého algoritmu BFS. Hodnoty, které budou do pole uloženy, jsou opět reprezentovány pomocí grafických prvků, stejným způsobem, jako při vytváření instancí podle hodnot zadaných uživatelem a jsou tedy uloženy v seznamu. Struktura vícerozměrného pole je zobrazena na obrázku 6.2.




Obrázek 6.2: Stromová reprezentace polí vícerozměrného pole.

V následujícím textu bude popsáno ukládání zadaných hodnot pouze do jediného pole. Dále se předpokládá, že pole již bylo vytvořeno. Třída `ArrayData` obsahuje atribut `fieldCount`, který udává délku koncových polí (těch co jsou v listech stromu) a také atribut `subArrayCount` udávající celkový počet těchto polí.

V první zmíněné metodě dochází k získání referencí na poslední (listová) pole. Pokud je pole jednorozměrné, je situace jednoduchá. Konečné pole je pole kořenové, atribut `subArrayCount` bude tedy mít hodnotu *1* a atribut `fieldCount` bude mít hodnotu danou délkou pole. Pro uložení hodnot tedy dojde k zavolání druhé zmíněné metody. Tato metoda přiřazuje hodnoty podle typu, na který je pole inicializováno v cyklu, který běží od *0* do `fieldCount` a prochází hodnoty v seznamu prvků, podle kterých přiřazuje hodnoty (v případě nastavování více polí mohou být hodnoty opět změněné podle zadaného intervalu) do pole na jednotlivé pozice dané proměnnou v cyklu.

V případě, že je pole vícerozměrné, je nutné projít strom a získat listová pole. Nejprve dojde k inicializaci proměnných typu `int`. Proměnná `dimension` označuje z jaké dimenze aktuální pole pochází. Druhá proměnná, `nodeIndex`, pak informaci o tom, kolik polí z aktuální dimenze již bylo rozvětveno. Obě proměnné jsou nastaveny na *0*. K práci algoritmu je opět potřeba použít frontu, jelikož se jedná o BFS. Zpočátku jsou do fronty uložena všechna pole z první (nulté) dimenze. Dále následuje zvýšení hodnoty proměnné `dimension` o *1*. Následně začne vybírání dat z fronty, ale pouze v případě, že aktuální dimenze je menší, než celkový počet dimenzí, protože nechceme z fronty vybírat listová pole. Při vybrání prvku z fronty dojde k jeho rozvětvení a přidání jeho potomků do fronty, načež následuje zvýšení hodnoty `nodeIndex` o *1*. Pokud se hodnota `nodeIndex` rovná délce předchozí dimenze, dojde k zvýšení hodnoty `dimension` o *1* a nastavení hodnoty `nodeIndex` na *0*. Po ukončení tohoto procesu zůstanou ve frontě pouze listová pole.

Na konci celého postupu dojde k nastavení proměnné `nodeIndex` na *0* a postupnému vybírání koncových polí z fronty. Pro nastavení hodnot pole je opět zavolána metoda `assignValues()`, která uloží hodnoty do pole a vrátí index v seznamu prvků, od kterého začínají prvky s hodnotami pro další koncová pole, na který je nastavena proměnná `nodeIndex`. Celý postup lze popsat pseudokódem v algoritmu 5.

## 6.6 Další použité postupy

### 6.6.1 Nalezení tříd v projektu

Nalezení tříd v projektu (balíku) provádějí metody implementované ve třídě `ClassFinder` a jsou jimi `getClasses()`, která je veřejná, a `findClasses()`. Vstupem metody je název kořenového balíku, ve kterém poté metoda najde všechny třídy. Zároveň najde všechny třídy ve všech podbalících. Ke své

**Algorithm 5** Algoritmus pro získání listových polí a uložení dat do pole

```
dims    ▷ Pole s velikostmi dimenzí pole, jeho velikost udává celkový počet dimenzí
procedure SETFIELDS(array)
    queue
    dimension = 0
    nodeIndex = 0
    for i = 0; i < dims[dimension]; i++ do
        queue.add(get(array, i))    ▷ Přidá podpole z array na pozici i do fronty
    end for
    dimension++
    while queue is not empty and dimension < dims.length - 1 do
        currArray = queue.poll()
        for i = 0; i < dims[dimension]; i++ do
            queue.add(get(currArray, i))    ▷ Přidá podpole z currArray na pozici i do fronty
        end for
        nodeIndex++
        if nodeIndex >= dims[dimension - 1] then
            dimension++
            nodeIndex = 0
        end if
    end while
    nodeIndex = 0
                                ▷ Fronta teď obsahuje už jen listová pole
    while queue is not empty do
        currArray = queue.poll()
        nodeIndex = assignValues(currArray, nodeIndex)
    end while
end procedure
```

funkčnosti používá práci s `ClassLoaderem`. Jejím výstupem je pole všech nalezených tříd. Implementace byla převzata a mírně upravena a lze jí nalézt zde [35].

### 6.6.2 Inicializace JavaFX Toolkitu

V obvyklých JavaFX aplikacích je inicializace JavaFX Toolkitu prováděna při volání metody `launch()`, kde třída, která představuje otevírané okno,

musí překrývat metodu `start()`, která je zděděná ze třídy `Application`. Při volání metody `launch()` je ale obtížné získat instanci spouštěné třídy a hlavně lze tuto metodu zavolat jen jednou v průběhu programu, což je v případě použití naší knihovny za účelem testování nemožné, jelikož může docházet k několikanásobnému vytváření instancí pro testování. Tento způsob byl tedy zvolen jako nevhodný a nahrazen druhým způsobem. Tímto způsobem bylo použití třídy `PlatformImpl`, která obsahuje metodu pro inicializaci a poté je možné otevřít okno voláním metody `runLater()`. I tento způsob ale není příliš vhodný, jelikož třída `PlatformImpl` je vnitřní třída implementace JavaFX. Nakonec bylo implementováno řešení inicializace JavaFX Toolkitu pomocí vytvoření instance třídy `JFXPanel` v konstruktoru třídy `TestingTool`. S touto instancí se dále nic neprovádí, ale její vytvoření provede právě inicializaci JavaFX Toolkitu. Otevření okna se poté provede voláním `runLater()` třídy `Platform`, která již není vnitřní třídou. Je také nutné zavolat metodu `setImplicitExit()` třídy `Platform` s hodnotou `false`. Tím se zabrání ukončování JavaFX.

# 7 Testování

## 7.1 Testování implementace

Implementace metod a tříd byla otestována jednotkovými testy pomocí frameworku *JUnit*. Třídy a metody vyžadující interakci s uživatelem, byly otestovány pomocí frameworku *TestFX* a jednotkových testů zároveň. Framework TestFX umožňuje vytváření oken v testu a jejich zobrazování. K zobrazení oken stačí ve třídě, která reprezentuje testovací příklad, zdědit třídu `ApplicationTest` a poté překrýt metodu `start()`, do níž lze vložit kód z JavaFX. Tato metoda se spouští před každou testovací metodou. V případě složitějších testů ji tedy lze použít jen jednou pro celou testovací třídu. Dále umožňuje měnit hodnoty grafických prvků, jako jsou například textová pole, nebo používat tlačítka a podobně, pomocí jednotlivých metod implementovaných v TestFX frameworku. Pro toto je nutné nastavit každému grafickému prvku určitý identifikátor (*ID*).

Pro testování vytvářené knihovny ale tento způsob použit nebyl, protože během psaní kódu nebyly jednotlivým grafickým prvkům přidávány jednotlivé identifikátory a jejich doplnění by bylo časově náročné. V našem případě lze toto testování považovat za testování pomocí scénáře. Pro metody ve třídách GUI bylo provedeno nepřímé otestování. To znamená, že například v případě metody, která se stará o provedení akce po stisku tlačítka pro vytvoření instance, se otestuje, zda je výsledná hodnota atributu vytvořené instance nastavená na stejnou hodnotu, kterou tester zadal do příslušného pole pro parametr konstruktoru.

Všechny vytvořené testy, jak pro JUnit, tak pro TestFX, jsou uloženy v balíku `tests`, který se dělí na podbalíky vytvořené podle balíků hlavního balíku a tyto balíky obsahují další podbalíky v případě více testů jedné třídy (nebo okna).

Dále testovací balík obsahuje podbalík s třídami, které slouží pro testování. Tyto třídy obsahují různé typy atributů a implementace konstruktorů. Jednou z tříd je také třída reprezentující výčtový typ.

### 7.1.1 Jednotkové testy

Klasické jednotkové testy byly vytvořeny pro třídy z balíků `app` a `support`. Muselo zde ale také dojít k použití frameworku TestFX v případě, že bylo nutné vytvořit třídám z balíku `support` některé grafické prvky, jako jsou

textová pole nebo tlačítka a třídě `TreeData` vytvořit stromovou reprezentaci pomocí třídy `ItemData`, která také obsahuje grafické prvky. Pro vytvoření těchto grafických prvků je zapotřebí mít inicializovaný JavaFX Toolkit, což provede právě TestFX. Výjimkou je v tomto testování třída `TestingTool`, která je testována jak pomocí jednotkových testů, tak pomocí jednoduchého scénáře.

Jednotkové testy byly pak vytvářeny pro každou veřejnou metodu z testovaných tříd s tím, že *getry* a *setry* testovány nebyly z toho důvodu, že jejich implementace byla provedena vygenerováním pomocí použitého vývojového prostředí.

Pro zjištění pokrytí byl použit plugin *Code Coverage* zabudovaný do vývojového prostředí *IntelliJ IDEA*. Jeho výhodou je, že je součástí instalace samotného vývojového prostředí a tak není nutná žádná další instalace. Stačilo pouze nastavit způsob zjišťování pokrytí. Při spuštění testů se sledováním pokrytí bylo dosaženo 100% pokrytí tříd, 79% pokrytí metod, 93% pokrytí řádek kódu a 81% pokrytí větví v balíku `app`. V balíku `support` bylo dosaženo 100% pokrytí tříd, 86% pokrytí metod, 94% pokrytí řádek kódu a 100% pokrytí větví.

| Element | Class, % | Method, % | Line, % | Branch, % |
|---|---|---|---|---|
| ArrayData | 100% (1/1) | 80% (28/35) | 96% (171/178) | 80% (79/98) |
| ArrayListData | 100% (1/1) | 77% (14/18) | 92% (50/54) | 100% (10/10) |
| ClassFinder | 100% (1/1) | 100% (7/7) | 95% (83/87) | 87% (28/32) |
| CreateInstanceData | 100% (1/1) | 89% (25/28) | 96% (92/95) | 80% (32/40) |
| HashMapData | 100% (1/1) | 80% (46/57) | 91% (118/129) | 100% (18/18) |
| TestingTool | 100% (1/1) | 80% (4/5) | 79% (23/29) | 100% (2/2) |
| TreeData | 100% (1/1) | 55% (10/18) | 93% (108/116) | 74% (53/71) |

Obrázek 7.1: Výsledné pokrytí testy balíku app.

| Element | Class, % | Method, % | Line, % | Branch, % |
|---|---|---|---|---|
| Constants | 100% (1/1) | 100% (7/7) | 100% (27/27) | 100% (0/0) |
| DataConverter | 100% (1/1) | 100% (1/1) | 100% (13/13) | 100% (3/3) |
| ItemData | 100% (1/1) | 80% (17/21) | 85% (24/28) | 100% (1/1) |

Obrázek 7.2: Výsledné pokrytí testy balíku support.

Pokrytí tříd znamená, že každá třída byla v testu alespoň jednou použita, pokrytí metod pak udává procento metod zavolaných v testech. Dále pak pokrytí řádek udává, kolik procent řádek kódu bylo při běhu testu programem navštíveno. Procento pokrytých větví pak udává, kolik větví programu bylo navštíveno. Příkladem větve je například if-else blok. Nízké číslo pokrytí metod je dáno netestováním getrů a setrů z důvodů výše popsaných. Poměrně

nízké pokrytí řádek kódu a větví kódu je opět dáno netestováním getrů a setrů. Výsledná pokrytí testy balíků `app` a `support` jsou na obrázcích 7.1 a 7.2. Tyto obrázky byly pořízeny z vývojového prostředí.

### 7.1.2 Testování grafického uživatelského rozhraní

Testování uživatelského rozhraní bylo provedeno podle zadaných scénářů a s pomocí TestFX frameworku. V tomto případě bylo každé okno reprezentováno jako jeden celek. Vždy tedy došlo k vytvoření instance třídy reprezentující určité okno a dále k otevření okna v nastaveném základním tvaru s vloženými daty o typech, se kterými bude okno v testu pracovat. Poté vždy došlo k nastavení hodnot vyžadovaných okny podle zadaných scénářů a k jejich potvrzení. Například při testování okna pro vytváření instancí došlo k zadání hodnot jednotlivých parametrů a k vytvoření instance stiskem příslušného tlačítka. Vše bylo poté potvrzeno a okno zavřeno. Z instance okna pak došlo k získání vytvořené instance pomocí příslušných getrů. U této získané instance došlo k použití jednotlivých porovnávacích metod z frameworku JUnit. V jedné testovací metodě bylo použito několik těchto porovnávacích metod, protože vždy docházelo k nastavování více hodnot.

Pokrytí kódu testy bylo opět možné zjistit pomocí pluginu použitého u jednotkových testů, ale jelikož se jedná o testování grafického uživatelského rozhraní, nebude toto pokrytí v textu zmíněno, protože hodnoty pokrytí pro třídy grafického uživatelského rozhraní nemají velkou vypovídající hodnotu vzhledem k tomu, že v testovacích metodách nedocházelo ke kontrolám například správného umístění grafických prvků.

Nevýhodou testování pomocí TestFX bylo, že dochází k vypršení času pro probíhající testovací metodu. Hodnoty do grafických prvků oken je tedy nutné zadávat urychleně. Řešení tohoto problému nebylo nalezeno, ale důvodem nejspíše bude, že TestFX framework neočekává, že tester bude hodnoty zadávat ručně, ale pomocí příkazů napsaných v kódové podobě s použitím jednotlivých identifikátorů.

## 7.2 Testování celkové funkcionality

Testování celkové funkcionality bylo provedeno na uměle vytvořeném projektu, který je k dispozici na přiloženém CD. V tomto umělém projektu byly použity různé typy atributů, včetně výčtových typů, vnitřních tříd, nebo atributů typovaných na rozhraní, například `List`. Pro dvě třídy umělého projektu byly vytvořeny JUnit testy, jelikož je aplikace určena především pro jednotkové testování. V tomto případě již nebylo nutné přidávat

do testovacího projektu framework TestFX. Stačil tedy pouze JUnit, jelikož v tomto způsobu testování je vlastně vytvořená knihovna reálně používána. Pro vytvoření objektu zvolené třídy je tedy použita vytvořená knihovna. Tento objekt je vytvořen podle předem definovaných pravidel. Stejně tak jsou podle pravidel nastaveny jeho atributy, které mohou být také komplexní a tak mohou být také vytvořeny a nastaveny knihovnou. Některé atributy také nastaveny nejsou, nebo je u nich použita možnost zabránění nastavování jejich atributů.

| **Projekt** | **IDE** |
|---|---|
| TestProject | IntelliJ IDEA/Eclipse |
| OKS - oks-prj-03 | Eclipse |
| OKS - oks-prj-02 | IntelliJ IDEA |
| UZI - Piškvorky | IntelliJ IDEA |
| UIR - Automatická detekce událostí | IntelliJ IDEA |
| UPS - Online Pexeso | IntelliJ IDEA |
| UUR - FileManager | IntelliJ IDEA/Eclipse |
| UPG - Hra dělostřelec | Eclipse |
| PRO - FindingSCC | Eclipse |

Tabulka 7.1: Tabulka, zobrazující ve kterých vývojových prostředích a ve kterých projektech byla knihovna testována.

V prvním testu je objekt vytvářen při spuštění celého testu. U metody, ve které je vytvářen, je tedy použita anotace `BeforeClass`. Ve druhém testu je v metodě pro vytváření objektu použita anotace `Before`, to znamená, že objekt bude vytvářen vždy před spuštěním jednoho testu (jedné testovací metody). V testovacích metodách jsou poté testovány jednotlivé hodnoty atributů vytvořeného objektu, nebo více vytvořených objektů. U nich se ověřuje, zda jsou nastaveny na očekávané hodnoty podle zvoleného scénáře. Scénáře, podle kterých lze provést tyto testy, je možné nalézt v příloze C.

Dále bylo provedeno testování na dalších projektech vytvořených pro různé předměty vyučované na univerzitě. Mezi těmito projekty jsou projekty z předmětů OKS, UIR nebo UZI. V tabulce 7.1 lze vidět seznam testovaných projektů a informaci o tom, ve kterých vývojových prostředích byla knihovna testována. V projektu *Online Pexeso* nebylo možné vytvořit některé objekty, protože pro svojí funkčnost, kterou prováděly v konstruktoru, potřebovaly připojený soket. Chybová hlášení z funkcionality testovaného projektu ale nejsou zobrazena v aplikaci, ale v konzoli používaného vývojového prostředí. U projektu *FileManager* nebylo možné načíst pole tříd vytvořených v pro-

jektu, protože docházelo k inicializačním problémům s JavaFX, ve které je projekt FileManager vytvářen. Tento problém lze obejít nastavením názvu hlavní knihovny projektu na `""` a poté vytvářet instance pomocí knihovny, jako by všechny třídy v projektu do projektu nepatřily (tedy vždy provádět úplné prohledávání). Také bylo zjištěno, že nelze vytvářet instance z balíku `javafx.beans.property`.

Celková funkcionalita byla testována v systému *Microsoft Windows 10*, ve vývojovém prostředí *IntelliJ IDEA 2019.1*. Dále byl hlavní projekt pro otestování funkcionality vytvořen také ve vývojovém prostředí *Eclipse Oxygen.3* a otestován v něm. U testů se také nebral ohled na to, jaká verze frameworku JUnit byla použita. Použila se vždy ta, kterou vývojové prostředí navrhovalo. Pro otestování na uměle vytvořeném projektu byl ale vždy použit framework JUnit4.

# 8 Omezení knihovny a návrhy na další rozšíření

Knihovna neumožňuje vytváření instancí objektů typovaných na rozhraní, jelikož rozhraní nemají žádné konstruktory. To může způsobit nemožnost vytvoření instancí tříd, které mají parametr typovaný na rozhraní a v konstruktoru je nezbytné jeho zadání. Také není možné vytváření tříd, které neobsahují modifikátor public a zároveň žádný veřejný konstruktor, a také tříd, které mají jen konstruktory protected nebo private.

Někdy také může dojít k potížím při zadání špatných dat do konstruktoru, který s nimi dále provádí další práci. Tyto výjimky nelze zobrazit v aplikaci, ale jsou vypsány do konzole příslušného vývojového prostředí. Běh knihovny ale pokračuje dále. Při nastavování hodnoty atributu s modifikátorem final nedojde k uložení hodnoty v případě, že hodnota tomuto atributu již byla přiřazena v kódu.

Dále může pro uživatele být časově náročnější a méně přehledné vytváření a nastavování objektů se složitou strukturou. To je dáno tím, že některé typy mohou obsahovat velké množství různých atributů, které opět nemusejí být pouze primitivní. Z toho důvodu je výsledná grafová struktura velmi velká a celkový postup vytváření a nastavování může kvůli cyklům probíhat do nekonečna, protože i přes to, že cykly jsou při nastavování atributů omezeny, uživatel může pomocí okna pro vytváření instancí vytvořit instanci pro atribut, který je vlastně na listu stromové reprezentace a pro tento atribut opět otevřít okno pro nastavování atributů. To má za následek, že uživatel může při vytváření objektů postupovat rekurzivně, do hloubky, a tím vytvářet velké množství oken, ve kterých je poté snadné se ztratit.

Dalším omezením může být chybějící speciální podpora pro další běžně používané datové typy (například `LinkedList`). V tomto případě je uživatel nucen nastavovat atributy těchto vytvořených instancí a způsob uložení objektů ručně, stejně jako by se jednalo o jakýkoliv jiný obecný objekt, což při běžném vytváření zmíněného `LinkedListu` obvykle nenastává.

Jedním z omezení také je nemožnost vytváření polí s celkovým součinem dimenzí větším než 500. Při pokusu o vytvoření pole se součinem větším docházelo k poměrně dlouhým prodlevám a v některých případech došlo i k nedostatku paměti na haldě. Prodleva v GUI také nastává při zvolení prohledávání všech tříd, pokud je výsledná struktura moc velká.

Také je zde důležité zmínit, že při vytváření instancí vnitřních tříd, není

ve vytvořeném objektu uložena reference na instanci třídy vnější, lze pouze vytvořit novou instanci vnější třídy, ale pokud by bylo zapotřebí použít už někdy vytvořenou instanci vnější třídy, nebude možné jí tuto instanci přiřadit. V případě statických vnitřních pak nelze vytvořit instanci této třídy, pokud nemá deklarovaný žádný konstruktor. Knihovna také neumožňuje vytvářet pole generických typů, pokud mají být uložena do `ArrayListu`, nebo `HashMapy`.

Dalším rozšířením aplikace může být tedy speciální podpora dalších datových typů, nebo vyřešení vytváření objektů typovaných na rozhraní nebo objektů tříd, které nemají veřejný konstruktor, ale jen konstruktory protected. Dále je možné doplnit správné přiřazovaní referencí na instance vnější třídy u vytvořených instancí vnitřních tříd.

Při automatickém vytváření instancí může algoritmus skončit nezdarem v případě, že první konstruktor (ten, který je zvolen) provádí se zadanými parametry další činnost, ne jen přiřazování hodnot atributům. Toto by mohlo jít zmírnit pomocí postupného zkoušení všech konstruktorů, ale ani to by nemuselo problém plně vyřešit a uživatel by byl v některých případech nucen objekty s těmito konstruktory vytvářet ručně.

Pro použití knihovny je nutné mít nainstalovaný framework JavaFX ve verzi 8 nebo vyšší, který je ale obvykle součástí vývojových prostředí.

# 9 Závěr

V této práci byly popsány vybrané způsoby generování dat pro testování a dále byla implementována knihovna umožňující uživateli vytváření objektů, obvykle určených pro testovací účely. Na základě provedených testů lze usuzovat, že knihovna pracuje správně pro běžné projekty. Pomocí knihovny je tedy možné vytvářet instance většiny tříd, které mají veřejné konstruktory a žádný z jejich parametrů, který je typovaný na rozhraní, není nezbytný pro vytvoření instance.

Knihovna vlastně rozšiřuje postup popsaný v kapitole 3.6 [21] tím, že umožňuje vytvářet objekty s různými konstruktory a také nastavovat hodnoty komplexních atributů. Vytvořená knihovna odpovídá požadavkům zadání. Další možnosti vylepšení jsou popsány v kapitole 8.

# Použité zkratky

**PSO** Particle Swarm Optimization – meta-heuristická metoda používaná v oblasti umělé inteligence

**GUI** Graphical User Interface – rozhraní umožňující ovládání aplikace pomocí grafických prvků

**UML** Unified Modeling Language – grafický jazyk pro vizualizaci programů

**BFS** Breadth-first search – algoritmus pro procházení grafů

**DFS** Depth-first search – algoritmus pro procházení grafů

# Bibliografie


[1] *What is Software Testing? Introduction, Definition, Basics & Types.* URL: https://www.guru99.com/software-testing-introduction-importance.html (cit. 23. 06. 2019).

[2] *Why is Testing Expensive?* URL: https://blog.gurock.com/why-is-testing-expensive/ (cit. 23. 06. 2019).

[3] *Typy testování software (třídění testů).* URL: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/ (cit. 20. 04. 2019).

[4] *What is BLACK Box Testing? Techniques, Example & Types.* URL: https://www.guru99.com/black-box-testing.html (cit. 20. 04. 2019).

[5] *What is WHITE Box Testing? Techniques, Example, Types & Tools.* URL: https://www.guru99.com/white-box-testing.html (cit. 21. 04. 2019).

[6] *What is Grey Box Testing? Techniques, Example.* URL: https://www.guru99.com/grey-box-testing.html (cit. 21. 04. 2019).

[7] *Unit testy v Javě a JUnit.* URL: https://www.itnetwork.cz/java/pokrocile/java-unit-testy-v-junit (cit. 22. 04. 2019).

[8] *Integration Testing: What is, Types, Top Down & Bottom Up Example.* URL: https://www.guru99.com/integration-testing.html (cit. 22. 04. 2019).

[9] *What is Functional Testing? Types & Examples (Complete Tutorial).* URL: https://www.guru99.com/functional-testing.html (cit. 22. 04. 2019).

[10] *What is Regression Testing? Definition, Test Cases (Example).* URL: https://www.guru99.com/regression-testing.html (cit. 23. 04. 2019).

[11] *Manual Testing Tutorial for Beginners: Concepts, Types, Tool.* URL: https://www.guru99.com/manual-testing.html (cit. 23. 04. 2019).

[12] *AUTOMATION TESTING Tutorial: What is, Process, Benefits & Tools.* URL: https://www.guru99.com/automation-testing.html (cit. 23. 04. 2019).

[13] *What is Exploratory Testing? Techniques with Examples.* URL: https://www.guru99.com/exploratory-testing.html (cit. 23. 04. 2019).

[14] Mohammad F.J. KLaib et al. „A Parallel Tree Based Strategy for Test Data Generation and Cost Calculation for Pairwise Combinatorial Interaction Testing“. In: *F. Zavoral et al. (Eds.): NDT 2010, Part II, CCIS 88, pp. 509–522, 2010. © Springer-Verlag Berlin Heidelberg 2010* (2010).

[15] Mohammed I. Younis, Kamal Zuhairi Zamli a Nor Ashidi Mat Isa. „IRPS - An Efficient Test Data Generation Strategy for Pairwise Testing“. In: *I. Lovrek, R.J. Howlett, and L.C. Jain (Eds.): KES 2008, Part I, LNAI 5177, pp. 493–500, 2008. © Springer-Verlag Berlin Heidelberg 2008* (2008).

[16] Jiatong Huo et al. „Genetic Programming for Multi-objective Test Data Generation in Search Based Software Testing“. In: © *Springer International Publishing AG 2017 W. Peng et al. (Eds.): AI 2017, LNAI 10400, pp. 169–181, 2017.* (2017).

[17] V. Mařík, O. Štěpánková a J. Lažanský a kol. *Umělá inteligence 4*. Praha: Academia, 2003.

[18] *Data Flow Testing*. URL: `https://www.tutorialspoint.com/software_testing_dictionary/data_flow_testing.htm` (cit. 23. 04. 2019).

[19] Narmada Nayak a Durga PRASAD MOHAPATR. „Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization“. In: *S. Ranka et al. (Eds.): IC3 2010, Part II, CCIS 95, pp. 1–12, 2010. c © Springer-Verlag Berlin Heidelberg 2010* (2010).

[20] James Kennedy a Russel Eberhart. „Particle swarm optimization“. In: *IEEE International Conference, vol.4; pp. 1942-1948 vol.4; Nov/Dec 1995* (1995).

[21] Furqan Naseer, Shafiq Ur Rehman a Khalid Hussain. „Using Metadata Technique for Component Based Black Box Testing“. In: *2010 6th International Conference on Emerging Technologies (ICET)* (2010).

[22] Jakob Jenkov. *Java Reflection Tutorial*. 2018. URL: `http://tutorials.jenkov.com/java-reflection/index.html` (cit. 25. 04. 2019).

[23] Jakob Jenkov. *Java Reflection - Classes*. 2014. URL: `http://tutorials.jenkov.com/java-reflection/classes.html` (cit. 25. 04. 2019).

[24] Jakob Jenkov. *Java Reflection - Private Fields and Methods*. 2018. URL: `http://tutorials.jenkov.com/java-reflection/private-fields-and-methods.html` (cit. 25. 04. 2019).

[25] Jakob Jenkov. *Java Reflection - Fields*. 2016. URL: `http://tutorials.jenkov.com/java-reflection/fields.html` (cit. 25. 04. 2019).

[26] Jakob Jenkov. *Java Reflection - Constructors*. 2014. URL: http://tutorials.jenkov.com/java-reflection/constructors.html (cit. 25. 04. 2019).

[27] Jakob Jenkov. *Java Reflection - Getters and Setters*. 2014. URL: http://tutorials.jenkov.com/java-reflection/getters-setters.html (cit. 25. 04. 2019).

[28] Jakob Jenkov. *Java Reflection - Methods*. 201. URL: http://tutorials.jenkov.com/java-reflection/methods.html (cit. 25. 04. 2019).

[29] Jakob Jenkov. *Java Reflection - Generics*. 2018. URL: http://tutorials.jenkov.com/java-reflection/generics.html (cit. 25. 04. 2019).

[30] Jakob Jenkov. *Java Reflection - Arrays*. 2014. URL: http://tutorials.jenkov.com/java-reflection/arrays.html (cit. 25. 04. 2019).

[31] Jakob Jenkov. *Java Reflection - Annotations*. 2014. URL: http://tutorials.jenkov.com/java-reflection/annotations.html (cit. 25. 04. 2019).

[32] *mock object*. URL: https://searchsoftwarequality.techtarget.com/definition/mock-object (cit. 25. 04. 2019).

[33] Jakob Jenkov. *Java Reflection - Dynamic Proxies*. 2014. URL: http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html (cit. 25. 04. 2019).

[34] Jakob Jenkov. *Java Reflection - Modules*. 2018. URL: http://tutorials.jenkov.com/java-reflection/modules.html (cit. 25. 04. 2019).

[35] Victor Tatai. *Get All Classes Within A Package*. Lis. 2007. URL: https://dzone.com/articles/get-all-classes-within-package (cit. 09. 06. 2019).

# A Uživatelská dokumentace

## A.1 Spuštění knihovny

Pro spuštění knihovny je nutné vytvořit instanci třídy `TestingTool` s parametrem, který udává název hlavního balíku zvolené aplikace. Poté je možné zavolat jednu z jejích metod. Metoda `createObject(String className)` spustí vytváření jediné instance, která je vrácena jako objekt typu `Object`, který je pro další použití nutné přetypovat. Parametr `className` udává název třídy, jejíž instanci chce uživatel vytvořit. Jméno třídy musí být zadané celé. Pokud se tedy třída jmenuje `ClassA` a leží v balíku `packageB` a ten leží v balíku `packageA`, bude jméno třídy `"packageA.packageB.ClassA"`. Metoda `createObjects(String className, int count)` poté vrací pole instancí zadané třídy, typované na `Object`. Jednotlivé instance tohoto pole je poté vhodné opět přetypovat na požadovaný typ. Parametr `count` zde udává počet požadovaných objektů. Akce pro vytoření jednoho objektu je zobrazena na obrázku A.1.

```
private static TestingTool testingTool = new TestingTool( packageName: "src");
private static ClassB classB;

@BeforeClass
public static void setUpBeforeClass() throws Exception {
    classB = (ClassB) testingTool.createObject( className: "src.ClassB");
}
```

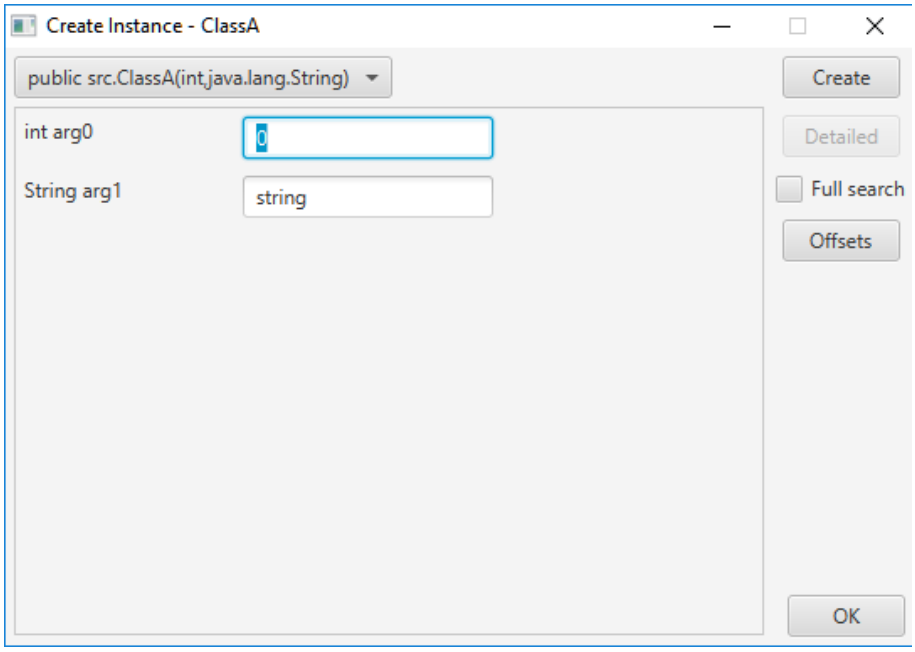
Obrázek A.1: Ukázka použití knihovny.

## A.2 Hlavní okno pro vytváření instancí

Po zavolání jedné z metod se spustí hlavní okno. Toto okno umožňuje zvolení konstruktoru pro vytvoření instancí a vyplnění hodnot parametrů zvoleného konstruktoru. Dále obsahuje tlačítko *Offsets*, které umožňuje nastavení náhodných změn primitivních číselných hodnot u vytvářených instancí, kromě té první, a tlačítko *Detailed*, které je použito pro zobrazení okna s nastavením všech atributů objektu a je možné ho použít až po vytvoření instancí. K tlačítku *Detailed* patří také zaškrtávací box *Full Search*, který umožní při vytváření stromu s atributy prohledávání tříd, které nejsou v projektu, nebo nastavování atributů vytvářené třídy, pokud není v projektu. Tlačítko

*Create* slouží pro vytvoření instancí. Pokud nebyla vyplněna všechna políčka, zobrazí se hlášení a žádná instance nebude vytvořena. Pokud nebyly vytvořeny objekty pro hodnoty komplexních parametrů, zobrazí se hláška s dotazem, zda-li chce uživatel nechat tyto hodnoty nastavené na `null`. Pokud zvolí ne, nedojde k vytvoření instancí, pokud ano, dojde k vytvoření instancí. Tlačítko *OK* poté potvrdí změny. Pokud nebyly uloženy, zobrazí hlášení. Stisknutím tlačítka pro zavření okna dojde k nastavení instancí na `null`. Možný vzhled okna pro vytváření instancí je zobrazen na obrázku A.2.

Pokud třída, ze které se pokoušíme vytvořit instance neobsahuje žádný veřejný konstruktor, zobrazí se hlášení, že instance nelze vytvořit a okno se neotevře.

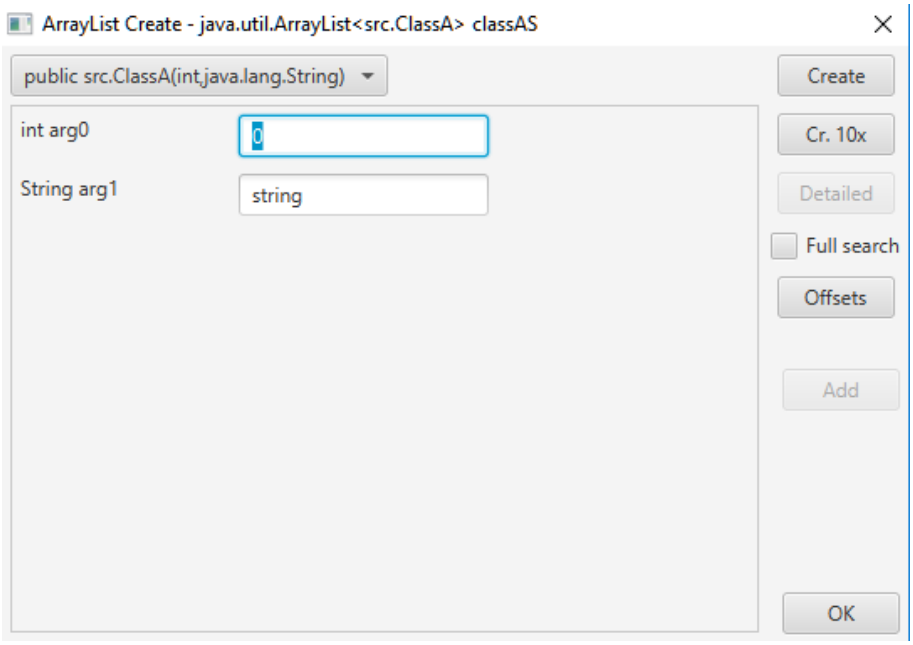



Obrázek A.2: Okno pro vytváření instancí.

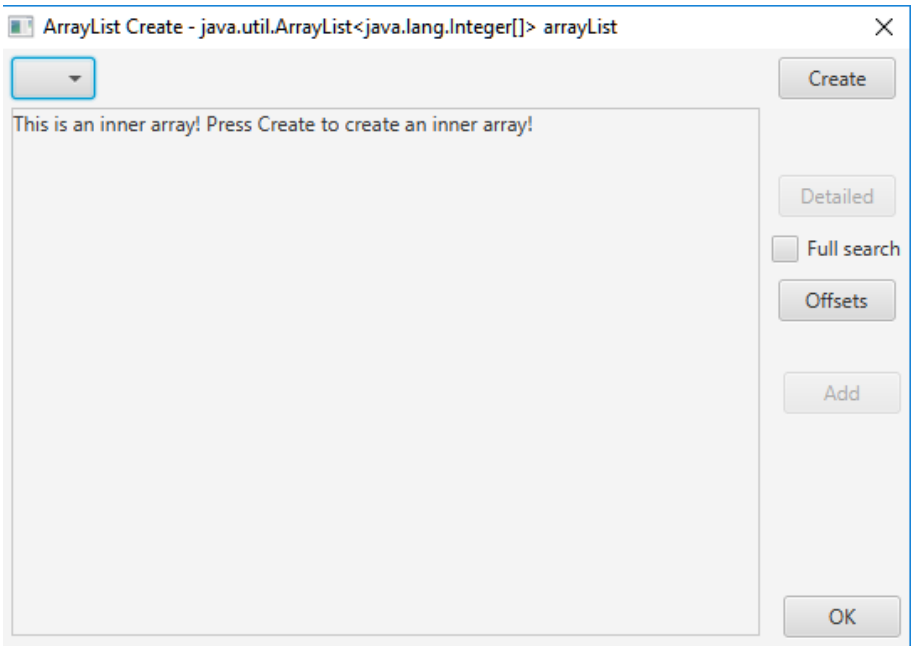
## A.3 Okno pro vytváření ArrayListu

Okno pro vytváření `ArrayListu` obsahuje stejné základní prvky jako okno pro vytváření instancí. Dále obsahuje tlačítko *Cr. 10x*, které umožní vytvoření deseti instancí, které lze pak všechny po skupinách vložit do vytvářených `ArrayListů`. Po vytvoření objektů a nastavení atributů je nutné použít tlačítko *Add*, které provede samotné přidání prvků do *ArrayListů*. Po přidání prvků už nelze dále tyto prvky upravovat. Toto okno je zobrazeno na obrázku A.3.

Pokud je vytvářený `ArrayList` složen z několika vnořených `ArraListů`, nebo `HashMap` a nebo polí, nebude možné zvolit žádný konstruktor, ale pouze vytvořit vnitřní `ArrayList` nebo `HashMapu` nebo pole, vždy ale po jednom, pomocí příslušných oken. To lze vidět na obrázku A.4.




Obrázek A.3: Okno pro vytváření ArrayListů.

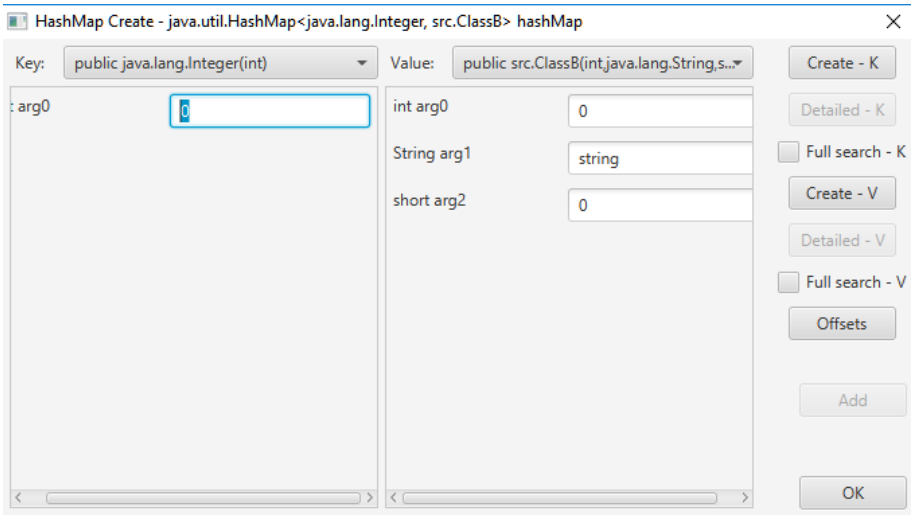



Obrázek A.4: Okno pro vytváření vnitřních ArrayListů.

## A.4 Okno pro HashMapu

Okno pro `HashMapu` je opět řešeno podobně jako okno pro vytváření instancí. Obsahuje možnosti výběru konstruktorů pro klíč i pro hodnotu a zobrazuje prvky pro zadání hodnot parametrů pro oba tyto konstruktory odděleně. Dále obsahuje tlačítka *Create - K* a *Create - V*, která provedou vytvoření instancí klíčů, respektive hodnot, a tlačítka *Detailed - K* a *Detailed - V*, která slouží pro nastavení atributů klíčů a hodnot. Tato tlačítka opět obsahují zaškrtávací boxy pro nastavení prohledávání tříd mimo projekt při nastavování atributů. Tlačítko *Add* pak opět provede přidání dat do `HashMapy` s tím, že data už nelze nijak měnit. Vzhled okna lze vidět na obrázku A.5.

Stejně jako u `ArrayListu` nebude možné volit konstruktor, pokud typ klíče nebo hodnoty bude `ArrayList` nebo `HashMap` nebo pole. Půjde jen vytvořit jejich instance pomocí příslušných oken. Vzhled tohoto přidávání bude podobný jako na obrázku A.4.

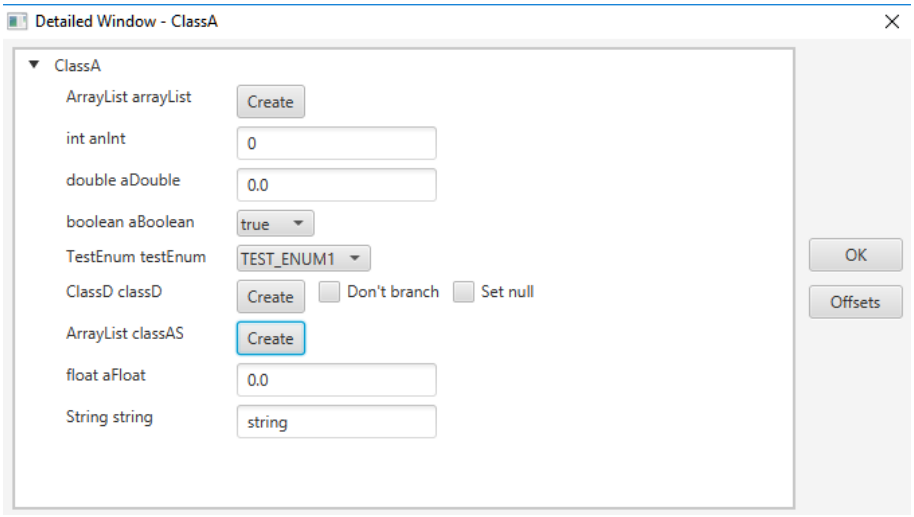



Obrázek A.5: Okno pro vytváření HashMapy.

## A.5 Okno pro nastavování atributů

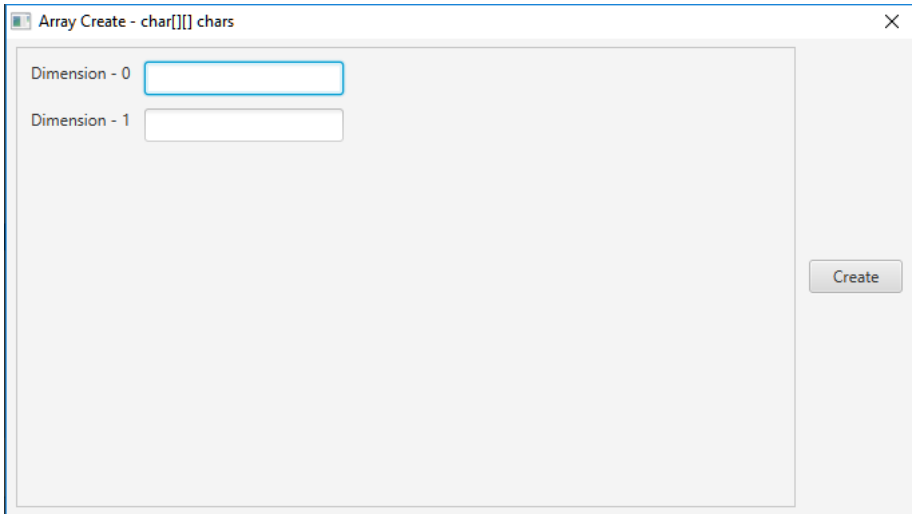
Okno pro nastavování atributů obsahuje stromové zobrazení struktury třídy a jejích atributů. Obsahuje prvky pro zadávání hodnot nebo jejich nastavování, nebo vytváření. Opět je zde tlačítko *Offsets*, které provede změnu hodnot primitivních číselných atributů u dalších nastavovaných instancí. V případě, že uživatel nenastaví hodnotu komplexního atributu, který není typu `ArrayList` nebo `HashMap` a nebo pole, a ani ho nenastaví na `null`,

dojde k automatickému vytvoření instance při přiřazování hodnot. Přiřazování hodnot probíhá po stisku tlačítka *OK*. Pokud v nějakém poli nebyla zadána hodnota, dojde k zobrazení hlášky a atributy nebudou nastaveny. Oproti předchozím oknům ze zde ale nekontroluje vytvoření instance atributu. Toto okno lze vidět na obrázku A.6.




Obrázek A.6: Okno pro nastavování atributů.

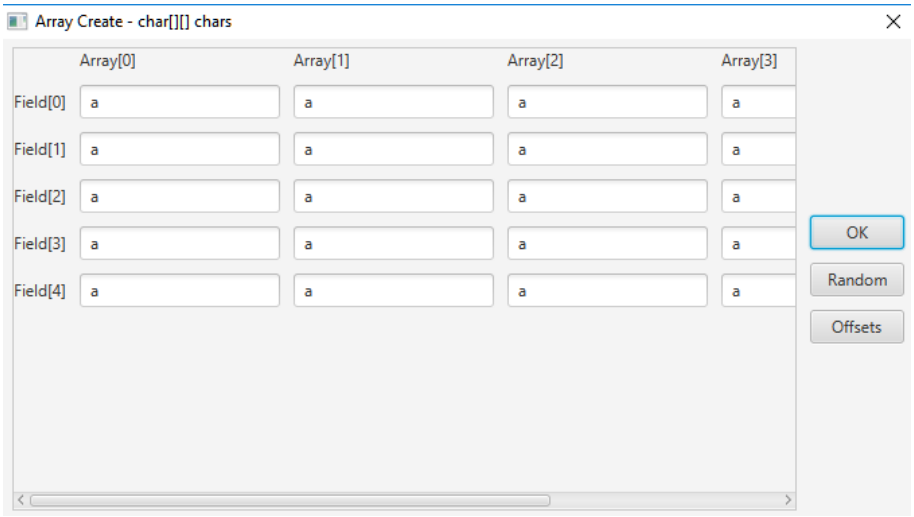
## A.6 Okno pro vytváření polí




Obrázek A.7: Okno pro zadávání velikostí dimenzí pole.

Po otevření okna pro vytváření polí se uživateli zobrazí několik textových políček v závislosti na počtu dimenzí vytvářeného pole. Po zadání velikostí

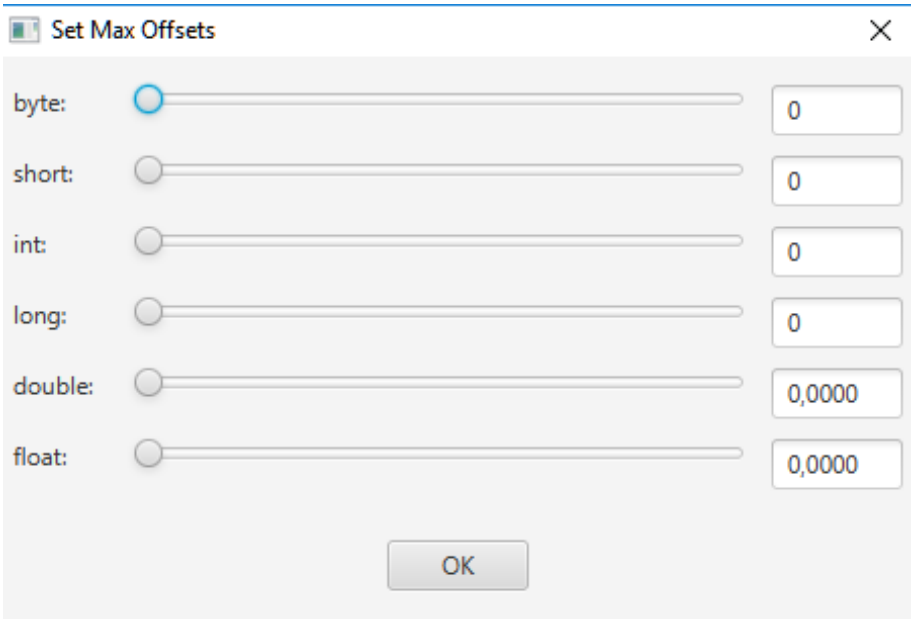
dimenzí se zobrazí mřížková reprezentace pole, kde v horní části jsou vypsány jednotlivé indexy polí a v levé části jednotlivé indexy políček v těchto polích. Pokud je součin dimenzí větší než 500, je zobrazeno hlášení, že pole je moc velké a uživatel musí zmenšit velikosti dimenzí. Dále je k dispozici tlačítko *Offsets*, které slouží ke stejným účelům jako u předchozích oken a tlačítko *Randomize*. Tlačítko *Randomize* umožňuje vygenerování náhodných hodnot primitivních datových typů. Tlačítko *OK* opět slouží k uložení hodnot. Tlačítko pro zavření okna opět neuloží žádné změny a pole nastaví na hodnotu `null`. Vzhledy oken pro práci s poli lze vidět na obrázcích A.7 a A.8.




Obrázek A.8: Okno pro ukládání dat do pole

## A.7 Okno pro nastavování intervalů

V okně pro nastavování intervalů je několik posuvníků, kde každý slouží pro jeden primitivní číselný typ, hodnoty posuvníků jsou vypsány napravo od každého. Podle hodnot nastavených v posuvníku se poté provádí náhodná změna hodnot primitivních číselných parametrů nebo atributů vytvářených nebo nastavovaných tříd, kromě té první (ta bude mít vždy přesně zvolené hodnoty). Pokud je zvolená hodnota pro typ `int` například 2, budou hodnoty všech parametrů nebo atributů typu `int` náhodně změněny v intervalu *+- 2* ze zadané hodnoty pro parametr nebo atribut. Zvolené intervaly nejsou globální, ale platí jen pro aktuální okno. Okno lze vidět na obrázku A.9.




Obrázek A.9: Okno pro nastavování intervalů.

## A.8 Nastavování hodnot parametrů a atributů a hodnot v polích, nemožnost vytvoření instance a generické typy

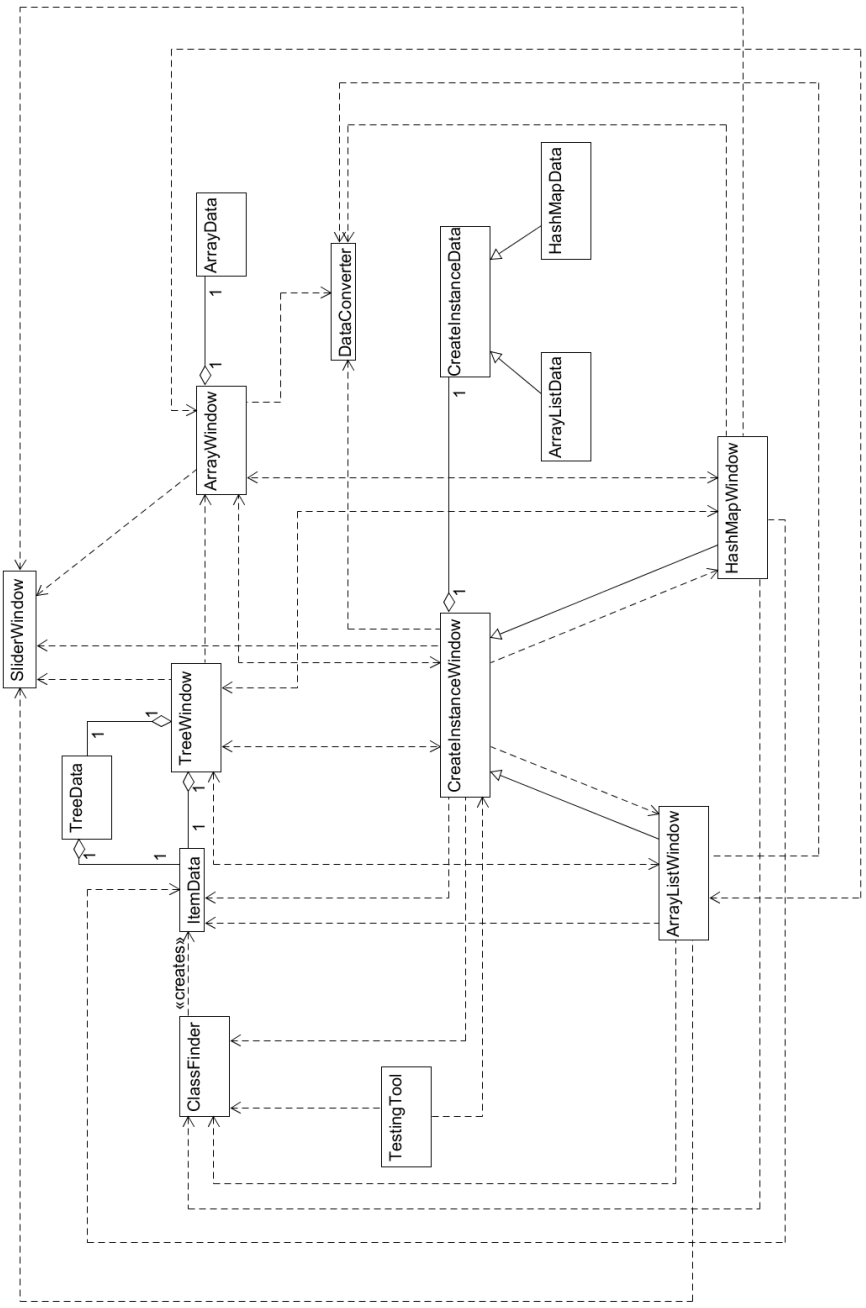
Každý typ parametru nebo atributu má svůj grafický prvek. Pro výčtové typy a typ `boolean` je použit výběrový box, pro další primitivní typy a typ `String` slouží textové pole. Pro ostatní slouží tlačítko. Při vytvoření nějaké hodnoty pomocí tlačítka a potvrzení této hodnoty již nelze tuto hodnotu nastavit na `null` nebo ji měnit, je ji možné pouze přepsat opětovným vytvořením. Nastavit hodnotu na `null` lze pouze u přiřazování atributů, které nejsou pole, `ArrayList` nebo `HashMap`, pomocí zaškrtávacích boxů. Zaškrtávací box se jménem *Set null* slouží pro nastavení hodnoty na `null`, druhý, který se jmenuje *Don't branch*, slouží k nastavení toho, že u daného atributu dojde jen k přiřazení jeho hodnoty, ale již nedochází k nastavení jeho atributů.

Při přejetí myši na název parametru nebo atributu se zobrazí celý název, což je vhodné v případě dlouhých názvů.

Při chybě při vytváření instance ze zobrazí okno s chybovým hlášením a výpisem výjimky, která při vytváření nastala.

Jediné podporované generické typy jsou `ArrayList` a `HashMap`. U ostatních dochází k vytváření instancí bez nastavení genericity.

# B UML diagram tříd




Obrázek B.1: UML diagram tříd. Pomocné třídy byly vynechány.

# C Hlavní testovací scénáře

## C.1 Scénář 1 - `ClassBTest`

Tímto scénářem se testuje celková funkčnost knihovny s využitím všech tříd pro vytvoření jednoho objektu. Postup je následující:

1. Uživatel vytvoří instanci třídy stisknutím tlačítka *Create*, bez nastavování hodnot parametrů
2. Po úspěšném vytvoření instance stiskne tlačítko *Detailed*
3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel nastaví kořenové třídě hodnotu atributu `aShort` na *2* a hodnotu zděděného atributu `anInt` na *1*
5. Uživatel nastaví kořenové třídě hodnotu zděděného atributu `string` na `"test"`
6. Uživatel nechá hodnotu zděděného atributu nastavenou na `TEST_ENUM1`
7. Uživatel vytvoří dvourozměrné pole atributu `chars` o velikosti 3x3, stisknutím na tlačítko *Create* vedle něj
8. Uživatel nastaví hodnoty v poli, aby v prvním podpoli byly `a, b, c`; ve druhém `b, c, d`; a ve třetím `c, d, f`; vše potvrdí stiskem tlačítka *OK*
9. Uživatel nevytvoří instanci atributu `classC`, ale zaškrtne u atributů `socket` a zároveň `list` box *Set null*, dále pak u atributů `innerClassA` a zároveň `innerClassB` box *Don't branch*
10. Uživatel u atributu `classC` vytvoří instanci atributu `arrayLists` a to tak, že:
    (a) Uživatel klikne na tlačítko *Create* u zmíněného atributu
    (b) Uživatel vytvoří vnitřní `ArrayList`
    (c) Uživatel použije tlačítko *Cr. 10x* bez nastavování hodnot parametrů konstruktoru, tím vytvoří *10* instancí
    (d) Uživatel instance přidá do `ArrayListu` stisknutím tlačítka *Add*

(e) Uživatel nastaví hodnotu parametru `arg1` typu `String` na hodnotu `"test111"`, poté použije tlačítko *Create* pro vytvoření jedné instance

(f) Uživatel instanci přidá do `ArrayListu` stisknutím tlačítka *Add*

(g) Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno

(h) V tomto okně uživatel stiskne tlačítko *Add*, pro přidání `ArrayListu` do hlavního `ArrayListu`

(i) Uživatel stiskne tlačítko *Create* pro vytvoření druhého `ArrayListu`

(j) Uživatel nastaví hodnotu parametru `arg1` typu `String` na hodnotu `"test21"`, poté použije tlačítko *Create* pro vytvoření jedné instance

(k) Uživatel instanci přidá do `ArrayListu` stisknutím tlačítka *Add*

(l) Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno

(m) V tomto okně uživatel stiskne tlačítko *Add*, pro přidání `ArrayListu` do hlavního `ArrayListu`

(n) Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno

11. Uživatel u atributu `classC` vytvoří instanci atributu `HashMap hashMap` a to tak, že:

    (a) Uživatel klikne na tlačítko *Create* u zmíněného atributu

    (b) Uživatel nastaví v levé části okna hodnotu parametru `arg0` typu `int` na hodnotu *33*

    (c) Uživatel nastaví hodnotu parametru `arg1` typu `String` na hodnotu `"hash"`

    (d) Uživatel vytvoří instance klíče a hodnoty stisknutím tlačítek *Create - K* a *Create - V*

    (e) Uživatel přidá data do `HashMapy` stisknutím tlačítka *Add*

    (f) Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno

12. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno

13. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

## C.2 Scénář 2 - `ClassATest`

Tento scénář obsahuje několik kratších testů pro otestování funkčnosti vytváření více instancí nebo vytváření instancí tříd mimo projekt.

### C.2.1 Test - `testCreateObject()`

Tento test ověřuje funkčnost vytvoření objektu.

1. Uživatel nastaví hodnotu parametru `arg0` typu `int` na hodnotu *64*
2. Uživatel nastaví hodnotu parametru `arg1` typu `String` na hodnotu `"Create Object"`
3. Uživatel vytvoří instance stisknutím tlačítka *Create*
4. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.2 Test - `testNonDetailedClassD()`

Tento test ověřuje funkčnost nevětvení stromu z uzlů, jež reprezentují třídy nepatřící do hlavního balíku.

1. Uživatel vytvoří instance stisknutím tlačítka *Create*
2. Po úspěšném vytvoření instancí stiskne tlačítko *Detailed*
3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel klikne na tlačítko *Create* u atributu `classD` typu `ClassD`, otevře se okno pro vytváření objektů
5. Uživatel zadá hodnotu parametru `arg0` typu `int` na *65*
6. Uživatel vytvoří instance stisknutím tlačítka *Create*
7. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
8. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
9. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.3 Test - `testDetailedClassD()`

Tento test ověřuje funkčnost větvení stromu z uzlů, jež reprezentují třídy nepatřící do hlavního balíku, po zaškrtnutí boxu *Full Search*.

1. Uživatel vytvoří instance stisknutím tlačítka *Create*
2. Po úspěšném vytvoření instancí zaškrtne box *Full Search*, poté stiskne tlačítko *Detailed*
3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel nastaví hodnotu atributu `anInt` typu `int` v atributu `classD` na hodnotu *65*
5. Uživatel nastaví hodnotu atributu `string` typu `String` v atributu `classD` na hodnotu `"detailed"`
6. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
7. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.4 Test - `testBooleanAndEnum()`

Tento test ověřuje funkčnost nastavování hodnot výčtových typů a typu boolean.

1. Uživatel vytvoří instance stisknutím tlačítka *Create*
2. Po úspěšném vytvoření instancí stiskne tlačítko *Detailed*
3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel nastaví hodnotu atributu `aBoolean` typu `boolean` na hodnotu `false`
5. Uživatel nastaví hodnotu atributu `testEnum` typu `TestEnum` na hodnotu `TEST_ENUM3`
6. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
7. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.5 Test - `testOffsets()`

Tento test ověřuje funkčnost nastavování intervalů pro primitivní číselné datové typy pro vygenerování hodnot podle zadaných intervalů při vytváření více instancí.

1. Uživatel vytvoří instance stisknutím tlačítka *Create*
2. Po úspěšném vytvoření instancí stiskne tlačítko *Detailed*
3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel nastaví hodnotu atributu `aniInt` typu `int` na hodnotu *10*
5. Uživatel nastaví hodnotu atributu `aDouble` typu `double` na hodnotu 10.0
6. Uživatel stiskne tlačítko *Offsets*
7. Otevře se okno pro nastavování intervalů změn primitivních číselných atributů
8. Uživate posune posuvník pro typ `int` tak, aby ukazoval hodnotu *2*
9. Uživate posune posuvník pro typ `double` tak, aby ukazoval hodnotu *2.0*
10. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
11. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
12. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.6 Test - `testAddTen()`

Tímto testem se testuje, že správně proběhne vytvoření dvou `ArrayListů` pro dvě vytvářené instance a poté přidání deseti prvků do každého z `ArrayListů` a to, že se pokaždé jedná o jinou instanci.

1. Uživatel vytvoří instance stisknutím tlačítka *Create*
2. Po úspěšném vytvoření instancí stiskne tlačítko *Detailed*

3. Otevře se okno pro nastavování hodnot atributů
4. Uživatel vytvoří instance atributu `classAS` typu `ArrayList` stisknutím na tlačítko *Create* u příslušného atributu
5. Uživatel stiskne tlačítko *Cr. 10x* pro vytvoření *10* instancí pro přidání do polí (celkem je to tedy *20* instancí)
6. Uživatel stiskne tlačítko *Add* pro přidání instancí do `ArrayListů`
7. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
8. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se vrátí na předchozí okno
9. Uživatel vše potvrdí stiskem tlačítka *OK*, aplikace se zavře a proběhne vyhodnocení testu

### C.2.7 Test - `testAnything()`

Tuto testovací metodu lze spustit pro osobní otestování funkčnosti grafického uživatelského rozhraní.