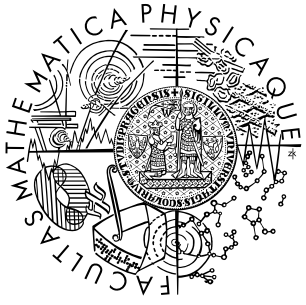
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Martin Babka

Automatizace hierarchických experimentů

Katedra softwarového inženýrství

Vedúci bakalárskej práce: doc. RNDr. Tomáš Skopal, Ph.D.

Študijný program: Informatika, Obecná informatika

2008

Chcem sa poďakovať vedúcemu tejto práce doc. RNDr. Tomášovi Skopalovi, Ph.D. za čas, ktorý mi venoval a nápady, ktorými značne obohatil túto prácu.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 1. 8. 2008 Martin Babka

Obsah

1 Úvod **6**

2 Hierarchický experiment **7**

2.1 Motivácia 7

2.2 Hierarchický experiment 8

2.3 Definícia 9

2.4 Predstava, možné využitie a obmedzenia 10

3 Používateľská dokumentácia **11**

3.1 Rozdelenie programu na moduly 11

3.2 C++ knižnica 12

3.3 Program pre tvorbu experimentov 12

3.4 Inštalácia, požiadavky a spustenie programu 13

3.5 Základné použitie 13

3.6 Nový experiment, práca s experimentom 14

3.7 Spájanie experimentov 16

3.8 Meno a spoločné vlastnosti experimentov 16

3.9 Nastavenia diagramu 17

3.10 Overenie správnosti experimentu 17

3.11 Práca s hierarchickým experimentom 18

3.12 Generovanie výstupného kódu 19

3.13 Nastavenie programu 21

4 Popis jednotlivých experimentov **22**

4.1 Premenné experimentov 22

4.2 Experiment „Composite“ 25

4.3 Experiment „Loop“ 26

4.4 Experiment „Collector“ 26

4.5 Experiment „Object creator“ 28

4.6 Experiment „Method caller“ 28

4.7 Experiment „Code block“ 29

4.8 Experiment „Data collector“ . . . 30
4.9 Experiment „Directory lister“ . . . 32
4.10 Experiment „Line enumerator“ . . . 33
4.11 Experimenty typu „Exponential loop“ . . . 33

5 Spracovanie výstupných dát **35**
5.1 Definovanie prezentovaných dát . . . 35
5.2 Výstupy typu „*Table*“ . . . 37
5.3 Výstup „*Graph 2D*“ . . . 37

6 Implementácia **39**
6.1 Knižnica *Experiments* . . . 39
6.2 Uchovanie generovanej hodnoty . . . 40
6.3 Kontrola typu hodnoty . . . 41
6.4 Implementácia uchovania hodnoty a kontroly typu . . . 42
6.5 Ukladanie výstupných veličín . . . 44
6.6 Experiment knižnice *Experiments* . . . 47
6.7 Definícia experimentu . . . 48
6.8 Program *ExperimentsCreator* . . . 50
6.9 Experiment programu *ExperimentsCreator* . . . 50
6.10 Hierarchický experiment, „pracovná plocha“ . . . 52
6.11 Inicializácia knižnice . . . 54
6.12 Grafické rozhranie . . . 55
6.13 Spracovanie výstupu . . . 56

7 Vývoj experimentu **57**
7.1 Experiment knižnice *Experiments* . . . 57
7.2 Integrácia experimentu . . . 59
7.3 Experiment programu *ExperimentsCreator* . . . 61

8 Záver **64**

Literatúra **66**

A Obsah priloženého média **68**

B Kompilácia zdrojových kódov **69**

C Problémy **70**

D Policy classes a *SmartPointer* **72**

E Knižnica *Diagram* **76**

Názov práce: Automatizace hierarchických experimentů
Autor: Martin Babka
Katedra (ústav): Katedra softwarového inženýrství
Vedúci bakalárskej práce: doc. RNDr. Tomáš Skopal, Ph.D.
e-mail vedúceho: tomas.skopal@mff.cuni.cz

Abstrakt: Cieľom tejto práce je navrhnúť a implementovať aplikáciu potrebnú k vytvoreniu hierarchického experimentu. V tejto práci uvádzame presnú definíciu hierarchického experimentu a analyzujeme jeho schopnosti. Princíp hierarchického experimentu je následne použitý k implementovaniu knižnice testujúcej softvér spôsobom zvaným testovanie bielej skrinky. Knižnica je určená hlavne k overeniu výkonnosti algoritmov. Pretože je vytvorená v jazyku C++, testovaný kód musí byť naprogramovaný v C alebo C++. Výstupom vytvoreného programu je používateľom definovaný testovací kód pre skúmané algoritmy a potrebný projektový súbor. V aplikácii je tiež možné aj jednoduché spracovanie zozbieraných dát. Test editujeme vizuálne v grafickom rozhraní.

Kľúčové slová: testovanie software, hierarchický experiment

Title: Automatizace hierarchických experimentů
Author: Martin Babka
Department: Department of Software Engineering
Supervisor: doc. RNDr. Tomáš Skopal, Ph.D.
Supervisor's e-mail address: tomas.skopal@mff.cuni.cz

Abstract: The aim of this work is to design and implement an application needed to create hierarchic experiments. The exact definition and purpose of hierarchic experiments are analysed in this work. The principle of hierarchic experiments is then used to implement a framework for white box software testing. It should by utilized mainly for benchmarking tests. The tested code must be in the C or C++ languages since the framework is developed in C++. Output of the application is a C++ source code that tests the user's algorithm in the selected manner. The application also outputs the needed makefiles and uses visual editing to create the testing set. It is also capable of simple processing of the collected performance data.

Keywords: software testing, hierarchic experiment

Kapitola 1

Úvod

K testovaniu softvéru, máme na výber nepreberné množstvo nástrojov. Tie sa líšia najmä ich použitím a cieľom, na ktorý sú zamerané. Pretože testovanie je dôležitou, ale často zanedbávanou súčasťou vývoja, je nutné mať aplikáciu, ktorá ho čo najviac zjednoduší a automatizuje.

Po naprogramovaní ľubovoľného programu ho musíme otestovať. Skúmame, či sa správa podľa našich očakávaní, a to nielen po stránke korektnosti výstupov, ale či spĺňa aj požadované výkonnostné parametre.

Počas skúmania rôznych algoritmov je dobré zaujímať sa, ako sa držia v „konkurenčnom boji“. Často chceme *naše* riešenie porovnať s už existujúcimi, alebo zachytiť jeho výkonnosť na rôznych vstupoch. Najprehľadnejším porovnaním je výstup v podobe tabuľky či grafu, ktorý na všetkých zaujímavých vstupoch zachytí sledované parametre jednotlivých riešení. Zostavenie nástroja, ktorý uľahčí tvorbu testu, generovanie vstupov a zachytenie výsledkov je cieľom tejto práce.

Ide nám o vytvorenie knižnice a prostredia na testovanie, ktoré spĺňa istú predstavu testu ako stromu. Tento strom nazývame hierarchický experiment, je to akýsi „pokus“ na testovanom algoritme. Počas priebehu tohoto „pokusu“ najprv dôjde k inicializácii vstupu. Neskôr k zostaveniu výstupu – vykonaniu algoritmov a napokon k jeho zozbieraniu. Jednotlivé kroky musia byť používateľsky definovateľné a konfigurovateľné, musia byť schopné prispôsobiť sa testovanému algoritmu.

V nasledujúcej kapitole rozoberieme princíp hierarchického experimentu a možnosti jeho použitia. V nasledujúcich dvoch kapitolách opíšeme vlastnosti aplikácie a prácu s ňou. Ďalej sa pozrieme na základnú sadu experimentov dodanú s inštaláciou programu. V šiestej kapitole načrtneme, ako systém funguje, hlavne prepojenie tvorby a behu experimentov. Napokon, s využitím týchto znalostí, budeme schopní definovať vlastné experimenty a rozšíriť tak možnosti, ktoré nám hierarchický test ponúka. Kvôli presnejšiemu vysvetleniu jednotlivých problémov je dobré pozrieť si video prílohu, objaví sa niekoľko základných otázok a, hlavne, odpovedí.

Kapitola 2

Hierarchický experiment

2.1 Motivácia

Testovanie softvéru, najmä testovanie rozličných algoritmov, v mnohých prípadoch vyžaduje zostrojenie vstupných dát, spustenie algoritmov na vytvorených dátach a vyhodnotenie výstupov. Takéto testy môžu mať rozličnú povahu. Podľa toho, čo je ich cieľom, rozlišujeme napríklad testy správnosti v rôznych podobách, ako je unit testing, regresné testovanie, alebo testovanie výkonnosti[1].

Počas zostavovania testovacích dát je vhodné dodržiavať pravidlá, ktoré sa menia najmä podľa zvoleného zámeru. Pokiaľ overujeme korektnosť kódu, je nutné dohliadať na pokrytie kódu, na vyčerpanie čo najväčšieho množstva možností priebehu programu. Pri testovaní výkonnosti sa naopak snažíme skonštruovať niekoľko sád, často podobných dát, ktoré sa líšia veľkosťou. Na výstupných dátach analyzujeme najmä ich závislosť na veľkosti vstupu. Nie je neobvyklé sledovať hodnoty viacerých premenných, ktoré zachytávajú výkonnostné parametre programu, zhodných pre algoritmy riešiace daný problém a porovnávať ich pre rôzne veľkosti a typy vstupov.

Testovaný algoritmus si predstavíme ako jednoduchú funkciu alebo metódu objektu, ktorá má nejaké vstupné parametre a jej výstupné parametre obsahujú skúmané premenné. Našou úlohou ostáva vytvoriť vstupné premenné, argumenty, s ktorými zavoláme zmienené podprogramy. Ďalej chceme zachytiť a spracovať požadované výstupy. Ako predstavu generovania vstupných dát môžeme použiť postupné vytváranie premenných, ktoré nakoniec použijeme ako vstup pre testovaný algoritmus.

Prirodzenou súčasťou tohoto postupu je, že tvorba nových premenných sa postupne vnára do hlbšej úrovne. Kód, ktorý inicializuje neskôr generované premenné, je vnorený do kódu obsluhujúceho už vygenerované hodnoty. Takto vzniká akási hierarchia krokov, ktoré nás dovedú až k vykonaniu testovaného kódu na vytvorených hodnotách. Jeden krok môže, samozrejme, nadobúdať rôzne podoby, napríklad cyklus, volanie iného podprogramu či skonštruovanie objektu. Každý krok sa, prirodzene,

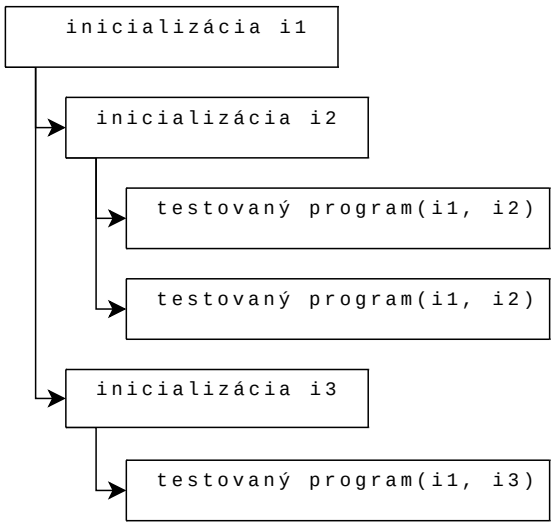
môže odkazovať na už predtým vytvorené premenné.

Napríklad počas testovania sčítania dvoch dlhých čísel máme zrejme vytvorené dva zanorené cykly. Ich iterátory budú vstupom pre testovanú funkciu. Samozrejme, nepokryjeme všetky možné vstupy, musíme zvoliť správny prístup a vybrať len tie najdôležitejšie. Toto rozhodnutie vykonáme na základe nárokov kladených na analýzu programu [1].

2.2 Hierarchický experiment

Kroky, nazvime ich *experimenty*, ktoré viedli k vytvoreniu jednotlivých premenných oddelíme a umiestnime ich do grafu. V ňom hranami spojíme len tie experimenty A a B, pre ktoré platí, že vykonanie experimentu A bezprostredne predchádza vykonaniu kroku B a B je vnorený do A. Takto získame cestu vedúcu až k finálnemu volaniu testovaného kódu reprezentujúceho jeden testovaný algoritmus.

Pokiaľ testujeme viac algoritmov, získame viac volaní funkcií a teda aj viac ciest. Pretože volané funkcie sa môžu odlišovať v signatúre, môžeme vytvoriť cesty, ktoré majú rôzne nielen listové experimenty, ale aj tie vnútorné. Takto získaný strom[2] považujeme za *hierarchický experiment*. Ak vzniknuté cesty, ktoré vedú k rôznym listovým experimentom, nemajú žiadny spoločný vrchol, potom pre všetky stromy v tomto lese[1] dodefinujeme ich spoločný koreň. Takto dosiahneme to, aby hierarchický experiment bol za každých okolností strom.



Obrázok 2.1: Hierarchický experiment

[1]Les označuje graf tvorený množinou disjunktných stromov, komponenty súvislosti sú práve tieto stromy.

2.3 Definícia

Nech A_1 až A_n sú testované algoritmy, $n_j \in \mathbb{N}$ je počet vstupných argumentov a $i_{j,1}, \ldots, i_{j,n_j}$ sú rôzne vstupné parametre algoritmu A_j. Označme $Call(A_j)$ experiment, ktorý vykoná algoritmus A_j s hodnotami parametrov $i_{j,1}, \ldots, i_{j,n_j}$. Nech $Init(i_{j,k})$ je experiment, ktorý viedol k inicializácii hodnoty parametra $i_{j,k}$.

Nech $\bar{H} = (V, E)$ je orientovaný graf, kde:

$$V = \left\{Init(i_{j,k}) \mid \forall j \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, n_j\}\right\} \cup \left\{Call(A_j) \mid \forall j \in \{1, \ldots, n\}\right\}$$
$$E \subseteq V \times V$$

Je nutné, aby graf $\bar{H}$ bol les a z toho plynie, že množina hrán E definuje hierarchiu experimentov, teda poradie vytvárania vstupu na cestách od koreňa k listom.

Táto požiadavka vychádza z myšlienky, že každý z experimentov dostáva na vstupe hodnoty už inicializovaných argumentov a môže ich počas svojho výpočtu použiť. Jeho úplným výstupom sú hodnoty zo vstupu a tie hodnoty, ktoré sám vytvára. Pokiaľ by vznikol v $\bar{H}$ cyklus, nemohli by sme určiť, čo je výstupom experimentov ležiacich na kružnici.

Ďalšou nutnou požiadavkou je, aby na ceste od koreňa komponentu, kde leží experiment $Call(A_j)$ boli inicializované všetky parametre $i_{j,1}, \ldots, i_{j,n_j}$ hodnotami z $Dom(i_{j,1}), \ldots, Dom(i_{j,n_j})$. $Dom(i)$ označuje priestor hodnôt premennej i.

Potom *hierarchickým experimentom* pre algoritmy A_1, ..., A_n nazveme strom:

$$H = \begin{cases} \bar{H} & \text{ak } \bar{H} \text{ je strom} \\ & \\ & \text{inak, } \bar{H} \text{ je les} \\ (\bar{V}, \bar{E}) & \bar{V} = V \cup \{Composite\} \\ & \bar{E} = E \cup \{(Composite, v) \mid v \text{ je koreň stromu v grafe } \bar{H}\}) \end{cases}$$

Výstupom experimentu E, označme $Out(E)$, je n-tica ním vytváraných hodnôt. $Out(Init(i_{j,k})) = (i_{j,k})$ je hodnota z $Dom(i_{j,k})$. $Out(Call(A_i))$ sú hodnoty sledovaných výstupov pre zadaný vstup a $Out(Composite) = ()$.

Hierarchický experiment H určuje spôsob, akým sa generujú vstupné dáta pre každý z listov, ich domény a vnorenie. Pokiaľ je H pestovaný strom[2], čo budeme odteraz predpokladať, môžeme jednoznačne definovať aj poradie volania experimentov ako preorder priechod stromu H. Definovaním poradia získavame jednoznačný predpis pre testovací program.

[2]Pestovaný strom má presne určené poradie potomkov každého uzla a koreň. Presnú definíciu nájdeme v [2].

2.4 Predstava, možné využitie a obmedzenia

Hierarchický experiment môže byť principiálne využitý k vykonaniu a analyzovaniu akéhokoľvek kódu. Jeho hlavnou úlohou je zostrojiť dáta potrebné pre spustenie rôznych podprogramov. Tieto podprogramy majú byť ďalej skúmané najmä tak, že budú porovnávané hodnoty ich spoločných charakteristík na rôznych vstupoch. Hodnoty analyzovaných veličín musí testovaný program sám produkovať a vracať typicky v listovom experimente. Tieto hodnoty budú ďalej zachytávané pomocou experimentov, ktoré sa starajú o zber dát. Akýkoľvek uzlový experiment nemá predpísanú úlohu v hierarchii. Oproti predchádzajúcej definícii, je možné, aby modifikoval už vytvorené hodnoty, aby sa staral o zber výstupov, alebo aby len vytváral a inicializoval nové premenné.

Cieľom hierarchických experimentov je umožniť vytvorenie takmer ľubovoľného testovacieho kódu a uľahčenie jeho tvorby v mnohých prípadoch. Pokiaľ umožníme, aby jeden z experimentov mohol obsahovať užívateľom definovaný kód, zbavíme sa akýchkoľvek obmedzení. Hlavným cieľom avšak stále ostáva využitie preddefinovanej množiny experimentov ako základných prostriedkov pre definíciu hierarchického stromu. Pri využití už definovanej sady oproti klasickému písaniu vlastného programu získavame výhodu rýchlejšej tvorby testov a väčšej prehľadnosti – test je dekomponovaný do menších krokov.

Hierarchické experimentovanie taktiež presne neurčuje účel a cieľ testovania. Výstupné veličiny je možné interpretovať ľubovoľným spôsobom; napríklad ako informáciu o úspechu či zlyhaní pri testovaní správnosti, alebo ako hodnotu sledovaného výkonnostného parametra.

Určite za vhodné využitie môžeme považovať také prípady, kde sa premenné zostavujú väčšinou vo vnorených cykloch. Ďalšie z nich vzniknú vytvorením používateľských objektov a výsledky sú väčšinou návratové hodnoty metód týchto objektov. Uľahčenie tvorby testu spočíva v tom, že máme preddefinovanú sadu cyklov, možnosť inicializovať objekty a nastaviteľné experimenty pre zber dát. Menej výhodné je použitie pokiaľ nám chýba potrebný „krok“. Jediná zostávajúca možnosť je použiť experiment obsahujúci používateľský podprogram, alebo definovať si vlastný experiment. Vytvorenie nového experimentu sa vyplatí najmä v prípadoch jeho opakovaného použitia.

Kapitola 3

Používateľská dokumentácia

V tejto časti budeme popisovať prácu s programom *ExperimentsCreator* – s grafickým rozhraním, postupy potrebné k vytvoreniu hierarchického experimentu, jeho načítanie a ukladanie. Objasníme integráciu vlastných experimentov a konfiguráciu programu. Spôsob použitia vytvoreného programu a jeho integráciu do testovaného projektu popíšeme na záver tejto kapitoly.

3.1 Rozdelenie programu na moduly

Výsledkom tejto práce je program pre tvorbu hierarchických experimentov, podporu zberu výstupných dát, ich jednoduchú analýzu a spracovanie. Testované programy sú v jazyku C++, program pre tvorbu hierarchických experimentov je implementovaný v jazyku C#. S programom užívateľ pracuje tak, že v prostredí pre tvorbu experimentov si najprv graficky nadefinuje vlastný test, potom nechá vygenerovať jeho zdrojový kód a ten následne preloží a spustí.

Vygenerované zdrojové kódy sa opierajú o už naprogramovanú knižnicu, ktorá riadi vytvorenie, inicializáciu, beh experimentu a uloženie výstupov. Pri využití tejto knižnice vzniká otázka, ako sa bude spájať s už existujúcim testovaným projektom. Existujú dve alternatívy. Užívateľ buď pripojí jej zdrojové kódy k testovanému projektu alebo využije už vopred skompilované knižnice, čo je preferovaný spôsob použitia. Pri vygenerovaní testov z grafického rozhrania (GUI) bude umožnené vytvorenie alebo modifikácia existujúceho projektu tak, aby potrebné knižnice boli pridané automaticky.

Jedným z najdôležitejších problémov je možnosť čo najväčšieho prispôsobenia sa testovanému kódu. Základné riešenie tohoto problému je možnosť vykonania používateľsky vkladaného kódu. V jednoduchších prípadoch postačí zavolanie metódy objektu alebo funkcie. Pri zložitejších a opakujúcich sa problémoch je riešením možnosť vytvoriť si vlastný experiment, ktorý sa bude dať použiť nielen v rámci

jedného testovacieho kódu. Samozrejme, bude s ním možné pracovať aj počas grafického návrhu.

3.2 C++ knižnica

Knižnica *Experiments*, ako bolo uvedené, slúži ako podpora vygenerovanému testovaciemu kódu. Obsahuje definíciu experimentu, spôsob vykonania celého testu a časť, ktorá sa stará o uchovávanie hodnôt, prenos už vytvorených hodnôt pri komunikácii medzi experimentmi a zber výstupných dát. Jednotlivé časti tejto knižnice musia byť vymeniteľné a dostatočne prispôsobiteľné účelu testovania. Preto obsahuje najmä abstraktné triedy a ich základné implementácie.

Súčasti knižnice:

- Definícia experimentu; experimentu vo vnútri stromu a listového experimentu pre zľahčenie tvorby vlastných experimentov.
- Základné experimenty použiteľné vo všetkých testoch.
- Polymorfné objekty uchovavájúce hodnoty generovaných premenných.
- Úložiská vygenerovaných dát, experimenty ukladajúce výstupné dáta.
- Objekty obsluhujúce výpis týchto úložísk.

3.3 Program pre tvorbu experimentov

Program pre tvorbu experimentov, *ExperimentsCreator*, je naprogramovaný v jazyku C#. *ExperimentsCreator* má dve rozhrania, dá sa použiť ako konzolová aplikácia alebo ako program s GUI.

Účelom GUI je definícia ľubovoľného hierarchického experimentu a následné vytvorenie zdrojového kódu. V GUI užívateľ graficky definuje hierarchický experiment, nastaví parametre jednotlivých podexperimentov a zapojí ich do želanej hierarchie. Aby bola uľahčená práca s vygenerovaným programom, existuje možnosť využiť vygenerovanie alebo úpravu už existujúceho C++ projektu programu *Microsoft Visual Studio*. Ďalšou dôležitou vlastnosťou je možnosť pridávať a vytvárať vlastné experimenty tak, aby boli použiteľné ako v GUI, tak aj vo výslednom programe. Ďalšie možnosti programu:

- Možnosť uložiť a načítať už vytvorené hierarchické experimenty.
- Podpora pre „nekonečné“ Undo a Redo.

- Diagram obsahujúci zoom, možnosť automaticky rozložiť jednotlivé experimenty, použitie mriežky pre presné umiestnenie objektov.
- Základná podpora pre spracovanie výsledkov – graf a tabuľka zostavená z výstupných dát.

Úlohou konzolovej verzie programu je premena vopred definovaného experimentu na testovací kód.

3.4 Inštalácia, požiadavky a spustenie programu

Program ExperimentsCreator k svojmu behu vyžaduje knižnicu .NET Framework vo verzii 3.5. Túto knižnicu je možné získať na stránkach firmy Microsoft[3]. Jej inštalácia je ponúkaná aj počas inštalácie v prípade, že ju používateľ ešte nemá nainštalovanú, avšak k jej prevzatiu je nutné pripojenie k sieti Internet.

Pokiaľ užívateľ zvolí manuálnu inštaláciu z komprimovaného archívu a nebude mať nainštalovaný .NET Framework v 3.5, program sa odmietne spustiť a skončí chybou, pretože nie je možné nájsť požadované systémové súbory.

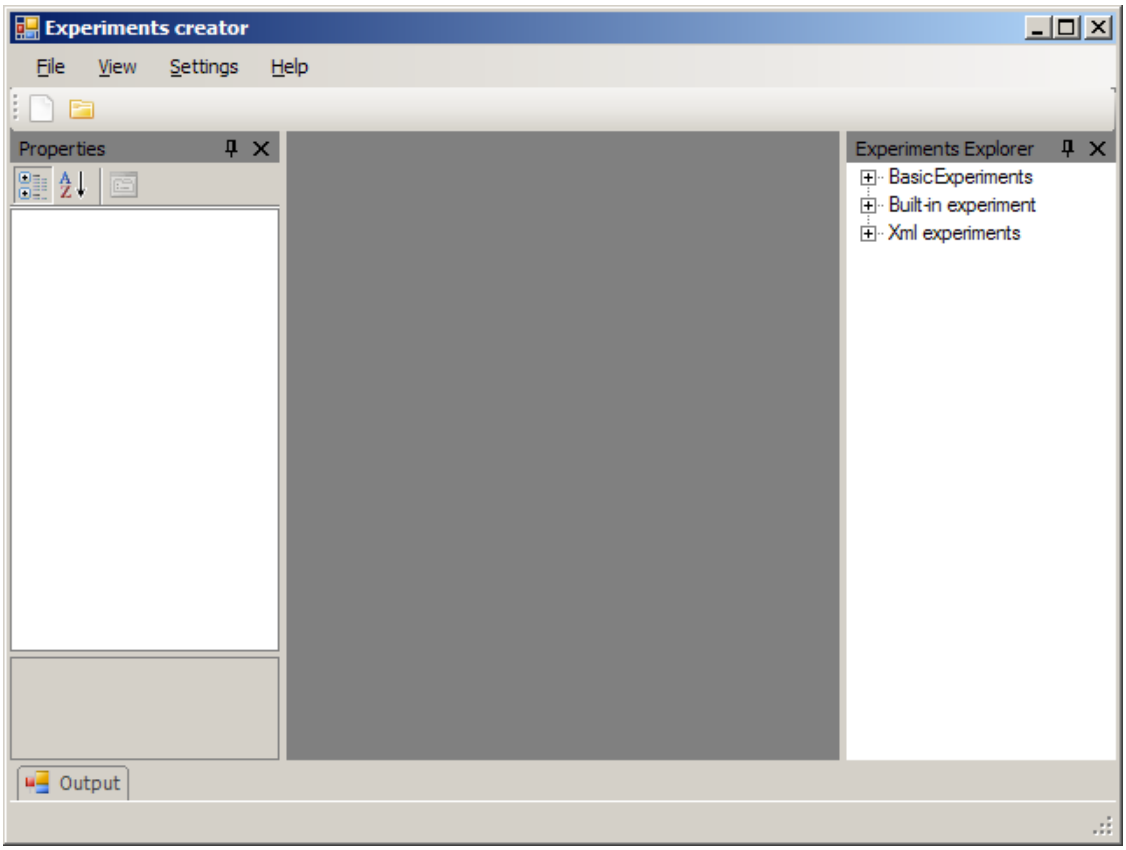
Vytvárané programy a knižnica Experiments využívajú C++ projekty programu *Microsoft Visual Studio 2008* alebo makefile programu cmake[4]. Poskytnuté statické knižnice sú vhodné pre kompilátor gcc, platformy x86 a x86_64, a MS VS 2008.

Grafické rozhranie programu ExperimentsCreator vyvoláme spustením súboru *ExperimentsCreatorGUI.*exe. Ten nájdeme v inštalačnom adresári alebo v adresári, kde sme rozbalili projekt, ak sme použili manuálnu inštaláciu. Odteraz budeme pojmom *inštalačný adresár* rozumieť obe možnosti, nakoľko ich obsah je rovnaký. Po spustení sa zobrazí základné okno editora experimentov tak, ako ho vidíme na obrázku 3.1.

Na ľavej strane sa nachádza záložka *Properties*, v ktorej sa zobrazujú a editujú vlastnosti experimentu – jeho meno, typ, jednotlivé vlastnosti. Vpravo nájdeme zoznam použiteľných experimentov, panel *Experiments explorer*, tie je následne možné zapájať do vytváraných hierarchických experimentov. V hornej časti programu môžeme použiť menu a panel nástrojov, čo nám uľahčuje prácu s programom. Činnosť aplikácie sledujeme dole v okne výstupov *Output* a stavovom riadku.

3.5 Základné použitie

Najdôležitejšími krokmi v používaní aplikácie je vytvorenie si vlastného hierarchického experimentu, jeho validácia a následný export do C++, aby bolo možné spustiť testovanie programu.



Obrázok 3.1: Hlavné okno programu ExperimentsCreator

Nový experiment vytvoríme použitím voľby *New* v menu *File*, prípadne rovnocenným tlačítkom v paneli nástrojov. Zobrazí sa plocha s diagramom, v ktorej je možné graficky definovať hierarchický experiment.

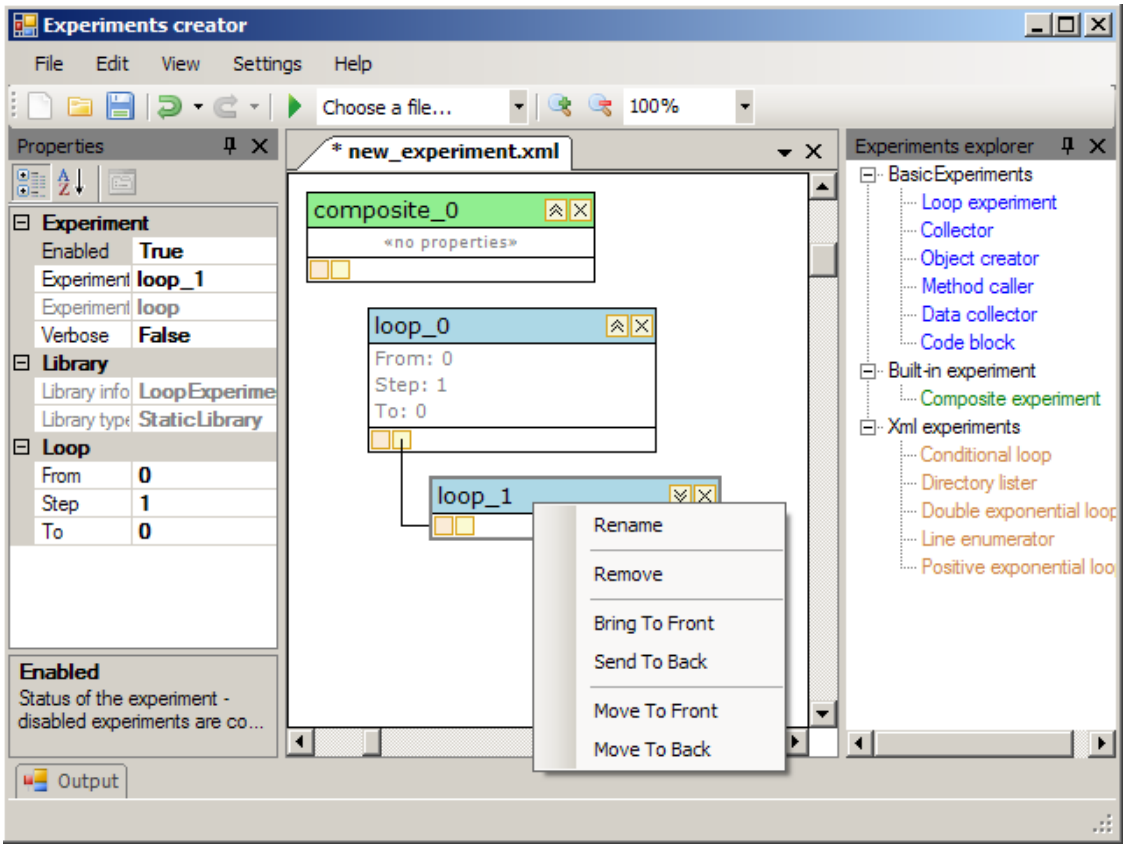
Existujúci experiment načítame kliknutím na položku *Open* v menu *File*. Rozpracovaný experiment uložíme pomocou tlačítka *Save* alebo *Save as*, majú rovnaký význam ako v podobných programoch. Pri ukladaní experimentu je potrebné, aby bol správny vo význame popísanom v časti 3.10, strana 17.

3.6 Nový experiment, práca s experimentom

Najdôležitejšou operáciou je pridanie nového experimentu. Túto úlohu vykonáme tak, že zvolený experiment pretiahneme z panelu *Experiments explorer* na vhodné miesto na editačnej ploche. Ak na experiment dvakrát klikneme, automaticky sa pridá do diagramu na pozíciu $(0, 0)$. Táto pozícia sa s každým, takýmto spôsobom vloženým objektom, pozmení. Pokiaľ je viditeľná oblasť diagramu posunutá, nový prvok môže byť vložený mimo zobrazovaný priestor.

Po vložení experimentu a jeho vybratí tak, že naň klikneme, môžeme meniť jeho vlastnosti v ľavom paneli *Properties*, ako je zrejmé z obrázku 3.2. Pokiaľ zadáme nesprávnu hodnotu, program nás upozorní varovným hlásením. Pri niektorých typoch experimentov nie je možné okamžite rozpoznať správnosť zadanej hodnoty. Pokiaľ by takáto vlastnosť bola zle nastavená počas ukladania či exportovania experimentu dostávame chybovú hlášku s menom experimentu a musíme hodnotu opraviť. Jednotlivé experimenty a ich vlastnosti nájdeme popísané v kapitole 4.

Na obrázku 3.2 vidíme kontextové menu, ktoré je možné vyvolať na každom objekte v diagrame. Ak je opisovaná ponuka vyvolaná na experimente, obsahuje navyše položku *Rename*, ktorou zmeníme meno experimentu. Ostatné položky slúžia na posúvanie vybratého objektu po osi z. Prvé dve posúvajú objekt na prvú, prípadne poslednú pozíciu, druhý pár posúva objekt len o jedno miesto. Túto funkciu využijeme najmä v prípade, ak by sme chceli skryť vzniknuté spojnice experimentov. Umiestnenie objektov na osi z sa mení ešte v prípade označenia objektu. Vybratý objekt je vždy najvyššie na osi z a teda je vždy viditeľný.



Obrázok 3.2: Práca s diagramom

Dôležitou možnosťou je experimenty znova rozpájať a mazať. Tieto operácie pre-

vedieme tak, že kliknutím si vyberieme mazaný objekt, na ňom vyvoláme kontextové menu a použijeme menu *Remove*. K rovnakému výsledku sa dopracujeme aj stlačením klávesy *Del* na označenom objekte. Pokiaľ vymažeme hranu spájajúcu dva experimenty, dôjde k ich korektnému rozpojeniu a príslušné porty je možné znovu použiť.

Reprezentácia experimentu v diagrame má dva stavy, môže byť rozbalená alebo minimalizovaná. V prvom stave sú zobrazené aj vlastnosti špecifické pre daný typ experimentu. Medzi týmito stavmi prepíname pomocou prvého tlačítka v hornej lište experimentu. Druhé tlačítko je alternatívny spôsob vymazania experimentu. Experimentom v diagrame môžeme pohybovať obdobným spôsobom ako oknom v bežných grafických prostrediach.

3.7 Spájanie experimentov

Počas vytvárania testu je dôležité vytvárať hierarchiu – experimenty musíme zapájať do stromu. Sémantika vytváraného stromu je taká, že najprv sa spustí koreňový experiment. Ten spustí svojich potomkov a tento proces sa rekurzívne opakuje na každej na úrovni v poradí preorder. Každý experiment po jeho spustení a vykonaní úloh, zavolá svojich potomkov so všetkými dosiaľ vytvorenými hodnotami na jeho úrovni. Práca, ktorá bude po spustení uzlu realizovaná, samozrejme, nadobúda s rôznymi „pokusmi“ rôzne podoby. Napríklad cyklus bude volať svojho potomka pre každú hodnotu, cez ktorú iteruje. *Composite* len zavolá potomkov v poradí ich zapojenia s hodnotami, ktoré mu boli poskytnuté.

Každý experiment obsahuje v dolnej časti dva malé štvorce, porty. Prvý z nich je vstupný port, druhý je výstupný. Spájanie experimentov prevedieme nasledovným postupom. Klikneme na výstupný, teda druhý, port rodičovského experimentu, t.j. toho experimentu, ktorého výstup bude vstupom jeho dieťaťa. Po kliknutí sa objaví modrá čiara, ktorá sa bude pohybovať za kurzorom. Vstupný port, na ktorý chceme napojiť predtým zvolený port a experiment, vyberieme znova kliknutím. Pokiaľ nie je možné vybrané prvky spojiť, obdržíme chybové hlásenie. Napríklad, nie je možné vytvoriť hranu medzi dvomi vstupnými či výstupnými portami. Výstup experimentu určeného na použite do listov nie je možné spojiť so žiadnym z portov. Takýmto experimentom je napríklad zberač dát – *Collector*.

3.8 Meno a spoločné vlastnosti experimentov

Jedna z vlastností experimentu je jeho meno. Každý experiment ho musí mať jednoznačne určené, t.j. v jednom hierarchickom experimente nemôžu existovať dva

experimenty s rovnakým menom, a musí to byť platný identifikátor jazyka C++. Tieto obmedzenia sú spôsobené tým, že vo vytváranom kóde sú premenné obsahujúce experimenty, nazvané práve zadaným menom. Pri uvedení hodnoty, ktorá nie je platný identifikátor, obdržíme príslušnú chybovú hlášku. V prípade, že sme použili duplicitné mená, prejaví sa to pri práci s celým experimentom, napríklad pri ukladaní a exportovaní do C++.

Na obrázku 3.2, strana 15, v paneli *Properties* pod kategóriou *Experiment* vidíme základné vlastnosti experimentu:

- *Enabled*: Povolenie či zakázanie experimentu. Pokiaľ je táto vlastnosť nastavená na *false*, experiment a celý jeho podstrom sa nebude v teste vykonávať. Do testovacieho kódu bude vygenerovaný v komentári. Počiatočná hodnota je u každého experimentu nastavená na *true*.

- *Experiment name*: Meno experimentu.

- *Experiment type*: Typ experimentu, hodnota tejto vlastnosti sa nedá zmeniť, zobrazuje sa informácia o type experimentu. Podľa tejto vlastnosti sa vyfarbujú aj horné lišty objektov v diagrame.

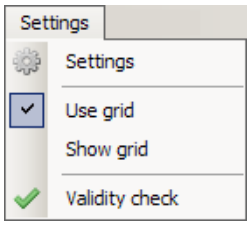
- *Verbose*: Vlastnosť, ktorá ovplyvňuje zobrazovanie hlásení experimentu počas behu testovacieho kódu. Typicky, vždy keď je experiment zavolaný a táto vlastnosť má hodnotu *true*, experiment vypíše na štandardný výstup hlásenie, že bol spustený.

3.9 Nastavenia diagramu

Pri práci s diagramom je vhodné mať možnosť rozmiestňovať jeho prvky podľa želania. Avšak je dobré, keď používateľ má k dispozícii mriežku, ku ktorej sa budú jednotlivé objekty automaticky zarovnávať. Tú zapneme povolením voľby *Use grid* v menu *Settings*. Ak je mriežka zapnutá, jej zobrazenie môžeme kontrolovať voľbou *Show grid*. Posledná ponuka *Validity check* slúži na stálu kontrolu správnosti diagramu.

3.10 Overenie správnosti experimentu

Pred tým, ako začneme experiment ukladať alebo exportovať do testovacieho kódu, experiment musí byť správne definovaný. Jeho korektnosť overíme voľbou *Check validity* v menu *Edit*. Základné požiadavky na bezchybný experiment sú nasledovné:



Obrázok 3.3: Nastavenia diagramu

- Experimenty musia mať rôzne mená. Pokiaľ táto požiadavka nie je splnená, zobrazí sa chybové hlásenie, *„Name 'experiment' is not unique.“*, ktorého obsahom je duplicitné meno. V tomto prípade to je „experiment“.

- Experiment musí byť strom – súvislý a acyklický graf. V prvom prípade dostaneme chybovú hlášku, *„Experiment is not connected.“*, oznamujúcu, že graf je nesúvislý. V druhom správu, *„Experiments contains a cycle ['Exp1' → 'Exp2'].“*, ktorá poukazuje na jednu hranu z cyklu.

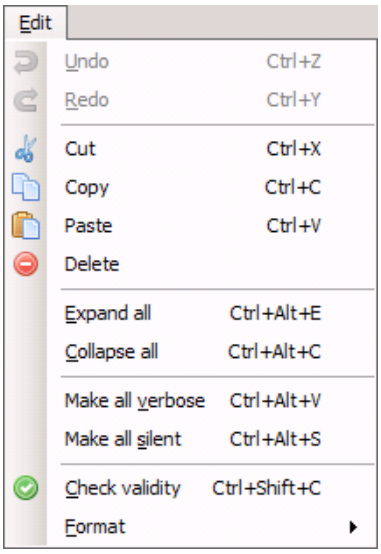
- Každý experiment musí byť správne nastavený. Jednotlivé prvky majú kontrolu nad tým, či sú splnené ich požiadavky, napríklad na vstup, alebo nie. Pokiaľ splnené nie sú, zobrazí sa okno so správou, *„Experiment 'experiment' is invalid.“*, s menom zle nastaveného experimentu. Ten je nutné opraviť podľa jeho špecifických vlastností a potrieb.

V predchádzajúcej sekcii sme uviedli nastavenie *Validity check*. Jeho úlohou je automatická kontrola správnosti celého hierarchického experimentu počas práce. Pokiaľ je projekt chybný, farba hornej lišty neplatne nastavených experimentov sa zmení na červenú. Akokoľvek, pokiaľ experiment nie je strom, kontrola správnosti sa ihneď zastaví a teda niektoré nesprávne experimenty nemusia byť vyfarbené. Táto kontrola funguje počas práce s experimentom, nutne po každej vykonanej akcii, preto jej zapnutie je náročnejšie na prostriedky užívateľa.

Kontrolu správnosti práve definovaného hierarchického experimentu ručne vyvoláme v menu Edit voľbou *Check validity*. Pokiaľ je experiment správny, dostaneme informáciu o jeho bezchybnosti, inak jednu z vyššie popísaných chybových hlášok.

3.11 Práca s hierarchickým experimentom

Na obrázku 3.4 je menu *Edit*, ktoré slúži na prácu s hierarchickým experimentom. Prvé dve voľby *Undo* a *Redo* slúžia na zrušenie alebo obnovenie vykonanej akcie. Počet „Undo“ operácií nie je dopredu nijakým spôsobom obmedzený.



Obrázok 3.4: Menu edit

K skopírovaniu a vloženiu označeného experimentu môžeme použiť voľby *Copy*, *Cut* a *Paste*. Pomocou *Delete* vymažeme aktuálne označený objekt. Tieto štyri voľby sú určené pre prácu s jedným objektom. Experimenty sa dajú vyberať, vkladať a kopírovať len samostatne.

Akcie *Expand all* a *Collapse all* využijeme v prípade, ak chceme vypnúť alebo zapnúť zobrazovanie vlastností objektov v diagrame; teda na prepnutie stavu všetkých objektov v diagrame na rozbalené, či minimalizované.

Položky *Make all verbose* a *Make all silent* zapínajú alebo vypínajú vlastnosť *Verbose* všetkých experimentov v diagrame.

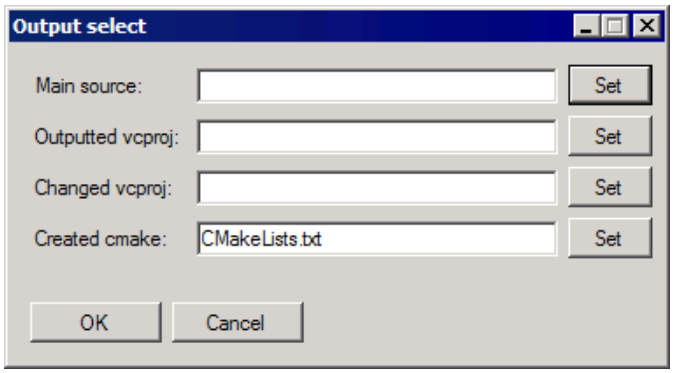
Možnosť *Check validity* sme rozobrali v predchádzajúcej sekcii.

Pod záložkou *Format* môžeme nechať automaticky rozložiť prvky v diagrame. Na výber máme z dvoch ponúk – *Horizontally* a *Visual code*. Prvá z nich preferuje zarovnávanie objektov horizontálne, druhá vytvorí rozloženie podobné tomu, aké sa bežne používa pri písaní zdrojových kódov.

3.12 Generovanie výstupného kódu

Testovací program vytvoríme tak, že na paneli nástrojov zvolíme zelenú šípku, prípadne použijeme menu *File* a v ňom voľbu *Export*. Následne sa otvorí dialóg, v ktorom popisujeme výstup. V zobrazenom formulári, obrázok 3.5, môžeme taktiež nastaviť generovanie projektu pre Visual Studio[5] alebo update už existujúceho projektu.

Po zmene hierarchického experimentu, pokiaľ chceme opätovne generovať upravený výstup, môžeme aplikovať už raz zadané nastavenie. V paneli nástrojov zvolíme



Obrázok 3.5: Dialóg určený k nastaveniu výstupu

už použitý profil. V rozbaľovacom paneli, je identifikovaný menom výstupného súboru, ktoré sme zadali. Vyberieme si vhodný profil a klikneme na tlačítko „generuj výstup“. Ak chceme zadať nový, zvolíme ponuku *Choose a file...*

Pole *Main source* určuje súbor, do ktorého bude vygenerovaná funkcia `main`. Vytvorený zdrojový kód taktiež obsahuje potrebnú inicializáciu experimentov a v komentári zoznam ostatných vygenerovaných súborov s potrebnými knižnicami. Správne vyplnenie tejto časti je vyžadované pri každom nastavení generovania výstupu.

V poli *Outputted vcproj* určíme meno nanovo generovaného projektu. Ten obsahuje len vytvorené zdrojové kódy a referencie na knižnice, ktoré boli zmienené pri tvorbe hierarchického experimentu. Toto nastavenie je nepovinné.

Nepovinnou časťou je možnosť zmeniť už existujúci projekt, pole *Changed vcproj*. Keď už máme algoritmus naprogramovaný a chceme ho otestovať, je výhodné, pokiaľ môžeme vytvorené testy priamo pridať do existujúceho projektu. K tomu je ďalej nutné nastaviť aj adresáre k potrebných hlavičkovým súborom a knižniciam. Práve zmienené úlohy vykoná práve popisovaná voľba. Pred modifikáciu projektu je, pochopiteľne, vytvorená jeho záloha. Vygenerované súbory nie sú pridané priamo do koreňa projektu, ale pod záložku `Test`. Pokiaľ sme nenastavili výstup priamo do adresára `Test` zvoleného projektu, musíme vytvorené súbory ručne premiestniť.

Poslednou ponukou *Created cmake* vygenerujeme makefile pre systém *CMake*[4][1]. Generovanie sa nedá ovplyvniť priamym nastavením adresárov. Používajú sa tie, ktoré sú určené pre štandardné hlavičkové adresáre a knižnice v release móde. Nie je to výrazné obmedzenie, pokiaľ program používame len pre tento systém. Potom ho musíme, samozrejme, prekonfigurovať. Ak chceme používať projekty *CMake* aj programu *Microsoft Visual Studio*, odporúčam zmenu šablóny použitú k vygenerovaniu makefilu. Nájdeme ju v inštalačnom adresári v zložke `Templates`, súbor `cmake.tpl`. Podľa vložených komentárov je zrejmé, akým spôsobom ju nastavíme.

[1]Táto časť je pomerne nová. Ide nám skôr o demoštráciu použitia aj pre iné platformy.

3.13 Nastavenie programu

V predchádzajúcej časti sme sa zmienili o tom, že počas vytvárania projektov programu *Microsoft Visual Studio*, nastavujeme cesty ku knižniciam a hlavičkovým súborom. Tieto adresáre je možné zmeniť vo formulári, ktorý vyvoláme položkou *Settings* v menu *Settings*, pod záložkou *Directories*.

Nastavenia adresárov, záložka *Directories*:

- *Include directories*: Adresáre, v ktorých sa nachádzajú hlavičkové súbory knižnice *Experiments*.
- *Debug libraries*: Adresár, v ktorom sú uložené statické knižnice skompilované pre *debug mode*. Tieto súbory obsahujú knižnicu *Experiments* a budú prilinkované k testovanému programu.
- *Release libraries*: Adresár, ktorý obsahuje súbory potrebné k prilinkovaniu knižnice *Experiments* pre preklad v *release mode*.
- *Additional experiments*: Zoznam umiestnení, v ktorých budú počas inicializácie programu hľadané sady vlastných a štandardných experimentov. Nájdené experimenty budú automaticky integrované do prostredia programu *ExperimentsCreator*.

Nastavenia prostredia, záložka *GUI*:

- *Background color*: Nastavenie farby pozadia diagramu.
- *Show experiments expanded*: Pri nastavení tejto voľby budú novo pridané experimenty rozbalené; ich vlastnosti sa budú zobrazovať aj v diagrame.
- *Experiments are verbose*: Ak je voľba zaškrtnutá, experimenty budú počas behu testovacieho programu zobrazovať hlásenia o ich vykonaní.
- *Perform validity check*: Nastavenie ovplyvňujúce automatickú kontrolu hierarchických experimentov. Ak ho povolíme, všetky experimenty budú automaticky kontrolované.
- *Use grid*: Nastavuje použitie mriežky, ku ktorej sa budú objekty automaticky zarovnávať.
- *Show grid*: Ak sa používa mriežka, ovplyvňuje jej zobrazovanie. Ak sa mriežka nepoužíva, samozrejme, nezobrazuje sa.
- *Step x*, *Step y*: Vzdialenosti medzi bodmi mriežky na osi x a y.

Zmienené nastavenia je možné ovplyvňovať pre jednotlivé experimenty zvlášť rovnomennými položkami, ktoré nájdeme v menu *Edit* a *Settings*.

Kapitola 4

Popis jednotlivých experimentov

Experimenty v programe *ExperimentsCreator* a knižnici *Experiments* sú úzko prepojené. Každý experiment v aplikácii *ExperimentsCreator*, musí mať svoj ekvivalent vo vygenerovanom testovacom kóde. Existuje niekoľko spôsobov, ako toto prepojenie vytvoriť. Najjednoduchší prípad takéhoto prepojenia je ten, že experiment je nadefinovaný v C++ a jediným cieľom GUI je nastaviť jeho vlastnosti. Najzložitejší spôsob spočíva v generovaní celého experimentu počas vytvárania testovacieho kódu.

Experimenty sa do programu vkladajú ako zásuvné moduly tak, ako je to presne popísané v kapitole 7. Na tomto mieste je však nutné podotknúť, že pokiaľ užívateľ nevykoná žiadne zmeny v nastavení programu, experimenty sa automaticky načítajú z adresára `Experiments`. Ten sa nachádza v inštalačnom adresári a obsahuje základnú sadu experimentov. Umiestnenia, z ktorých budú používateľsky definované experimenty načítané, sa dajú zmeniť v nastaveniach programu.

Máme dve možnosti ako integrovať sadu vlastných experimentov do programu. Buď naprogramujeme vlastnú assembly jazyka C# a tá sa automaticky nahrá alebo využijeme zjednodušený spôsob – definíciu vlastného experimentu. Pri zvolení prvej možnosti máme väčšie možnosti, avšak je nutné programovať aj časti, ktoré sa netýkajú samotného testovania. Druhá možnosť nám umožňuje jednoduchú integráciu testu do programu, stačí naprogramovať experiment v C++ a vo formulári nadefinovať jeho vlastnosti, avšak strácame niektoré možnosti.

Dodané experimenty, sú k programu pripájané nezávisle, rovnakým spôsobom ako používateľsky definované sady. Jediným vstavaným experimentom je *Composite*.

4.1 Premenné experimentov

Jednou z najdôležitejších úloh hierarchického experimentu je generovanie dát – či už vstupných alebo výstupných. K tomuto účelu musíme byť schopní vytvárať premenné a uchovávať ich hodnoty. Je nutné, aby sa experimenty mohli odkazovať na

už existujúce hodnoty, pracovať s nimi a na ich základe vytvárať nové údaje. Preto potrebujeme mať v knižnici *Experiments* flexibilný spôsob ako vytvárať úložiská pre hodnoty akéhokoľvek typu, aj dopredu neznámeho. V prostredí na tvorbu experimentov, programe *ExperimentsCreator*, zase chceme kontrolovať správnosť odkazov – existenciu premennej a jej typ.

Výstupné premenné sú veličiny, ktoré experiment generuje. Pri každej z nich je počas definície experimentu určený jej typ, meno a poradie v rámci experimentu. Z týchto údajov vieme premennú jednoznačne adresovať cestou k nej. Začneme od koreňa, určíme zoznam experimentov, cestu, k uzlu, ktorý ju generuje, a jej meno.

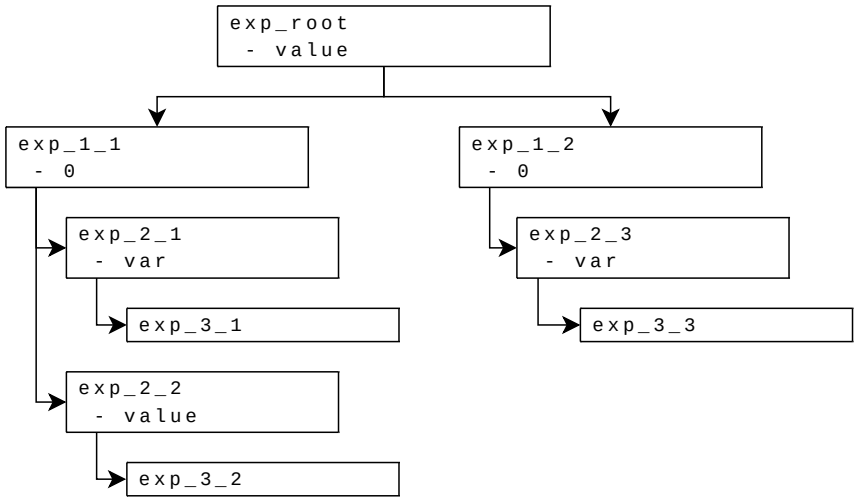
Vieme, že výstupné hodnoty každého experimentu sa stávajú vstupom jeho potomkov a teda premennú môžeme určiť jej poradím na vstupe niektorého experimentu. V tomto prípade je nutné k odkazu pripojiť aj informáciu o type hodnoty.

Vstupné premenné, na ktoré sa odkazujeme patria medzi požiadavky, ktoré musia byť splnené, aby bol experiment správne definovaný, t.j. musia existovať na jeho vstupe. Jednotlivé požiadavky vznikajú až v momente, keď sa odkážeme na veličinu. Premenné, ktoré experiment využíva, nutne vnikajú na ceste od koreňa k odkazujúcemu sa uzlu, aby vôbec mohli byť na jeho vstupe. Referencie vyjadríme dvomi spôsobmi, buď *menom premennej* – cestou k nej, alebo *číslom vstupnej premennej* a jej typom.

Prvá alternatíva odkazu na premennú, jej menom, funguje tak, že definujeme cestu k premennej. Tá sa podobá na cesty, ktoré používame k adresovaniu súborov v bežných súborových systémoch. Experimenty si predstavíme ako adresáre, premenné ako súbory. Pravidlá odkazov na premennú:

- Oddeľovač experimentov na ceste je znak '`/`'.
- Každý odkaz začína znakom '`$`', za ním nasleduje cesta.
- K adresovaniu konkrétnej premennej experimentu už nepoužijeme '`/`'. Odkaz má tvar `<meno experimentu>.<meno premennej>`.
- Máme možnosť posunúť sa o úroveň vyššie, použijeme experiment '`..`', podobne ako v cestách k súborom. Znak `.` je tiež povolený, avšak nie je potrebný.
- Absolútne cesty začínajú znakom '`/`'.
- Relatívne cesty nezačínajú '`/`'. Za začiatok cesty potom považujeme odkazujúci sa experiment.
- Povolené sú len odkazy na vstupné premenné experimentu, teda na premenné, ktoré vznikli na ceste z koreňa. Je chybou odkázať sa na premenné z inej cesty, prípadne na premenné odkazujúceho sa experimentu alebo jeho potomkov.

Napríklad cesta `$/composite_root/loop_1/loop_2.value` označuje premennú `value` experimentu `loop_2`, čo je v tomto prípade zrejme hodnota iterátora cyklu. Uzol `loop_2` má na ceste do koreňa `composite_root` ešte rodiča `loop_1`. Ďalšie príklady nájdeme v tabuľke 4.1.



Obrázok 4.1: Hierarchický experiment – príklad odkázania sa na premenné cestou k nim. Prvý riadok určuje meno experimentu, ďalšie sú mená jeho výstupných premenných.

Referencia	*Experiment*	*Správnosť*
`$/exp_root/exp_1_1.0` `$/exp_root.value`	`exp_2_1`, `exp_2_2`, `exp_3_1`, `exp_3_2`	Správne.
`$/exp_root/value`	Nesprávne, na oddelenie mena používame znak '`.`'.	
`$../exp_1_1.0`	`exp_1_2`	Nesprávne, `exp_1_1.0` nie je na ceste z koreňa do `exp_1_2`, nemôže byť vstupom.

Tabuľka 4.1: Príklady odkazovania sa na premenné cestou k nim.

Druhý spôsob odkazovania sa na vstupnú premennú vyjadríme len jednoduchou špecifikáciou, že na danom vstupe experimentu, očakávame hodnotu zvoleného typu. Vstupné premenné sú číslované od jednotky a ich poradie určené postupom, v akom sú vytvárané na ceste k experimentu. Výstupy na ceste od koreňa sa postupne skladajú za seba a tvoria vstup nasledujúceho experimentu. Zápisu toho odkazu má tvar:

{<typ premennej>@<poradie na vstupe>}.

Napríklad {`double@1`} vyjadruje očakávanie, že prvá vstupná premenná má mať typ {`double`}. Viac ukážok sa nachádza v tabuľke 4.2.

Typy vstupov	*Zadaná špecifikácia*	*Správnosť*
`int, std::string`	`{int@1}`	Správne, odkazujeme sa na prvú premennú.
`int, std::string`	`{int@0}`	Nesprávne, vstup je číslovaný od jednotky.
`int, std::string`	`{string@2}`	Nesprávne, nesedia identifikátory typov.
`vector<int>`	`{vector< int >@1}`	Nesprávne, identifikátor typu musí byť rovnaký (obsahuje medzery naviac).

Tabuľka 4.2: Príklady zadania požiadaviek na vstupné premenné.

Nakoľko sme v tomto projekte zvolili objektový prístup, všetky hodnoty sú za behu testu uchovávané v polymorfných objektoch knižnice *Experiments*. Pokiaľ sa chceme „ručne“ odkázať na hodnotu premennej, napríklad kvôli jej špecifickým vlastnostiam a následnú prácu s ňou, musíme poznať jej typ počas behu testu a použiť pretypovanie. Ak je pretypovanie neúspešné, vyvolá sa výnimka a program skončí chybovým hlásením.

Aby sme sa podobným situáciám vyhli, v programe *ExperimentsCreator* je zabudovaná jednoduchá typová kontrola. Pokiaľ zadáme odkaz na vstupnú premennú druhým spôsobom, pomocou špecifikácie, typ špecifikácie musí súhlasiť s typom vstupujúcej premennej na danej pozícii.

Program vykonáva len veľmi zjednodušenú kontrolu. Pri použití šablón, referencií a ukazovateľov, musia byť identifikátory typov napísané úplne rovnako. Nie sme schopní overiť napríklad klauzule `typedef` pri šablónach, dokonca neprevádzame žiadne parsovanie zadaného typu okrem určenia hĺbky odkazu pri ukazovateľoch. Príklad, akým spôsobom funguje typová kontrola, nájdeme v tabuľke 4.2 spolu s ukážkami adresovania premenných.

4.2 Experiment „Composite“

Experiment *Composite* je vnútorný uzol na vytváranie stromovitej štruktúry. Pretože všetky štandardné experimenty sú implementované tak, že môžu mať najviac jedného potomka, musíme mať nejaký spôsob rozvetvenia ciest a práve k tomu je tento experiment určený. Jeho úlohou je spustenie všetkých potomkov v takom poradí, v akom boli zapojení a s takým vstupom s akým je *Composite* zavolaný.

Služby tohoto experimentu oceníme napríklad, keď chceme inicializovať objekty mimo hlavnej testovacej cesty. Inicializácia bude tvoriť samostatnú cestu, vykoná sa skôr a jej výsledky nebudeme uchovávať. Ak chceme vykonať viac testov podobného typu, často na rôznych objektoch, ale s rovnakým vstupom, je vhodné, aby sme volania testov pre dané vstupy mali na tej istej úrovni, zapojené za experimentom typu *Composite*.

Jeho použitie spočíva vo vytvorení *zoznamu experimentov*. Ak máme experimenty, ktoré nemajú žiadne spoločné premenné, *Composite* môže byť aj koreňom a slúžiť na vytvorenie sady testov.

Často je dôležité poradie, v akom sú potomkovia tohoto experimentu vykonaní. Preto, ak *Composite* obsahuje viac ako dvoch potomkov, za ich menom nájdeme vypísané poradie ich volania. Poradie meníme rozpojením a opätovným spojením. Posledný pridaný experiment je vykonaný nakoniec.

Experiment type	*composite*
Typ knižnice	statická
Hlavičkový súbor	`CompositeExperiment.h`
Súbor s knižnicou	`Experiments.lib`

Tabuľka 4.3: Prehľad vlastností experimentu *Composite*

4.3 Experiment „Loop“

Experiment loop je jednoduchý cyklus, ktorý umožňuje vygenerovať hodnoty zo zadaného intervalu, pričom vieme ovplyvniť minimálnu, maximálnu hodnotu a krok. Pri nastavení týchto parametrov máme dve alternatívy. Buď sa budú odkazovať na vstupnú premennú alebo budú obsahovať konštantu. Na premennú sa odkazujeme *jedine cestou k nej*, prvým spôsobom popísaným v kapitole 4.1 na strane 22.

Experiment pri každom zavolaní načíta hodnoty premenných, pokiaľ sú definované odkazom, a vráti počiatočnú hodnotu. Poslednú výstupnú hodnotu v každom priebehu zvyšuje o zadaný krok, kým neprekročí zvolený strop. Maximum je považované za platnú hodnotu a môže byť vrátené.

4.4 Experiment „Collector“

Pri rôznych testoch je určite vhodné mať veľmi jednoduchý „zberač“ dát fungujúci takmer bez konfigurácie, práve k tomu je určený experiment „Collector“. Poskytuje ukladanie dát do textového súboru typu SSV či CSV[6], pričom oddeľovač hodnôt

Experiment type	*loop*
Start	Počiatočná hodnota.
Stop	Konečná hodnota (vrátane).
Step	Krok cyklu.
Typ knižnice	statická
Hlavičkový súbor	`LoopExperiment.h`
Súbor s knižnicou	`BasicExperiments.lib`

Tabuľka 4.4: Prehľad vlastností experimentu *Loop*

si môžeme sami definovať. *Collector* funguje tak, že všetky hodnoty, na ktorých je experiment zavolaný, sú zapamätané a neskôr zapísané do špecifikovaného súboru.

Collector nemodifikuje, nefiltruje ani neagreguje vstupné dáta, zbiera hodnoty všetkých výstupných premenných jeho predkov. Každý bod, v ktorom je požadované vykonanie experimentu, je uložený na jednom riadku. Jednotlivé hodnoty na ňom sú oddelené zadaným reťazcom, ten určíme vlastnosťou *Separator*.

Na začiatku súboru, na prvých dvoch riadkoch, je popis dát. Prvý riadok obsahuje reťazec:

```
Data set: <meno sady>, generated: <dátum vykonania testu>.
```

Druhý riadok obsahuje zoznam mien premenných, ktorých hodnoty budú týmto experimentom zachytené. Identifikátory premenných budú oddelené tým istým reťazcom, ako výstupné hodnoty.

Počet hodnôt na každom riadku musí byť rovnaký, žiadna hodnota nemôže byť vynechaná, čo je aj počas behu testovacieho kódu kontrolované. Kontrolované sú aj typy vstupných hodnôt. Takéto správanie je možné vyvolať naprogramovaním vlastných, neštandardne sa správajúcich experimentov, avšak v tomto prípade je nežiadúce.

Experiment type	*collector*
File name	Súbor, do ktorého budú uložené zozbierané dáta.
Separator	Oddeľovač hodnôt dát na riadku.
Typ knižnice	statická
Hlavičkový súbor	`CollectorExperiment.h`
Súbor s knižnicou	`BasicExperiments.lib`

Tabuľka 4.5: Prehľad vlastností experimentu *Collector*

4.5 Experiment „Object creator“

Počas testovania softvéru, najmä keď volíme prístup zvaný unit testing, testujeme samostatne nejaké moduly, funkcie alebo objekty. Skúšame ich správnosť na rôznych vstupoch, overujeme, či vrátia očakávané výstupy. *Object creator* umožní vytvorenie ľubovoľného objektu, ktorý bude ďalej distribuovaný ako výstupná premenná. Vytvorený objekt môžeme ďalej skúmať ľubovoľným spôsobom, môžeme na ňom volať ľubovoľné metódy, sledovať jeho výstupy a uplatniť princíp podobný ako pri „unit testingu“.

Použitie experimentu spočíva v správnom nastavení vlastnosti *Constructor*. Jej nastavením zvolíme, ktorým konštruktorom a akým spôsobom bude objekt vytvorený. Výstupom experimentu nie je objekt samotný, ale ukazovateľ na vytvorený objekt. Objekt sa vytvára pomocou operátora `new` na haldu.

Hodnota vlastnosti *Constructor* vyzerá podobne ako bežné volanie konštrukturu objektu v C++. Začína sa pomenovaním typu. Meno typu nemusí byť len „čistý“ identifikátor, preto nemáme žiadne obmedzenie na pomenovanie typu. Vďaka vypusteniu kontrol je jednoduché použitie šablóny alebo menných priestorov. Pri odkazovaní sa na vytváraný objekt, je nutné naďalej používať tento reťazec ako typ premennej.

Po identifikátore typu nasledujú ľavé zátvorky a zoznam parametrov konštruktora oddelených čiarkou. Parametrom môže byť ľubovoľný výraz, môžeme sa odkazovať na už existujúce premenné tak, ako je popísané v sekcii 4.1.

Napríklad `int(1)` vytvorí na halde premennú typu `int` s hodnotou jeden. Pokiaľ budeme požadovať a predpokladať, že prvá a druhá vstupná premenná boli typu `int`, použitím `int({int@1} + {int@2})` vytvoríme premennú obsahujúcu súčet prechádzajúcich hodnôt.

Výstupná premenná má meno `0`, je to ukazovateľ na vytvorený objekt, teda typu `<typ objektu> *`.

Počas kompilovania je dôležité, aby mal kompilátor k dispozícii všetky definície, ktoré potrebuje. Preto, pokiaľ je potrebné, nastavíme vlastnosť *Library* tak, nech odkazuje na príslušný hlavičkový súbor.

4.6 Experiment „Method caller“

Experiment *Method caller* umožňuje vyhodnotenie jedného akéhokoľvek výrazu alebo zavolanie ľubovoľnej funkcie. S týmto experimentom sme schopní emitovať najviac jednu výstupnú premennú – výsledok tohoto výrazu. Táto premenná ma meno 1.

Hodnota vlastnosti *Method* obsahuje definíciu výrazu, ktorý chceme vyhodnotiť. Najprv v zložených zátvorkách uvedieme jeho návratový typ. Potom definujeme sa-

Experiment type	*object_creator*
Constructor	Konštruktor použitý k vytvoreniu objektu.
Library	Knižnica potrebná k použitiu objektu.
Typ knižnice	generovaný kód
Hlavičkový súbor	`CollectorExperiment<meno experimentu>.h`
Súbor s knižnicou	`CollectorExperiment<meno experimentu>.cpp`

Tabuľka 4.6: Prehľad vlastností experimentu *Object creator*

motný výraz. V ňom sa môžeme odkazovať na predchádzajúce premenné podobne, ako v predošlom experimente. Ak zadáme ako výstupný typ `void`, experiment nebude generovať žiadnu premennú. V tomto prípade je jeho cieľom zrejme vykonanie nejakej úlohy s bočnými efektmi. Vhodným využitím tohoto princípu je vykonanie algoritmu, pričom výstup získame zavolaním ďalšej metódy na tomto objekte.

Napríklad výraz `{int} {int@1} + {int@2}` vytvorí súčet prvých dvoch premenných, `{int} *{int *@1}` dereferuje hodnotu ukazovateľa – prvej premennej.

Vlastnosť *Library*, ako v predchádzajúcom prípade, nám umožní nastaviť hlavičkový súbor potrebný k preloženiu výrazu.

Experiment type	*method_caller*
Method	Vyhodnocovaný výraz.
Library	Knižnica potrebná k preloženiu výrazu.
Typ knižnice	generovaný kód
Hlavičkový súbor	`CollectorExperiment<meno experimentu>.h`
Súbor s knižnicou	`CollectorExperiment<meno experimentu>.cpp`

Tabuľka 4.7: Prehľad vlastností experimentu *Method caller*

4.7 Experiment „Code block“

Experiment *Code block* vykoná používateľsky definovaný kód. Cieľom je prispôsobenie si testov bez toho, aby používateľ musel komplikovanejším spôsobom tvoriť vlastné experimenty. Programovanie takýchto súčastí nadobudne na význame hlavne v prípade ich častejšieho použitia.

Zdrojový kód, ktorý bude vykonaný počas behu aplikácie, zadávame vo vlastnosti *Code*. Prvý riadok definuje výstupné premenné vytvárané týmto experimentom, má tvar popísaný na obrázku 4.2.

{output_types: <identifikátory typov, oddelené čiarkou>}

Obrázok 4.2: Klauzula output_types experimentu *Code block*

Počet, poradie a typ výstupných premenných sa už ďalej nesmie meniť. Mená sú premenným pridelené automaticky, sú číslované od jednotky v poradí, v akom sú zadané v tejto klauzule.

Po tomto riadku môžeme vkladať ľubovoľný C++ kód, pričom sa smieme odkazovať na vstupy tohoto experimentu rovnako, ako v prípade prvku *Method caller*.

V ktorejkoľvek časti kódu môžeme k zavolaniu potomka využiť klauzulu output, do ktorej zadáme zoznam výstupných hodnôt. Ich typy sa musia zhodovať, musia byť automaticky konvertovateľné na typy zadané v klauzule output_types. V opačnom prípade, počas kompilovania vytvorených zdrojových kódov, obdržíme chybu prekladu. Počet výstupov sa kontroluje automaticky pri zadávaní vlastnosti a pri zlom nastavení sme upozornení chybovým hlásením.

{output: <zoznam hodnôt, oddelené čiarkou>}

Obrázok 4.3: Klauzula output experimentu *Code block*

Experiment type	*code_block*
Code	Vlastný zdrojový kód.
Library	Knižnica potrebná k preloženiu zadaného zdrojového kódu.
Typ knižnice	generovaný kód
Hlavičkový súbor	CodeBlock<meno experimentu>.h
Súbor s knižnicou	CodeBlock<meno experimentu>.cpp

Tabuľka 4.8: Prehľad vlastností experimentu *Code block*

4.8 Experiment „Data collector“

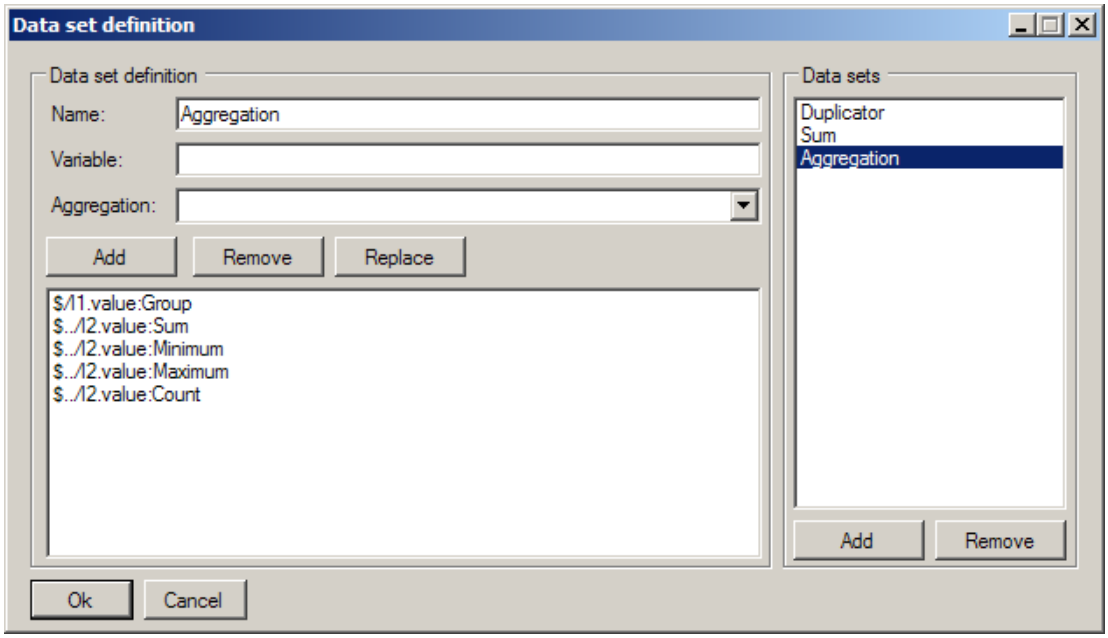
Data collector je prispôsobiteľný „zberač“ výstupných hodnôt. Povoľuje výstup do XML[7], ale aj do CSV súboru rovnako, ako je to v prípade experimentu *Collector*. Dajú sa nastaviť agregácie a filtrovanie dát, sledovať rôzne veličiny a sady výstupov.

Ukladanie výstupných dát funguje tak, že si definujeme dátové sady, ktoré sa ukladajú nezávisle. Rôzne sady môžu zdieľať rovnaké premenné. Každá z nich obsahuje len tie premenné, ktoré si používateľ zvolí, máme možnosť filtrovať hodnoty.

Všetky premenné, zbierané v spoločnej sade, musia pochádzať z jednej cesty experimentov. Odkazovať na ne sa môžeme len spôsobom ich úplného pomenovania, teda zadaním cesty absolútnej alebo relatívnej.

Na jednotlivé premenné môžeme aplikovať agregačné funkcie – suma, priemer, počet, maximum a minimum. Takéto premenné sa však už nesmú vyskytovať bez agregácie. Na každú veličinu môžeme použiť aj viac agregácií. Taktiež sa dajú vytvárať kolekcie len z čistých hodnôt, bez agregácie. V tomto prípade sa sada správa ako funkcia. Posledná definovaná premenná je chápaná ako výsledok v bode určenom hodnotami predchádzajúcich premenných. Nie je možné mať mapovanie na dve hodnoty. Ak počas testu táto situácia nastane, výstupná sada vyvolá výnimku. Tento mód využijeme najmä v prípade, že sledujeme hodnotu parametra, ktorý je výsledkom zadaných vstupov a tie sa nikdy nebudú opakovať.

Dátovú sadu tohoto experimentu definujeme tak, že vyvoláme dialóg na nastavenie vlastnosti *Data sets*, obrázok 4.4.



Obrázok 4.4: Definícia sady dát experimentu *Data collector*

V ňom definujeme jednotlivé dátové sady. Každá sada má svoje meno, ktoré musíme vyplniť. Obsahuje aj zoznam premenných s ich agregačnými nastaveniami. Odkazujeme sa ne z kontextu práve editovaného experimentu *Data collector*. Všetky premenné v sade musia pochádzať z jedinej cesty začínajúcej v koreni. Agregačné nastavenie môžeme zadať ľubovoľné, čo využijeme v prípade používateľsky definovanej agregácie. Naopak, nastavenie *Group* znamená, že táto premenná bude v skupine neagregovaných premenných. Potom ju už nie je možné v práve definovanej sade

agregovať.

Správne nastavenú premennú pridáme do zoznamu tlačítkom *Add*. Aktuálne označenú premennú zo zoznamu odoberieme tlačítkom *Remove*. Po označení a editovaní niektorej položky vykonané zmeny potvrdíme tlačítkom *Replace*.

So sadami, pracujeme podobným spôsobom. Ak sme sadu nadefinovali, pridáme ju tlačítkom *Add*, ktoré sa nachádza pod zoznamom sád. Označenú sadu zmažeme tlačítkom *Remove*.

Správnosť zadaných sád, vzhľadom na mená premenných, sa kontroluje až pri validácii experimentu, t.j. pri ukladaní alebo užívateľom vyvolanej kontrole. Takéto správanie je spôsobené tým, že experiment *Data collector* môžeme definovať skôr, ako zapojíme všetky experimenty. Prípadne nechceme byť obmedzení počas definovania pretože niektoré premenné ešte nemusia byť prítomné apod. Ak je vlastnosť *Data sets* nastavená zle, jej textová reprezentácia zobrazená v editore vlastností bude obsahovať chybové hlásenie.

Vlastnosť *File name* určuje výstupný súbor, zostávajúcim nastavením *Output type* prepíname medzi CSV súborom (*RawOutput*) a XML súborom (*XmlOutput*).

Experiment type	*data_collector*
Data sets	Zbierané sady dát. V GUI je vypísaný len ich počet.
File name	Výstupný súbor, obsahuje zozbierané sady dát.
Output type	Zvolený typ výstupu: XML alebo CSV.
Typ knižnice	generovaný kód
Hlavičkový súbor	`DataCollector<meno experimentu>.h`
Súbor s knižnicou	`DataCollector<meno experimentu>.cpp`

Tabuľka 4.9: Prehľad vlastností experimentu *Data collector*

4.9 Experiment „Directory lister“

Doplnkový experiment prechádza adresár a vracia súbory, ktoré vyhovujú zadanej maske. Maska podobne, ako pri bežných príkazoch príkazového riadka, povoľuje znaky „*“ a „?“ s rovnakým významom.

Výstupom je premenná, `file`, typu `string`. Obsahuje len meno súboru, nie celú cestu. Ďalším výstupom je premenná, `directory`.

Experiment type	*directory_lister*
Directory	Prechádzaný adresár.
Mask	Maska platných, „vypísaných“, súborov.
Typ knižnice	statická
Hlavičkový súbor	`DirectoryLister.h`
Súbor s knižnicou	`BasicExperiments.lib`

Tabuľka 4.10: Prehľad vlastností experimentu *Directory lister*

4.10 Experiment „Line enumerator“

Doplnkový experiment, ktorého úlohou je výpis všetkých riadkov zvoleného súboru. Súbor si volíme nastavením hodnoty vlastnosti *File*. Jej obsahom môže byť aj relatívna cesta, avšak odkazovať sa budeme na súbory v pracovnom adresári spusteného testovacieho kódu.

Prvým výstupom je vstupný súbor, premenná `file` typu `string`. Výstupná premenná pomenovaná `line`, typu `string`, už neobsahuje znak konca riadku.

Experiment type	*LineEnumerator*
File	Súbor, ktorého riadku budú výstupom.
Typ knižnice	statická
Hlavičkový súbor	`LineEnumerator.h`
Súbor s knižnicou	`BasicExperiments.lib`

Tabuľka 4.11: Prehľad vlastností experimentu *Line enumerator*

4.11 Experimenty typu „Exponential loop“

Experimenty typu „Exponential loop“ nám umožňujú vytvoriť logaritmický cyklus. Generujú hodnoty, ktoré sú prvkami zadanej rastúcej geometrickej postupnosti. Pretože typ `size_t` je na podobné účely pomerne obmedzený a jedná sa o rast predovšetkým v exponente, môžeme si vybrať aj cyklus, ktorého iterovaná premenná je typu `double`. Implementácia tohoto experimentu je šablóna[8] a teda je možné využiť akýkoľvek zvolený typ, po definovaní si vlastného experimentu.

Počas nastavenia cyklu máme dve alternatívy, akým spôsobom ho ukončiť. Prvá, bezpečnejšia, umožní vykonať zadaný počet krokov. V prípade pretečenia máme garantované, že cyklus skončí po predpokladanom počte iterácií. Tento „mód“ zapne-

me nastavením vlasnosti *Steps count usage* na *true* a zadaním *Steps count*. Druhá možnosť ukončí cyklus, keď hodnota iterátora prekročí zadanú hodnotu *Ending value*, pričom vlastnosť *Steps count usage* je nastavená na *false*.

Pri zadávaní vlastností *Starting value*, *Ending value*, *Quotient* a *Steps count* sa môžeme, podobne ako v prípade experimentu *LoopExperiment*, odkázať na hodnoty vstupných premenných, nemusia byť konštantné.

Experiment type	*DoubleExponentialLoop*, *PositiveExponentialLoop*
Ending value	Maximálna možná hodnota, vrátane.
Quotient	Kvocient geometrickej postupnosti.
Starting value	Počiatočná hodnota, vrátane.
Steps count	Počet krokov cyklu.
Steps count usage	Príznak určujúci, či k zastaveniu použijeme *Steps count* alebo *Experiment type*.
Typ knižnice	statická
Hlavičkový súbor	`ExponentialLoop.h`
Súbor s knižnicou	`BasicExperiments.lib`

Tabuľka 4.12: Prehľad vlastností experimentu typu *Exponential loop*

Kapitola 5

Spracovanie výstupných dát

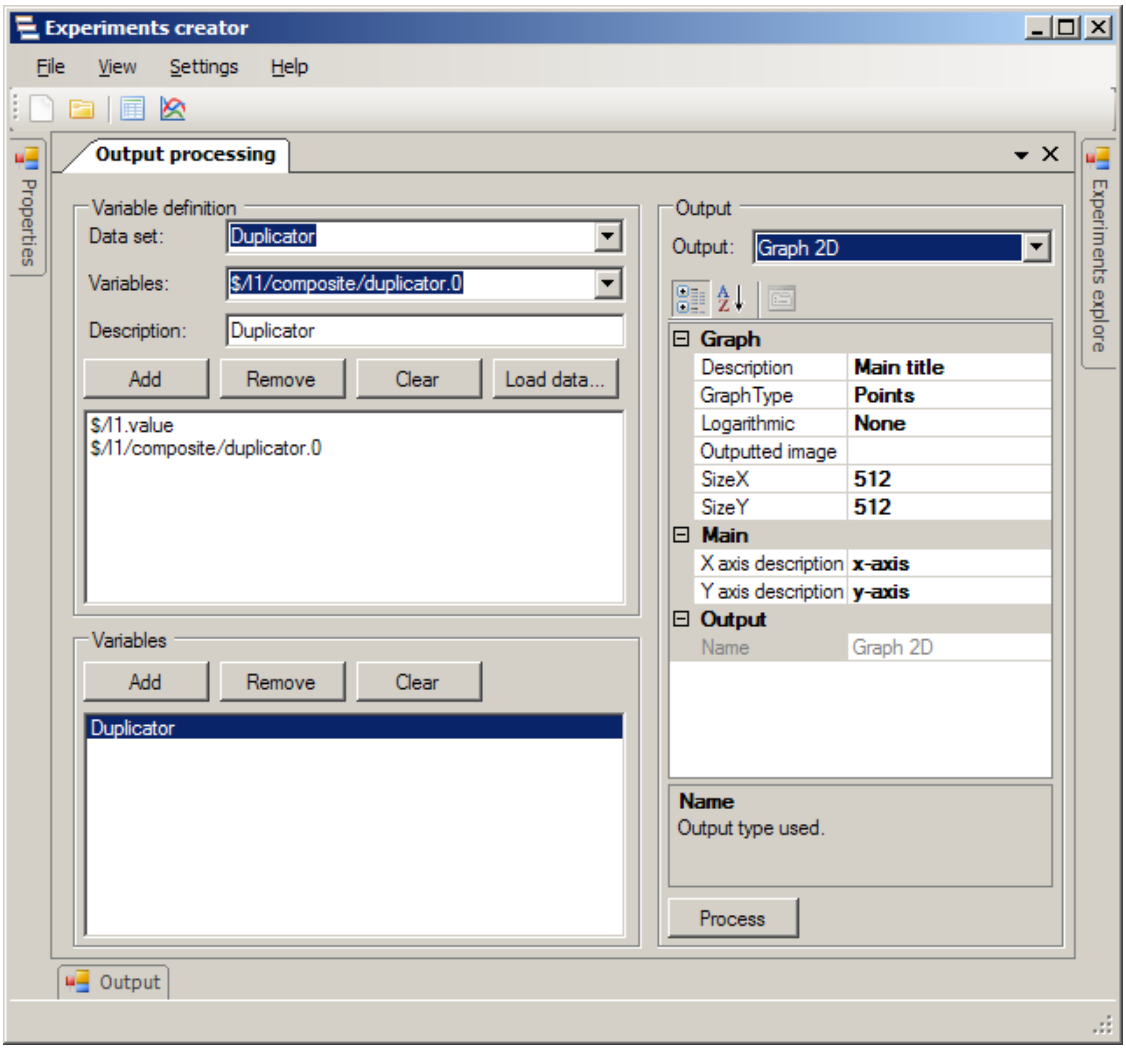
Po vytvorení výstupných dát je výhodné mať možnosť ich prezentácie. Program *ExperimentsCreator* nám poskytuje dve základné, veľmi jednoduché, alternatívy. Môžeme vytvoriť tabuľku vo formáte HTML[9] alebo Latex[10], prípadne dvojrozmerný graf. V poslednom prípade sa v skutočnosti vygeneruje skript programu *R-project*[11] vytvárajúci požadovaný graf. Tento skript môžeme ďalej upraviť podľa vlastných požiadaviek.

Výstupné dáta spracujeme použitím formulára, ktorý nájdeme na obrázku 5.1. Tento dialóg nám pomáha definovať prezentované veličiny. Umožňuje jednoduchým spôsobom využiť zoznam premenných a dátových sád, ktoré nájde vo zvolenom výstupnom súbore. Podporované formáty súborov sú tie, ktoré generujú experimenty *Collector* a *DataCollector* v oboch podobách – XML aj oddelené hodnoty. Opisovaného sprievodcu vyvoláme v menu *File* položkou *Output processing*.

5.1 Definovanie prezentovaných dát

Ako prvé pri definícií prezentácie výstupu, je vhodné, ak vyberieme súbor obsahujúci zozbierané informácie. Tento postup je výhodný najmä preto, lebo po vybratí vstupného súboru sa načítajú zozbierané sady dát a zoznamy ich premenných. Potom sme schopní jednoduchým spôsobom nastaviť sledované premenné – vyberieme ich z ponúknutého zoznamu. Súbor s dátami vyberieme tlačítkom *Load data...* alebo jeho ekvivalentom v paneli nástrojov. Pokiaľ zvolíme súbor CSV, musíme si byť istí, že ako *oddeľovač je* použitá *medzera*.

Ďalšou úlohou je zvoliť sledované veličiny. Predstava grafu alebo tabuľky je taká, že pre každú sledovanú veličinu zvolíme dve výstupné premenné. Prvá sa bude nanášať na x-ovú os a druhá v poradí na os y. Pritom predpokladáme, že premenná, ktorú sme vybrali ako druhú, je funkciou premennej nanesenej na os x. Pri porušení tohoto predpokladu, sme pri spracovaní dát informovaní chybovou správou.



Obrázok 5.1: Formulár určený k spracovaniu výstupných dát.

Takto môžeme nadefinovať viac zobrazených veličín a teda údaje týmto spôsobom porovnávať. Každá sledovaná veličina sa v grafe zobrazí ako nová závislosť. K jej zobrazeniu máme na výber vykreslenie pomocou bodov alebo pomocou čiar spájajúcich hodnoty v zadaných argumentoch. Do tabuľky sa každá veličina premietne ako nový riadok.

Spôsob zadania sledovanej veličiny je nasledovný:

1. Vyberieme dátovú sadu obsahujúcu obe premenné veličiny, pole *Data set*.
2. Ako prvú zvolíme nezávislú premennú nanášanú na os x, pridáme ju do zoznamu tlačítkom *Add*.
3. Ako druhú zvolíme premennú nanášanú na os y.

4. Pomenujeme sledovanú veličinu, pole *Description*.

Takto vytvorenú veličinu pridáme do prezentácie tlačítkom *Add* v skupine *Variables*.

K tomu, aby sme mohli vytvoriť spracovaný výstup, je ešte nutné zadať spôsob, akým bude prezentovaný. To urobíme zvolením hodnoty v poli *Output*. Následne môžeme ovplyvniť jednotlivé spôsoby prezentácie nastavením ich vlastností. K vytvoreniu výstupu použijeme tlačítko *Process* alebo jeho ekvivalent v paneli nástrojov.

5.2 Výstupy typu „*Table*“

Výstupy typu „*Table*“ vytvárajú z predložených dát tabuľku vo formáte HTML alebo Latex. Ako vidíme na príklade 5.2, tabuľka obsahuje popisy pre jednotlivé veličiny. Hodnoty osi x sa nachádzajú v prvom riadku, hodnoty sledovaných veličín sú v stĺpcoch pod nimi.

	0	1	2	3	4	5
$y = 2x$	0	2	4	6	8	10

Obrázok 5.2: Príklad tabuľky vygenerovanej programom *ExperimentsCreator*.

5.3 Výstup „*Graph 2D*“

Výstup „*Graph 2D*“ generuje skript programu *R-project*. Pri použití tohoto typu výstupu predpokladáme, že používateľ má nainštalovaný program *R-project* a vie ho ovládať na základnej úrovni.

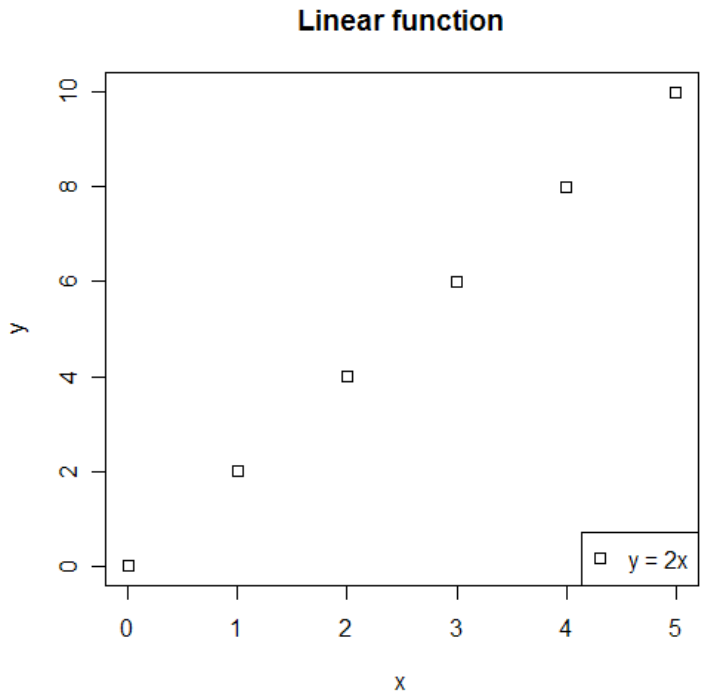
Vytvorený skript môžeme následne použiť k vygenerovaniu požadovaného grafu, napríklad príkazom `RScript.exe graph.r`. Je pravdepodobné, že cesty k súborom `RScript.exe` a `graph.r` bude nutné uviesť presnejšie. Súbor `graph.r` označuje skript vygenerovaný ako grafovú prezentáciu výstupu.

Program `RScript.exe` je súčasťou balíka *R-project*, nájdeme ho v inštalačnom adresári, v zložke `bin`. `RScript.exe` vykoná požadovaný skript, ktorý dostane ako parameter na príkazovom riadku. Táto aplikácia umožňuje viac nastavení pri tvorbe grafiky, ako môžeme v prostredí ovplyvniť. Vygenerovaný skript si používateľ môže ďalej uspôsobiť podľa jeho predstáv.

Nastavenia, ktoré môžeme ovplyvniť pri generovaní grafu:

- *Description*: Titulok grafu.

- *Graph type*: Typ grafu určuje, či vykreslené hodnoty budú len body (*Points*) alebo body spojené čiarou (*Lines*).
- *Logarithmic*: Osi, na ktorých chceme použiť logaritmické merítko. Hodnoty na týchto osiach musia byť kladné.
- *Outputted image*: Cesta k súboru, ktorý bude obsahovať vytvorený obrázok. Použitý formát je *PNG*[12].
- *SizeX*: Šírka vytvoreného obrázku v pixloch.
- *SizeY*: Výška vytvoreného obrázku v pixloch.
- *X axis description*: Popisok hodnôt na osi x.
- *Y axis description*: Popisok hodnôt na osi y.



Obrázok 5.3: Príklad grafu vygenerovaného programom *ExperimentsCreator*.

Kapitola 6

Implementácia

V tejto kapitole popíšeme fungovanie testovacieho programu, jeho základné triedy a ich spoluprácu, teda knižnicu *Experiments*. Ďalej načrtneme spôsob fungovania programu *ExperimentsCreator*, prepojenie experimentov napísaných v C++ a experimentov v tomto programe. S nadobudnutými znalosťami budeme môcť v nasledujúcej kapitole vytvárať vlastné experimenty.

6.1 Knižnica *Experiments*

Ako už bolo v predchádzajúcich častiach niekoľkokrát zmienené, práve knižnica *Experiments* riadi beh testu a zbiera generované dáta. Tieto úlohy sú rozdelené na tri základné časti a jednu pomocnú:

1. Uchovanie jedinej hodnoty, abstraktná trieda *Value*
2. Uchovanie vytvorených výsledkov, abstraktná trieda *DataStorage*
3. Riadenie behu hierarchického experimentu, generovanie dát, abstraktná trieda *Experiment*
4. Podporné úlohy, najmä menný priestor *Utils*[1].

Celá knižnica *Experiments* je dodaná v dvoch podobách – ako skompilované statické knižnice a ako zdrojový kód. Obe nájdeme v adresári `ExperimentsLibrary`, ten sa nachádza priamo v inštalačnom adresári programu. Riešenie knižnice je rozdelené na tri časti:

- Časť *Variables* obsahuje sadu tried, ktoré pracujú s veličinami a premennými. Jedná sa najmä o triedy *Value* a *DataStorage*.

[1] Jedná sa najmä o triedu *SmartPointer*, ktorá sa používa pri správe pamäti. Vysvetlenie princípov a použitia nájdeme v dodatku D.

- Projekt *Experiments* obsahuje definíciu experimentu, triedu *Experiment* a implementáciu experimentu *Composite*. V tejto časti sú tiež implementované triedy uľahčujúce definíciu nových experimentov – *SimpleInnerExperiment*, *IterativeInnerExperiment* a *LeafExperiment*.

- Projekt *BasicExperiments* definuje základnú sadu experimentov. Obsahuje experimenty popísané v kapitole 4 tejto práce.

6.2 Uchovanie generovanej hodnoty

V celom hierarchickom experimente dochádza k šíreniu generovaných hodnôt, je nutné ich predávať medzi rôznymi experimentmi, pamätať si vygenerované výstupy. Hodnoty rôznych typov sa musia líšiť v poskytovaných operáciách, v ich správaní. K tomuto účelu môžeme využiť dva mechanizmy, ktoré nám programovací jazyk ponúka – statický alebo dynamický polymorfizmus[13, 14]. V knižnici *Experiments* sme sa rozhodli pre druhú možnosť[2].

Prvá alternatíva má veľkú výhodu – zachytenie typovej informácie. Nestrácame typ žiadnej premennej, môžeme plne využiť typový mechanizmus a typovú kontrolu jazyka. Za túto výhodu by sme v prípade objektového programovania zrejme zaplatili zložitosťou experimentov a tried na zbieranie dát.

Táto skutočnosť je spôsobená princípom, akým experimenty fungujú. Čím hlbšie je experiment vnorený, tým viac parametrov prijíma. Pokiaľ by sme sa rozhodli pre mechanizmus statického polymorfizmu, každý experiment by musel mať metódu `run`[3] uspôsobenú tak, aby sa zhodovala s jeho použitím v hierarchii. Museli by sme generovať každý experiment zvlášť a mať premyslený taký spôsob ich zapájania, aby sme neprišli o prístup k špecializovanej metóde. Zrejme by sa jednalo o pomerne komplikované použitie šablón a techniky nazvanej typelists[14]. V skratke povedané, pre každé použitie experimentu by sme mali samostatnú inštanciu šablóny triedy, samostatný typ. Experiment by mal modifikovanú metódu `run` tak, aby fungovala s požadovanými parametrami – vstupnými premennými experimentu.

Druhou alternatívou statického polymorfizmu, k objektovému programovaniu, by bolo vygenerovať celý experiment naraz, napríklad ako jednu funkciu. To znamená, že experimenty by neboli oddelené v triedach, našli by sme ich v jednom zdrojovom kóde vnorené tak, ako boli vytvorené. Tu narážame napríklad na problém s konfliktmi názvov premenných, možnosťou vzájomne sa ovplyvňovať. Pri použití používateľského kódu by bolo obtiažne zabrániť zmene hodnoty, ktorá sa netýka len priamo definovaného experimentu.

[2] V knižnici *Experiments* sme použili objektovo orientované programovanie a jazyk C++. Cieľom bolo testovanie programov v naprogramovaných v C++, mohli sme zvoliť aj použitie jazyka C.

[3] Metóda `run` vykoná úlohu experimentu.

Pri dynamickom polymorfizme prichádzame naopak o typovú kontrolu hodnôt, ktoré sú parametrom experimentov. Túto stratu nahradíme až kontrolou za behu. Na rozdiel od predchádzajúcej verzie, získavame jednoznačnú signatúru metódy pre spustenie experimentu a vďaka tomu môžeme jednoducho použiť virtualitu k uspôsobeniu experimentov. Ďalej je nutné mať rovnaký spôsob pre uchovanie akejkoľvek hodnoty, ktorá je parametrom experimentu. Môže sa jednať o objekt, hodnotu iterátora, ukazovateľ, typ nie je dopredu známy. Práve k tomuto účelu je určená trieda *Value*, tá uchováva zvolenú hodnotu.

Uchovanie hodnoty musí byť pre všetky typy rovnaké. Chceme pracovať s jednotným rozhraním. Použitím takéhoto rozhrania strácame typovú informáciu. Údaj o type je potrebný k tomu, aby sme mohli s rôznymi hodnotami pracovať a vykonávať nad nimi pre ne špecifické operácie. Nakoľko sme o typovú informáciu prišli, neostáva nám nič iné, ako si pretypovaním[15] vynútiť vrátenie uchovanej hodnoty. To je ukryté vo funkcii `value_cast`. Tá v prípade neúspechu, napríklad ak poskytneme iný typ hodnoty ako chceme získať, zlyhá výnimkou `std::invalid_argument`.

Trieda *Value* je polymorfná obálka hodnoty. Umožní nám jej jednotné a bezpečné predávanie medzi rôznymi volaniami experimentov a jej správne uchovanie nezávisle na type pamätanej veličiny. Obsahuje metódy, ktoré ju vedia skopírovať (`clone`) a taktiež definuje konverziu na typ `std::string` (`toString`). Táto konverzia sa používa pri výpise hodnoty.

6.3 Kontrola typu hodnoty

K overeniu zhody (vo všeobecnejšom prípade kompatibility) typov je určená trieda *ValueDescription*, metódy `equals` alebo `equalDescriptions`. Je zrejmé, že jej inštancie sa používajú pri typovej kontrole. Hlavne pri ukladaní dát chceme mať všetky zapamätané body zložené z hodnôt rovnakých typov. To je spôsobené možným použitím rôznych agregácií a tie pochopiteľne pracujú len s hodnotami jedného typu.

Každá inštancia triedy *Value* je schopná vrátiť svoj popis, ktorým sme schopní za behu odlíšiť rôzne typy, trieda *ValueDescription*. Hodnoty rôznych typov musia vracať iné popisy, musia sa správať odlišne. Tento rozdiel dosiahneme použitím dynamického polymorfizmu – virtuálnych metód. Tie sú v jazyku C++ zviazané len s ukazovateľmi, či referenciami[13, 14]. Preto *ValueDescription* v skutočnosti tvorí len obal nad polymorfnými objektmi triedy *ValueDescriptor*. *ValueDescription* sa stará o ich správne kopírovanie, mazanie, deleguje na ne volané metódy. Týmto sme zaistili polymorfné správanie sa popisov hodnôt.

Zmienené popisy hodnôt majú zvyčajne malú, niekedy dokonca nulovú, veľkosť a navyše sú nemenné. Preto stačí vytvoriť len jednu inštanciu popisu pre všetky hodnoty akéhokoľvek typu. Časté vytváranie a mazanie malých objektov na halde

môže viesť k fragmentácii pamäti. Z tohoto dôvodu pri ich kopírovaní používame *reference counting*, ukladáme ich vo vnútri objektu *SmartPointer*[14]. Samozrejme, pre typy, ktorých použitie predpokladáme, ako sú napríklad (`std::string`, `int`, `double`, apod.), máme už na začiatku behu programu vytvorenú inštanciu. Jedná sa o statickú premennú triedy príslušného popisu. Toto riešenie nie je možné rozšíriť na všetky použité typy. Väčšinou sú dopredu neznáme a definície takýchto premenných nie je vhodné umiestňovať do hlavičkových súborov, [13], v ktorých sú šablóny popisov. Z optimalizačných dôvodov je výhodné naprogramovať vlastnú alokáciu malých objektov tak, ako je popísaná v [14]. Toto riešenie však nie je v tejto verzii programu implementované. Úložisko by bolo vhodné nielen pre popisy, ale aj pre mnohé iné objekty v tejto knižnici – napríklad malé hodnotové typy, ako sú ukazovateľ, `bool` alebo číselné hodnoty.

6.4 Implementácia uchovania hodnoty a kontroly typu

Bolo by veľmi nevýhodné, dokonca nemožné, implementovať rozhranie *Value* pre každý v teste použitý typ jazyka C++. Preto máme vytvorenú šablónu *ValueWrapper*, ktorá definuje celé rozhranie *Value* jednoduchým spôsobom pre akýkoľvek zvolený typ. Parciálnou alebo explicitnou špecializáciou tejto šablóny môžeme doplniť ďalšie metódy alebo pozmeniť už existujúce funkcie. Takto si uspôsobíme šablónu špeciálne pre účely implementovaného typu `Type`:

```
template <...>
class GenericValueWrapper :
        public Value,
        public ValueStorer<Type>
{
        ...
};

template <>
class ValueWrapper<Type> :
        public GenericValueWrapper<Type>
        // Predkovia, rozhranie Convertable.
{
        ...
};
```

Predok *ValueStorer* je využitý ako deklarácia, že táto implementácia rozhrania *Value* obsahuje hodnoty typu `Type`. Tento vzťah dedičnosti sa uplatňuje pri získavaní skutočnej hodnoty funkciou `value_cast`. Pretože `value_cast` nám umožňuje

priamy prístup k hodnote, nie je bežné, aby sme špecializovali priamo šablónu *ValueWrapper*. Väčšinou stačí pozmeniť pomocné šablóny, ktoré sa starajú o popis typu a konverziu na `std::string`. Použitie `value_cast` na získanie uloženej hodnoty:

```
Value * value = new ValueWrapper<Type>();
...
Type tmp = value_cast<Type>(value);
```

Šablóna *ValueWrapper* má niekoľko vlastností, ktoré je nutné zmieniť. Popisované vlastnosti umožňujú jej použitie pre takmer ľubovoľný typ, ale zároveň z nich plynú niektoré obmedzenia.

Ak chceme obaliť ľubovoľný používateľský objekt, nemôžeme predpokladať, že objekty tohoto typu sú porovnateľné alebo podporujú ďalšie operácie. V najvšeobecnejšom prípade, čo sa táto šablóna snaží zachytiť, nedefinujeme žiadne prídavné metódy. K hodnote sa vždy môžeme dostať použitím funkcie `value_cast`.

Jedinou požiadavkou šablóny *ValueWrapper* na typ poskytnutý ako jej parameter, je prítomnosť operátora určeného k výpisu. Konverziu hodnoty uloženej v šablóne *ValueWrapper* na typ `std::string` totiž vytvárame zavolaním operátora `ostream & operator <<(ostream &, const T &)`. Sme teda schopní konvertovať ľubovoľnú hodnotu, ktorú je možné vypísať do výstupného prúdu, v tomto prípade `std::stringstream`. Podobne ako v predchádzajúcom prípade, je možné upraviť konverziu pre zvolený typ. Zbavíme sa tým predpokladu, že hodnotu vieme vypísať do výstupného prúdu. Potrebnú zmenu vykonáme explicitnou špecializáciou šablóny *ValueWrapperConvertor*.

Pokiaľ chceme modifikovať správanie sa popisu hodnoty *ValueDescriptor*, explicitne špecializujeme šablónu *ValueWrapperDescriptor* pre zvolený typ. Pri vlastnej definícii špecializácie šablóny *ValueWrapperDescriptor* je výhodné, pokiaľ použijeme podobný spôsob jej ukladania, ako je použitý pre základné typy, napríklad `int`. Ako už bolo zmienené, jedná sa o prevenciu fragmentácie pamäti. Vytvoríme špecializáciu `ValueDescriptorCreator` pre typ `Type` v tomto tvare:

```
template <>
struct ValueDescriptorCreator<ValueWrapperDescriptor<Type> > :
        public StaticDescriptorCreator<
                ValueDescriptorCreator<ValueWrapperDescriptor<Type> >
        >
{
        friend struct StaticDescriptorCreator<
                ValueDescriptorCreator<ValueWrapperDescriptor<Type> >
        >;
private:
        typedef ValueWrapperDescriptor<Type> Descriptor;
        static ValueDescription::ValueDescriptorPointer instance;
};
```

Dôvod takejto špecializácie spočíva v tom, že šablóna *StaticValueCreator* vytvára za behu vždy len jedinú inštanciu popisu hodnoty. Tú uloží do definovanej statickej premennej `instance`. Aby *StaticValueCreator* mohol pristúpiť k tejto premennej, použijeme `friend` deklaráciu. Jedinou úlohou nami definovanej šablóny ostáva len správne vytvorenie premennej a tú ešte musíme definovať v implementačnom (`.cpp`) súbore danej triedy:

```
ValueDescription::ValueDescriptorPointer
ValueDescriptorCreator<ValueWrapperDescriptor<Type> >::instance(0);
```

Taktiež je možné vytvoriť si takéto správanie „ručne“. Táto konštrukcia má tú výhodu, že premenná `instance` je inicializovaná až pri jej prvom použití a šablóna *StaticValueCreator* správne zaobchádza s triedou *SmartPointer*.

Program *ExperimentsCreator* ako obal pre ľubovoľný typ, implementáciu triedy *Value*, zásadne používa inštancie šablóny *ValueWrapper*. Tieto inštancie parametrizuje jedine typom hodnoty, ostatné argumenty, konvertor na reťazec a popis hodnoty, sú predvolené typy. Negeneruje iné obalové typy, aj keď v knižnici *Experiments* je možné použiť aj iné implementácie rozhrania *Value* ako šablónu *ValueWrapper*. Z tohoto dôvodu pre zmenu správania sa hodnotových typov v generovaných testoch je nutné prispôsobovať len šablóny *ValueWrapper*, *ValueWrapperDescriptor* a *ValueWrapperConvertor*.

Zdrojové kódy, ktoré sú generované programom *ExperimentsCreator*, sa ďalej spoliehajú aj na to, že *ValueWrapper* definuje statickú metódu `getDescription`. Metóda `getDescription` vráti popis zvoleného typu bez toho, aby sme museli vytvárať inštanciu hodnoty.

6.5 Ukladanie výstupných veličín

Úlohou knižnice *Experiments* je aj pamätanie si výstupných hodnôt. Princíp, akým funguje ukladanie generovaných hodnôt si predstavíme tak, že za každý listový experiment umiestnime ďalší experiment (napríklad *Collector*). *Collector* si s každým jeho vykonaním zapamätá hodnoty všetkých argumentov, v ktorých je zavolaný. Ďalej budeme používať slovo *bod*[4] vo význame hodnôt premenných, ktoré chceme uložiť. K uloženiu dát máme vytvorené rozhranie *DataStorage*.

Zmienený prípad presne popisuje fungovanie experimentu *CollectorExperiment* a dátovej sady *DataSet*. Sada *DataSet* metódou `addValue` zachytáva všetky vektory hodnôt, v ktorých je požadované vykonanie príslušného experimentu *Collector*. Tieto hodnoty musia sedieť s popisom ukladaných hodnôt, ktorý bol zadaný v konštruktore objektu. Overenie vykonáme metódou `acceptsValue`. *DataSet* prijíma len ho-

[4] *Bod*, vektor hodnôt, je prvkom karteziánskeho súčinu domén ukladaných premenných.

mogénne dáta. Obsahuje tabuľku, ktorej riadky sú volania experimentu a stĺpce sú hodnoty jednotlivých sledovaných parametrov.

Všetky zapamätané hodnoty akejkoľvek dátovej sady musia byť vytvorené ako hlboká kópia hodnôt na vstupe. Správnu kópiu hodnoty získame zavolaním metódy `Value::clone`. Takto je zaručená existencia objektov až po zdeštruovanie skladu.

Aby sme neboli nútení ukladať iba jednoduchý zoznam hodnôt, máme predefinovanú dátovú sadu *TreeVariable*. Navyše nám umožňuje použiť agregácie na zvolené premenné. *TreeVariable* zobrazuje neagregované premenné na tie, ktoré sa vyskytujú v agregáciách. Štandardné agregácie, ktoré môžeme použiť sú suma, priemer, počet, maximum a minimum. Implementovaním čisto virtuálnych metód triedy `Aggregation` definujeme vlastné agregácie.

Spôsob, akým sada *TreeVariable* pracuje, definujeme už v konštrukture zadaním jeho argumentov. Rovnako ako v prípade triedy *DataSet* mu poskytneme meno sady a popis všetkých premenných, z ktorých sú vytvorené ukladané body. Naviac musíme doplniť zoznam agregovaných a neagregovaných premenných. Pri použití agregácií ešte dodáme inštanciu triedy `AggregationFactory`.

Z továrne metódou `createAggregation` získavame použité agregácie v závislosti na poradí premennej v zozname agregácií. Napríklad na premennú, ktorá je v zozname agregovaných premenných na indexe i chceme použiť agregáciu `Sum`. Potom `AggregationFactory` pre index i vráti vždy novú inštanciu agregácie `Sum`. Takto získané objekty, agregácie, budú správne zdeštruované počas mazania celej dátovej sady. Pri prípadnej implementácii továrne musíme dať pozor na to, že sa odkazujeme na indexy v zozname agregovaných premenných, nie na indexy premenných v popise bodu.

Pokiaľ nemáme definované žiadne agregácie, posledná premenná[5], je závislá na ostávajúcich. Takýto „neagregujúci sklad“ pracuje obdobne ako matematická funkcia. Zobrazenie nemôže mapovať rovnaké argumenty na rôzne hodnoty. V prípade, že chceme uložiť dva body, pričom ich nezávislé premenné nadobúdajú rovnaké hodnoty, sklad vyvolá výnimku. Napríklad neagregovane ukladáme premenné a, b a c. Premenná c je chápaná ako funkcia premenných a a b. V dátovej sade je už uložená trojica $(0, 0, 0)$. K nej chceme ešte pridať trojicu $(0, 0, 1)$. V tomto prípade by sme mapovali bod $(0, 0)$ na hodnoty 0 aj 1, čo však nie je možné. Sklad *TreeVariable* vyvolá výnimku `MappingStoredException` s textom „Mapping for the given point is already stored.“.

Dátovú sadu *TreeVariable* si jednoducho predstavíme ako strom[6]. Predpokladajme, že všetky ukladané premenné máme očíslované. Uzly na i-tej úrovni obsahujú

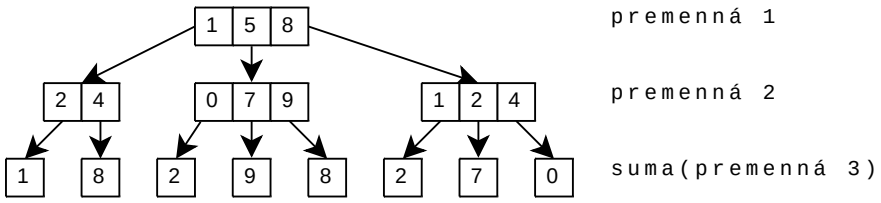
[5]Poradie je určené podľa parametru `group` konštruktura *TreeVariable*.

[6]Arita stromu, ktorým reprezentujeme *TreeVariable* nie je obmedzená. Strom vypadá podobne ako zložené indexy v databázach.

len hodnoty i-tej premennej. V koreni sú uložené všetky hodnoty, ktoré nadobúda prvá premenná. Ku každej hodnote v koreni máme pripojený ukazovateľ na podstrom. Každý takýto podstrom zostavíme len z tých bodov, ktoré v prvej premennej nadobúdajú odkazujúcu sa hodnotu. Z toho vyplýva, že v každom uzli i-tej úrovne nájdeme len niektoré hodnoty i-tej premennej. Pochádzajú z práve tých uložených bodov, ktoré v premenných $1, \ldots, i-1$ nadobúdajú hodnoty nájdené na ceste od koreňa do uzla.

TreeVariable pracuje tak, že pre každú ukladanú premennú vytvorí mapovanie – objekt, ktorý je implementáciou rozhrania `TreeVariable::Mapping`. Tieto mapovania interne využívajú kontajner `std::map`[7]. Takto získavame logaritmickú zložitosť na uloženie a vyhľadanie hodnoty v každom uzle. Celková zložitosť ukladania alebo vyhľadania je potom $O(p.log(n))$, kde p je počet premenných (hĺbka stromu) a n počet pamätaných bodov.

V konštruktore *TreeVariable* predávame zoznam premenných, ktoré chceme spracovať, prípadne aj zoznam agregovaných premenných. Mapovania premenných sa vytvárajú v tom poradí, v akom sa nachádzajú v zmieňovanom zozname. Uložené hodnoty každej premennej obsahujú odkaz na mapovanie nasledujúcej premennej. Spôsob ukladania môžeme vidieť na obrázku 6.1. Mapovanie prvej úrovne je vytvorené pre premennú pomenovanú „premenná 1“. Každá uložená hodnota prvej premennej obsahuje referenciu na mapovanie druhej premennej. Toto mapovanie obsahuje všetky hodnoty druhej premennej, ktoré sa v uložených bodoch vyskytli spolu s odkazujúcou sa hodnotou prvej premennej.



Obrázok 6.1: Mapovanie hodnôt uložených v *TreeVariable*.

Na obrázku 6.1 pre hodnotu prvej premennej 5, sa druhá premenná vyskytovala v hodnotách $0, 7, 9$. Súčet hodnôt tretej premennej je pre hodnoty $(5, 0)$ rovný 2. Táto situácia mohla vzniknúť napríklad uložením bodov $(5, 0, 1)$, $(5, 0, -1)$, $(5, 0, 2)$. Algoritmus vytvárania mapovaní a uzlov pri ukladaní bodu $(5, 0, 1)$ je nasledovný. Najprv sme do koreňa pridali hodnotu 5, k nej sme vytvorili mapovanie, uzol, druhej úrovne. Do neho sme vložili hodnotu 0. Následne sme pre túto hodnotu vytvorili agregáciu suma tretej premennej a tej sme priradili hodnotu 1. Pri pridávaní ďalšieho bodu, $(5, 0, -1)$, sme našli v koreni hodnotu 5. V mapovaní druhej úrovne, na ktoré sa

[7]Kontajner `std::map` garantuje logaritmickú zložitosť operácií.

odkazuje hodnota 5, nájdeme hodnotu 0. Ostáva len pozmeniť sumu tretej premennej, tá sa po pridaní druhého bodu zmení na nulu.

Za výpis zapamätanej dátovej sady zodpovedajú objekty typu *Output*. Fungujú s ľubovoľnou dátovou sadou. Máme na výber z dvoch možností: výstup do súboru oddelených hodnôt (CSV, SSV, vlastný oddeľovač) alebo XML výstup. Pred výpisom každej dátovej sady nastavíme metódou `setVariableDescriptions` popis výstupu. Ten pozostáva zo zoznamu zachytených premenných a mena sady, pričom tie budú pričlenené ako hlavička k vlastnej sade. S každým nastavením popisu začíname výstup novej sady, do jedného súboru môžeme uložiť viac zapamätaných sád. Výpis hodnôt vykonávame metódou `printValue`, ktorej poskytneme hodnotu a index premennej. Ten odpovedá indexu premennej vo vektore popisov, ktorý sme nastavili pred výpisom metódou `setVariableDescriptions`.

6.6 Experiment knižnice *Experiments*

Základom každého vytvoreného hierarchického experimentu sú jednotlivé kroky, z ktorých vytvoríme testovaný program. Práve k zachyteniu jediného kroku slúžia objekty odvodené od triedy *Experiment*.

Jednoznačne najdôležitejšou funkciou je vykonanie úlohy obsiahnutej experimentom. Túto operáciu vykonáme zavolaním metódy `run`, ktorej ako parameter zadáme vektor vstupných hodnôt – `ConstValuesVector`. Každý vytvorený experiment prispôsobuje telo metódy `run` svojím potrebám. Typicky vygeneruje potrebné hodnoty, zavolá jeho potomkov aj s novými vstupmi. Tento postup sa, pochopiteľne, môže opakovať v cykle, nekladieme žiadne obmedzenia.

Počas behu testu je výhodné, keď vidíme, ktorý experiment je práve vykonávaný. Zobrazovanie týchto informácií zapneme alebo vypneme metódou `setVerbose`. Vypisovanú správu upravíme preťažením metódy `printVerboseMessage`.

Po úspešnom vykonaní testu vygenerované dátové sady zozbierame metódou `getResults`. Táto metóda vracia dátové sady obsiahnuté v celom podstrome zakorenenom v experimente, na ktorom sme ju zavolali. Pretože priamočiarou implementáciou by dochádzalo k zbytočnému kopírovaniu dočasných objektov, je potrebné implementovať metódu `performGetResults`. Táto metóda dostáva ako parameter vektor, do ktorého pridávame obsiahnuté výstupné sady. Ak chceme na koniec vektora výsledkov pripojiť sady iného experimentu (potomka), použijeme statickú metódu `Experiment::AppendResults`. Experimenty, ktoré zbierajú dáta a ukladajú ich do súboru, môžu využiť volanie metódy `performGetResults` k tomu, aby vypísali uložené dátové sady.

Vo všeobecnosti *Experiment* nedefinuje žiadny spôsob zapájania do stromu. K tomu, aby sme vytvorili hierarchický experiment, máme niekoľko typov uzlov:

- *CompositeExperiment* Vytvára uzol, ktorý volá potomkov v želanom poradí zo zadanými vstupnými parametrami. Nevykonáva žiadne prídavné úlohy, len všetky volania deleguje na potomkov.

- *InnerExperiment* Používateľský experiment, obsahuje len jediného potomka. Úlohu, ktorú vykoná, musíme ručne implementovať. Neobsahuje žiadne dátové sady.

- *LeafExperiment* Používateľský experiment, neobsahuje potomka – je listom v hierarchii. Takýto experiment zrejme zbiera výsledky, preto v jeho konštruktore vyžaduje dátovú sadu určenú k ich zberu.

V triede *Experiment* nepredpisujeme žiadne metódy na vytvorenie grafu – hierarchie. Jediného potomka experimentu typu *InnerExperiment* nastavujeme metódou `setExperiment`. Aktuálneho potomka vracia funkcia `getExperiment`. Nakoľko *LeafExperiment* nemá podexperimenty, ani neudržiava odkaz na rodiča, neobsahuje žiadne metódy na jeho vkladanie do stromu. Zapájame ho len ako potomka iných experimentov. Naopak, *CompositeExperiment* obsahuje bohatšie rozhranie určené k nastaveniu jeho potomkov – k rozvetveniu ciest. Počas behu rešpektuje poradie, v akom boli deti zadané. Umožňuje aj prístup k jednotlivým potomkom podľa poradia, v ktorom budú vykonané.

Experimenty, ktoré sme zmienili, korektne vykonávajú a zľahčujú úlohy spojené s tým, že obsahujú potomkov alebo dátové sady. Samozrejme, nerobí problém, implementovať si vlastný typ experimentu alebo napríklad pridať ďalšiu funkcionalitu do už definovaných tried.

6.7 Definícia experimentu

Dôležitou triedou, ktorá nám umožní jednoduchším spôsobom vytvárať parametre – vstupné premenné pre experimenty, je *ExecPointCreator*. Ešte pred vygenerovaním nových hodnôt použitím metódy `setCallPoint` zadáme vektor, ktorý sme dostali na vstupe – parameter metódy `run`. Každú novú hodnotu pridáme zavolaním funkcie `addValue`. Takto pripravíme vektor hodnôt, ktoré budú vstupom pre potomkov. Vytvorený bod vracia metóda `getExecPoint`. Po zavolaní potomka budeme znova vytvárať nové hodnoty a typicky sa chceme vrátiť k vstupnému vektoru – metóda `resetExecPoint`. Pri tvorbe hodnoty môžeme definovať vlastnú hranicu, kam sa bude `resetExecPoint` vracať. Použitím `setResetPoint` postavíme „značku“ tak, aby sa zachovala posledná vložená hodnota.

Pokiaľ k definovaniu vlastného experimentu použijeme triedy *SimpleInnerExperiment* a *LeafExperiment*, čo je preferovaný spôsob, neimplementujeme priamo metódu

`run`. Namiesto toho preťažujeme funkciu `performExperiment`. Táto zmena je spôsobená tým, že obe triedy v `run` inicializujú objekt *ExecPointCreator*, ktorý neskôr oba experimenty používajú. Samozrejme, automaticky v nej dochádza k zavolaniu `performExperiment` s rovnakými parametrami, aké boli poskytnuté metóde `run`.

Trieda *InnerExperiment* má dve ďalšie špecializácie – *SimpleInnerExperiment* a *IterativeInnerExperiment*. Líšia sa v tom, akým spôsobom pracujú a volajú vnorený experiment. V prípade použitia triedy *SimpleInnerExperiment*, vnorený experiment vykonáme zavolaním niektorej z foriem funkcie `callNestedExperiment`. Ako parameter zadáme nové hodnoty vytvorené počas behu experimentu.

Ak sme sa rozhodli pre *IterativeInnerExperiment*, v algoritme 1 popíšeme postup akým sa volajú jednotlivé metódy. Ich implementáciou vytvoríme vlastný experiment.

Algoritmus 1 Metóda `run` experimentu *IterativeInnerExperiment*.

```
{Pripravíme experiment, inicializujeme ho.}
call prepare()

{Metódou finished riadime ukončenie behu experimentu.}
while not finished() do
  {Preťažením nasledovanej metódy vykonáme jeden krok.}
  call makeStep()

  {Metódou addValue počas behu pridávame vytvorené hodnoty. Po vykonaní
  každého kroku automaticky zavoláme potomka aj s novými hodnotami.}
  nestedExperiment.run(createdPoint)

  {Uvoľnenie prostriedkov, ktoré boli vytvorené v jednom kroku algoritmu.}
  call cleanStep()
end while

{Ukončenie behu experimentu, uvoľnenie všetkých prostriedkov.}
call finish()
```

Posledným dosiaľ neopísaným typom experimentu zostáva *LeafExperiment*. Úlohu, ktorú má experiment vykonať, definujeme preťažením čisto virtuálnej metódy `performExperiment`. Ako vieme, tento experiment nemá potomka, zvyčajne slúži na zber hodnôt. Dátovú sadu, do ktorej budú zozbierané údaje uložené, zadávame v konštrukture. Samotné uloženie jedinej vytvorenej hodnoty vykonáme zavolaním metódy `storeValue`. K zapamätaniu viacerých generovaných hodnôt využijeme funkciu `storeValues`. V oboch prípadoch bude pred zadané hodnoty vložený bod, v ktorom bol experiment zavolaný. Správanie týchto funkcií je podobné ako volanie

metódy `callNestedExperiment` v triede *SimpleInnerExperiment*.

6.8 Program *ExperimentsCreator*

Program *ExperimentsCreator* využívame k definícii samotného hierarchického experimentu. V nasledujúcich častiach opíšeme jeho fungovanie a najmä spôsob, akým je prepojený a spolupracuje s existujúcimi experimentmi v jazyku C++. Základnými súčasťami sú:

- *ExperimentsCreatorCore*: Obsahuje triedu *Experiment*. Vytvára prostredie pre prácu s celým hierarchickým experimentom (uloženie, načítanie, generovanie testu).
- *ExperimentsCreatorGUI*: Grafické rozhranie programu. K vytvoreniu používateľsky prívetivého prostredia využíva knižnice *DockPanelSuite*[16] a *Diagram*.
- *ExperimentsCreatorCLI*: Jednoduché konzolové rozhranie programu.
- *ExperimentsCreatorExperiments*: Základná sada experimentov. Využíva experimenty naprogramované v časti *BasicExperiments* knižnice *Experiments*.

6.9 Experiment programu *ExperimentsCreator*

Základom celej knižnice *ExperimentsCreatorCore* je definícia triedy *Experiment*. Objekty tohoto typu obsahujú metódy potrebné k načítaniu a uloženiu experimentu, ktorý reprezentujú, k deserializácii a serializácii do formátu XML. Musia byť schopné generovať časti kódu jazyka C++, ktoré ich správne inicializujú a zapoja do hierarchie. Dôležitým atribútom každého z experimentov je jeho parametrizácia. Experiment obsahuje zoznam jeho vlastností (trieda `ExperimentProperty`, vlastnosť `Properties`). Tie môžeme zmeniť a tým si prispôsobiť jeho správanie počas behu testu. Podstatnou časťou práce tejto triedy sú aj pomocné úlohy, ako je napríklad zapojenie experimentu do hierarchie – referencia na predka a zoznam prípadných potomkov, udržovanie podporných informácií – meno a odkazy na potrebné knižnice, či kontrola správnosti. Samozrejme, tieto úlohy experiment nezabezpečuje celkom sám, sú rozložené medzi viaceré triedy a podporné objekty.

K transformácii stromu experimentov využijeme rozhranie *IExperimentSaver*, ktoré sa nachádza v mennom priestore `ExperimentsCreator.Core.Savers`. Obsahuje metódu `SaveExperiment`, ktorou uložíme celý zadaný podstrom (hierarchický experiment) do zvoleného súboru. Výstupom tried implementujúcich zmienené rozhranie môže byť C++ kód, serializácia experimentu alebo projekt programu

SourceExperimentSaver	*SourceExperimentSaver* mení zadaný hierarchický experiment na zdrojový, testovací, kód. Vygeneruje súbor obsahujúci funkciu `main` s inicializáciu hierarchie a jednotlivých experimentov. Po úspešnom vykonaní testu sú všetky vygenerované dátové sady uložené do zadaných súborov.
VcProjExperimentSaver	Výpis, ktorý zo zadaného experimentu vytvorí projekt pre *Microsoft Visual Studio*. V konštrukture tohoto objektu zadávame súbor, ktorý obsahuje funkciu `main`.
VcProjUpdateExperimentSaver	Upravuje existujúci projekt tak, že v ňom vytvorí záložku „Test“ a do nej pridá testovací kód. V konštrukture tohoto objektu zadávame súbor, ktorý obsahuje funkciu `main` a projekt, do ktorého má byť testovaný kód pridaný.
XmlExperimentSaver	Serializuje hierarchický experiment do XML podoby, z ktorej sa dá znova načítať.

Tabuľka 6.1: Triedy implementujúce rozhranie *IExperimentSaver*.

Microsoft Visual Studio. Všetky triedy z tabuľky 6.9 predpokladajú, že parameter, ktorý popisuje ukladaný strom, je korektný[8]. Teraz ide hlavne o to, aby zadaný graf bol acyklický, čím sa vyhneme pretečeniu zásobníka. Pokiaľ je nejakým spôsobom serializovaný experiment nesprávny, nebude ho možné načítať. Postup, akým skontrolujeme správnosť celého hierarchického experimentu, bude vysvetlený v nasledujúcej časti.

Uloženie každého experimentu pozostáva najmä zo serializácie jeho vlastností do XML súboru. XML schému[17] dokumentu obsahujúceho serializovaný hierarchický experiment nájdeme v súbore `ExperimentStorage.xsd`. V niektorých prípadoch je potrebné, aby experiment mohol ovplyvniť, akým spôsobom má byť uložený. Tento problém vyriešime poskytnutím vlastnej implementácie rozhrania *IExperimentSaver*, menný priestor `ExperimentsCreator.Core`. Samozrejme, pokiaľ nám vyhovuje jednoduchý výpis vlastností, konverzia ich hodnoty na typ `string` a ich načítanie z reťazca, používame štandardný *GenericExperimentSaver*. Ten funguje automaticky, nemusíme sa explicitne zaujímať o proces ukladania experimentu.

Podobne ako rozhranie *IExperimentSaver* pri ukladaní, počas generovania testov,

[8]Korektnosť hierarchického experimentu bola definovaná v kapitole 3.10, strana 17.

funguje interface *IExperimentOutputter*. Jeho implementáciou ovplyvňujeme inicializáciu experimentu v testovacom kóde jazyka C++, prípadne vygenerujeme ďalšie zdrojové kódy. Nastavenie serializátora alebo objektu generujúceho výstup vykonáme najlepšie v konštruktore metódami `SetSaver` a `SetOutputter`.

Dôležitou súčasťou každého experimentu je informácia o jeho pozícii v hierarchii. K umiestneniu experimentu do grafu slúžia vlastnosti `Parent` a `Children`. Tieto vlastnosti nevyvolávajú výnimky v prípade, že vytvoríme cyklus alebo nesúvislý graf. Kontrola by bola zbytočne náročná na prostriedky užívateľa.

Načítanie experimentu z jeho serializovanej podoby v XML implementuje trieda *XmlExperimentLoader*. Vyžaduje, aby načítaný experiment bol validovaný proti schéme uloženej v súbore `ExperimentStorage.xsd`. Ak sa validácia alebo načítanie experimentu nepodarí, vyvolá výnimku `InvalidStorageException` s príslušnou chybovou správou. Aby sme mohli správne deserializovať jednotlivé experimenty, v ich XML podobe si pamätáme aj reťazec určujúci typ serializovaného experimentu. Počas inicializácie knižnice je vytvorený zoznam typov experimentov, ktoré knižnica rozpozná. Experimenty známych typov sú vytvárané objektmi implementujúcimi rozhranie *IExperimentCreator*. Ich zoznam je vytváraný v inicializácii knižnice a mapovanie z reťazca určujúceho typ experimentu udržuje továreň[18] *XmlExperimentFactory*.

Implementácia načítania experimentu (*IExperimentCreator*) musí presne odpovedať spôsobu, akým bol experiment serializovaný (*IExperimentSaver*). Pokiaľ používame štandardnú serializáciu *GenericExperimentSaver*, stačí doimplementovať abstraktnú triedu *GenericExperimentCreator* pre daný typ experimentu.

6.10 Hierarchický experiment, „pracovná plocha“

Prácu s celým hierarchickým experimentom umožňuje trieda *ExperimentWorkspace*. Pokiaľ sa ju rozhodneme použiť, ako to robí aj GUI, spájanie a rozpájanie experimentov, ako aj ich pridávanie, či mazanie, musíme vykonať pomocou metód pracovnej plochy.

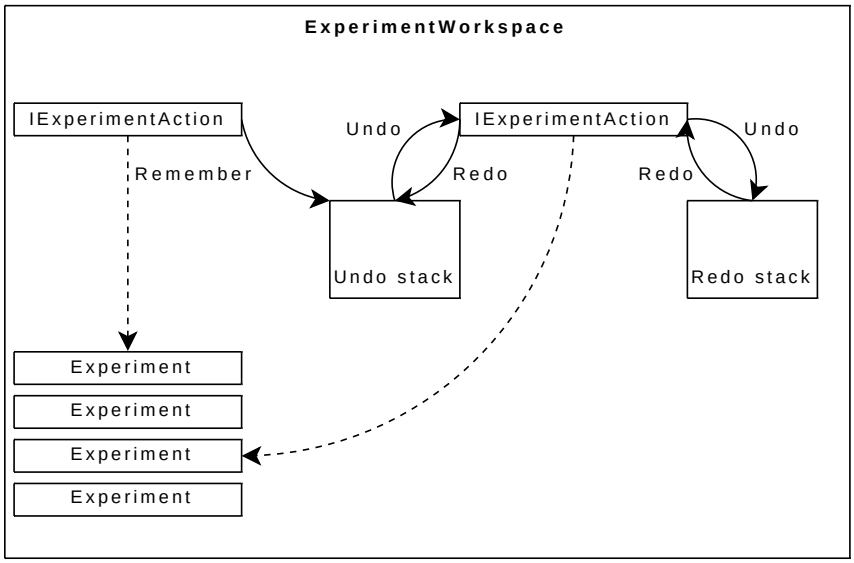
Po ich použití sa vytvára graf reprezentujúci celý experiment. Tento graf, trieda *TreeGraph*, obsahuje metódy na testovanie jeho súvislosti, prípadne vrátenie koreňa stromu – `TreeGraphAlgorithms.DetermineRoot`[9]. Z pracovnej plochy, pokiaľ je graf hierarchického experimentu strom, získame koreňový experiment metódou `DetermineRootExperiment`. Ak túto metódu použijeme a graf je nesúvislý alebo cyklický, bude vyvolaná výnimka, pretože nie je možné určiť koreň.

[9]Hierarchický experiment je orientovaný graf. V orientovanom strome vieme jednoznačne určiť koreň, je to vrchol, do ktorého nevedú žiadne hrany ($deg_{in}(root) = 0$).

Správnosť hierarchického experimentu uloženého v ploche kontroluje metóda `IsHierarchicExperiment`. V pracovnej ploche nájdeme aj metódy umožňujúce vyhľadanie duplicitných mien (`GetNonUniqueName`) alebo nesprávne nastavených experimentov (`GetInvalidExperiment`).

Hlavnou úlohou triedy *ExperimentWorkspace* je práca s celým vloženým hierarchickým experimentom. Preto v nej nájdeme funkcionalitu potrebnú k jeho ukladaniu alebo načítaniu. Samozrejme pred uložením alebo generovaním testovacieho kódu, je overená správnosť experimentu.

Pracovná plocha má vstavanú podporu pre operácie „Undo“ a „Redo“ v neobmedzenej hĺbke. Akákoľvek akcia, ktorú chceme na experimente vykonať, napríklad spojenie, či rozpojenie experimentov, premenovanie, pridanie alebo odobratie experimentu z plochy, implementuje rozhranie *IExperimentAction*. Tento interface obsahuje dve základné metódy, ktoré akciu buď vykonajú (`Perform`) alebo zrušia (`Undo`). Metódou `RememberAction` pracovnej plochy zvolenú akciu vykonáme a vložíme do zásobníka reverzibilných akcií. Plocha obsahuje aj ďalší podobný zásobník, ten je tvorený už zrušenými akciami.



Obrázok 6.2: Stavy akcie (IExperimentAction) vzhľadom k operáciám Undo a Redo.

Keď užívateľ vyvolá operáciu „Undo“, zavolá sa metóda `UndoLastAction` triedy *ExperimentWorkspace*. Akcia na vrchole zásobníka reverzibilných akcií je z neho vybraná. Ďalej sa zavolá jej metóda `Undo`, zrušíme ju. Následne sa vloží na zásobník zrušených akcií. Podobne metódou `RedoLastAction` vyvoláme operáciu „Redo“. Akcia na vrchole druhého zásobníka je z neho vybraná, vykoná sa metóda `Perform` a akcia sa uloží na zásobník reverzibilných akcií. Akcie v sebe neobsahujú kópie menených objektov. Inherentne obsahujú popis, ako akciu vykonať alebo zrušiť, ďalej

informácie o zmene (napríklad pôvodnú a novú hodnotu vlastnosti), a prípadne odkaz na experiment, ktorého sa akcia týka. Avšak tým, že si pamätáme referenciu na experiment, garbage collector ho nemôže uvoľniť.

6.11 Inicializácia knižnice

Knižnica *ExperimentsCreator.Core* obsahuje niekoľko tried, ktoré musia byť inicializované ešte pred jej použitím. Jedná sa najmä o zoznam známych experimentov (továreň *XmlExperimentFactory*), načítanie nastavení (*Settings*) a kontrolu prítomnosti XML schém.

Samotná inicializácia knižnice prebieha tak, že najskôr do singleton[18] triedy *ExperimentsInitializer* zadáme zoznam inicializátorov (`AddInitializer`) a inicializačných testov (`AddTester`), ktoré majú počas prebehnúť. Inicializáciu po pridaní všetkých inicializátorov a testov spustíme volaním metódy `Initialize` objektu *ExperimentsInitializer*. Trieda *ExperimentsInitializer* automaticky vykonáva niektoré základné inicializácie a testy.

Inicializátory aj testy sú implementáciou rozhrania *IInitializer*. Overenie funkčnosti programu, testuje sa napríklad prítomnosť schém, je spustené až po vykonaní všetkých inicializačných úloh. Zlyhanie inicializátora alebo testu je ohlásené vyvolaním výnimky, ktorá je následne zachytená. V prípade, že jedna z vyhodených výnimiek (*InitializationException*) má nastavený príznak `Fatal`, aplikácia skončí.

Zoznam experimentov sa určí načítaním používateľských experimentov. Tie sú implementované buď v jazyku C# alebo sú definované popisom uloženým v XML dokumente. Oba spôsoby nájdeme podrobne vysvetlené v kapitole 7. V používateľom nastavených adresároch sú počas inicializácie vyhľadané assembly[19] knižnice *.NET Framework* a zmieňované popisy.

Zo všetkých nájdených assemblies následne získame implementácie rozhrania *IExperimentExporter*. Z každej jeho implementácie je bezparametrickým konštruktorom vytvorená inštancia. Predpokladáme, že používateľ ho takýmto spôsobom implementuje. Rozhranie *IExperimentExporter* deklaruje vlastnosť `Creators`, ktorá obsahuje zoznam exportovaných experimentov. Pred jej použitím ešte zavoláme metódu `Initialize`, aby sa vykonali potrebné inicializácie.

Každý správne načítaný popis obsahuje jednu definíciu používateľského experimentu. Tento experiment bude ďalej rozpoznávaný a použiteľný počas ďalšieho behu programu.

6.12 Grafické rozhranie

Grafické rozhranie programu *ExperimentsCreator* z dôvodu užívateľského komfortu umožňuje zarovnávanie a presúvanie okien – knižnica *DockPanelSuite*[16]. Samotný hierarchický experiment editujeme v stavanom editore diagramov – knižnica *Diagram*.

Každé okno, ktoré sa zobrazuje vnútri „dock panelov“, je oddedené od triedy *DockContent*. Takýmto spôsobom sme dosiahli to, že okná sa dajú automaticky zarovnávať – o všetko sa postará knižnica *DockPanelSuite*. Navyše nám poskytuje možnosť zapamätať si rozloženie všetkých okien. Pri zatváraní aplikácie je aktuálne rozloženie panelov uložené a pri spustení opäť načítané. Súbor, v ktorom sa toto nastavenie nachádza, je v lokálnych nastaveniach používateľa programu. Všeobecne platí, že žiadne nastavenia programu *ExperimentsCreator* nie sú zdieľané medzi používateľmi programu.

S použitím diagramu, editácie pomocou GUI, vyplynulo niekoľko problémov. V knižnici *ExperimentsCreator.Core* nechceme žiadnym spôsobom podporovať niektorú z foriem editácie hierarchického experimentu. Preto každý experiment musíme obaliť tak, aby mohol byť pridaný do diagramu. Nakoľko pri vkladaní objektov do diagramu potrebujeme, aby spĺňali rozhranie *IVertex*, vytvorili sme obal experimentu – *ExperimentVertex*.

Obal *ExperimentVertex* zohráva ešte jednu veľmi dôležitú úlohu. Vlastnosti experimentu editujeme v komponente *PropertyGrid*[20]. Tento komponent ako zoznam editovateľných vlastností štandardne používa len tie, ktoré sú vlastnosťami objektu v jazyku C#. Pretože na každom experimente chceme editovať len vlastnosti z kolekcie `Properties`, implementovali sme rozhranie *ICustomTypeDescriptor*. Jeho naprogramovaním dosiahneme zobrazovanie nami definovaných vlastností v komponente *PropertyGrid*.

Ďalšou požadovanou schopnosťou grafického rozhrania je možnosť zarovnať experimenty v diagrame. K tomuto účelu sme si prispôsobili triedu *ExperimentsWorkspace* tak, aby obsahovala mapovanie ukladaných experimentov na ich reprezentácie v diagrame. Toto mapovanie nám umožní jednoducho prejsť strom experimentov a zarovnať reprezentácie zvoleným spôsobom. Objekty, ktoré vykonávajú zarovnanie, predpokladajú na svojom vstupe graf hierarchického experimentu, ktorý je strom, v opačnom prípade zlyhajú. Túto požiadavku, ešte pred zarovnaním, skontroluje pracovná plocha a v prípade neúspechu je používateľovi oznámená chyba.

V prostredí programu, samozrejme, chceme ovládať operácie „Undo“ a „Redo“. S každou zmenou vlastnosti experimentu, s každým pridaním či odobraním experimentu, vygenerujeme, vykonáme a do pracovnej plochy uložíme reverzibilnú akciu, ktorá vykonáva požadovanú úlohu. Problém nastáva so synchronizáciou pracovnej plochy

a diagramu. Napríklad metódou `Remove` triedy *GUIExperimentWorkspace* musíme vymazať experiment z plochy a jeho reprezentáciu vybrať z diagramu, poprípade ešte rozpojiť jeho spojenia. Pokiaľ objekt vymažeme priamo v diagrame, reprezentácia experimentu je odobraná automaticky. Ďalej je nutné odobrať experiment z plochy. Tu by sme sa dostali do akéhosi cyklu – mazanie z plochy maže z diagramu.

Tento cyklus riešime na úrovni akcií, ktoré vykonávajú jednotlivé úlohy. Obsahujú aj metódy, ktoré vykonajú len čiastkové úpravy plochy a diagramu, zosynchronizujú ich. Po zladení týchto objektov vytvorené akcie vložíme na „undo zásobník“. Počas vykonania „Undo“ alebo „Redo“ operácie na zásobník nevkladáme žiadne nové akcie. Prevedenie operácie „Undo“ alebo „Redo“ môže generovať novú akciu, typicky opačnú k práve vykonávanej.

6.13 Spracovanie výstupu

Jednou z možností, ktorú nám grafické prostredie ponúka, je spracovanie uložených výstupných sád, či už v ich XML alebo CSV podobe.

Spracovanie dát funguje jednoduchým spôsobom. Najprv zadefinujeme sledované premenné, t.j. tie premenné, ktoré sa majú po spracovaní objaviť vo výsledkoch. Potom vykonáme samotné spracovanie dátovej sady. Počas spracovania enumerujeme uložené body, z nich vyberáme hodnoty sledovanych premenných a nakoniec zo zachytených údajov vytvoríme požadovaný výstup.

Rozhranie *IDataProcessor* definuje základné metódy pre objekty spracúvajúce dátové sady – pridanie sledovanej premennej `AddDataSet`. Spracovanie výstupu spustíme metódou `Process`. Pred samotným spustením je ešte nutné zadať vstup, z ktorého budeme dáta vyberať, vlastnosť `DataProvider`.

Interface *IDataProvider* slúži na získanie popisu súboru obsahujúceho dátové sady a výpis v ňom obsiahnutých hodnôt. Zoznam uložených dátových sád získame metódou `EnumerateDataSets`. Zoznam premenných v sade zase zavolaním funkcie `EnumerateDataSetVariables`. Hodnoty, ktoré sú uložené v požadovanej sade, získame vykonaním metódy `ReadDataSet`.

Implementovaním týchto rozhraní získame prístup k iným typom dátových skladov alebo nové možnosti ich spracovania. Zatiaľ sú podporované len XML (*XmlDataProvider*) a CSV (*TextDataProvider*) súbory. Čo sa týka ich spracovania, výstupom môže byť tabuľka v HTML (*HtmlTable*) alebo v Latex-e (*LatexTable*), prípadne dvojrozmerný graf (*Graph2*).

Kapitola 7

Vývoj experimentu

V tejto kapitole je vysvetlený postup, ktorým vytvoríme nový experiment a integrujeme ho do prostredia programu *ExperimentsCreator*. K splneniu tohoto cieľa existujú dva postupy.

Prvý z nich, jednoduchší, vyžaduje naprogramovanie experimentu len v jazyku C++. „Do diagramu“ ho dostaneme pomocou formuláru, v ňom vytvoríme popis experimentu. Ten následne uložíme napríklad do adresára, z ktorého sa automaticky načítajú experimenty pri spustení programu. Ďalej už s týmto experimentom pracujeme štandardným spôsobom.

Definovanie experimentu pomocou formulára nepostihuje všetko, čo experiment knižnice *ExperimentsCreator.Core* môže využiť. Druhý postup vyžaduje naprogramovanie vlastného opisu experimentu v jazyku C#. Vytvorenú assembly platformy .NET Framework integrujeme do prostredia *ExperimentsCreator* implementovaním rozhrania *IExperimentExporter* a jej prekopírovaním do vhodného umiestnenia.

7.1 Experiment knižnice *Experiments*

V kapitole 6 sme ukázali triedu *Experiment* a jej rozšírenia, ktoré zjednodušujú vývoj nových experimentov. V tejto časti presne vysvetlíme princípy a poukážeme na problémy, ktoré sa počas tvorby nových typov experimentov vyskytnú a načrtneme ich riešenie.

Budeme vytvárať experiment prispôsobený tak, aby spĺňal všetky pravidlá pre použitie v nasledujúcej sekcii. Chceme vytvoriť jeho popis formulárom na definíciu nového experimentu, obrázok 7.1, a potom ho ihneď využiť v programe *ExperimentsCreator*.

Začneme definíciou triedy, ktorá je oddedená napríklad od *SimpleInnerExperiment*[1]. K vytvoreniu jej základných metód môžeme použiť pridaný program *Code-*

[1]Samozrejme, môžeme použiť iné triedy, napríklad *InnerExperiment*, *LeafExperiment*, ...

Generator. Ten generuje akúsi šablónu, ktorú musíme implementovať – „vyplniť“.

Z dôvodu použitia „asistovaného integrovania“ do programu *ExperimentsCreator*, vytvoríme konštruktor prijímajúci jediný parameter typu `std::string`, a to meno experimentu. Trieda *CustomExperiment* knižnice *ExperimentsCreator.Core*, ktorá popisuje takto vkladaný experiment, vytvára nové inštancie zásadne použitím tohto konštruktora.

K vykonaniu vlastného tela musíme preťažiť čisto virtuálnu metódu príslušnej špecializácie (`run`, `performExperiment`). Taktiež zavolanie potomka alebo ukladanie výsledkov závisí na použitom základe[2].

K parametrizovaniu experimentov slúžia ich vlastnosti. Pokiaľ chceme dodržať štandard vhodný pre integrovanie experimentu jednoduchým spôsobom, implementujeme ich tak, že pre každú vlastnosť vytvoríme metódy `get` a `set`. Napríklad časť experimentu pre vlastnosť `Start` by mohla vyzerať podobne:

```
/**
 * Starting point of the loop.
 *
 * @param start Starting point.
 */
virtual void setStart(ReferableProperty<int> start) {
        this->start = start;
}

/**
 * Performs the experiment at the given point @c values.
 *
 * @param values Parameters of the called experiment.
 */
virtual void performExperiment(const ConstValuesVector & values) {
        this->start.setFromVariables(this->calledValues);
        ...
}
```

Vytvorili sme vlastnosť `Start` typu `int`. Táto vlastnosť priamo nepoužíva typ `int`, ale inštanciu šablóny `ReferableProperty<int>`. Vďaka tomu je táto vlastnosť schopná odkázať sa na hodnotu vstupnej premennej. Prípadný odkaz užívateľ vyplní počas jej editácie v GUI. Pri každom priebehu experimentu je nutné inicializovať jej hodnotu zo vstupu, čo vykonáme zavolaním metódy `setFromVariables` šablóny *ReferableProperty*.

Podpora pre zapojenie experimentu do hierarchie ostáva na nás. Trieda *CustomExperiment* podporuje zapojenie jediného potomka a to metódou `setExperiment`, mali by sme ju poskytnúť.

[2]Presné metódy a ich použitie nájdeme v kapitole 6

Konštruktor	Musí existovať konštruktor prijímajúci jediný parameter typu `std::string` – meno experimentu.
Vlastnosti	K nastaveniu hodnoty konkrétnej vlastnosti vytvoríme metódu `set<meno vlastnosti>(const <typ> &)`. Vlastnosť odkazujúca sa na vstupnú premennú má ako svoj typ šablónu *ReferableProperty* parametrizovanú skutočným typom.
Potomok	K zapojeniu potomka používame metódu `setExperiment`.

Tabuľka 7.1: Požiadavky triedy *CustomExperiment* na experiment.

7.2 Integrácia experimentu

Popis experimentu, predpokladáme, že už je naprogramovaný v jazyku C++, vytvoríme najjednoduchšie v dialógu programu *ExperimentsCreator*. Formulár vyvoláme vybratím položky *New experiment* v menu *File*. Nájdeme ho na obrázku 7.1. Popis jednotlivých vlastností sa nachádza v tabuľke 7.2.

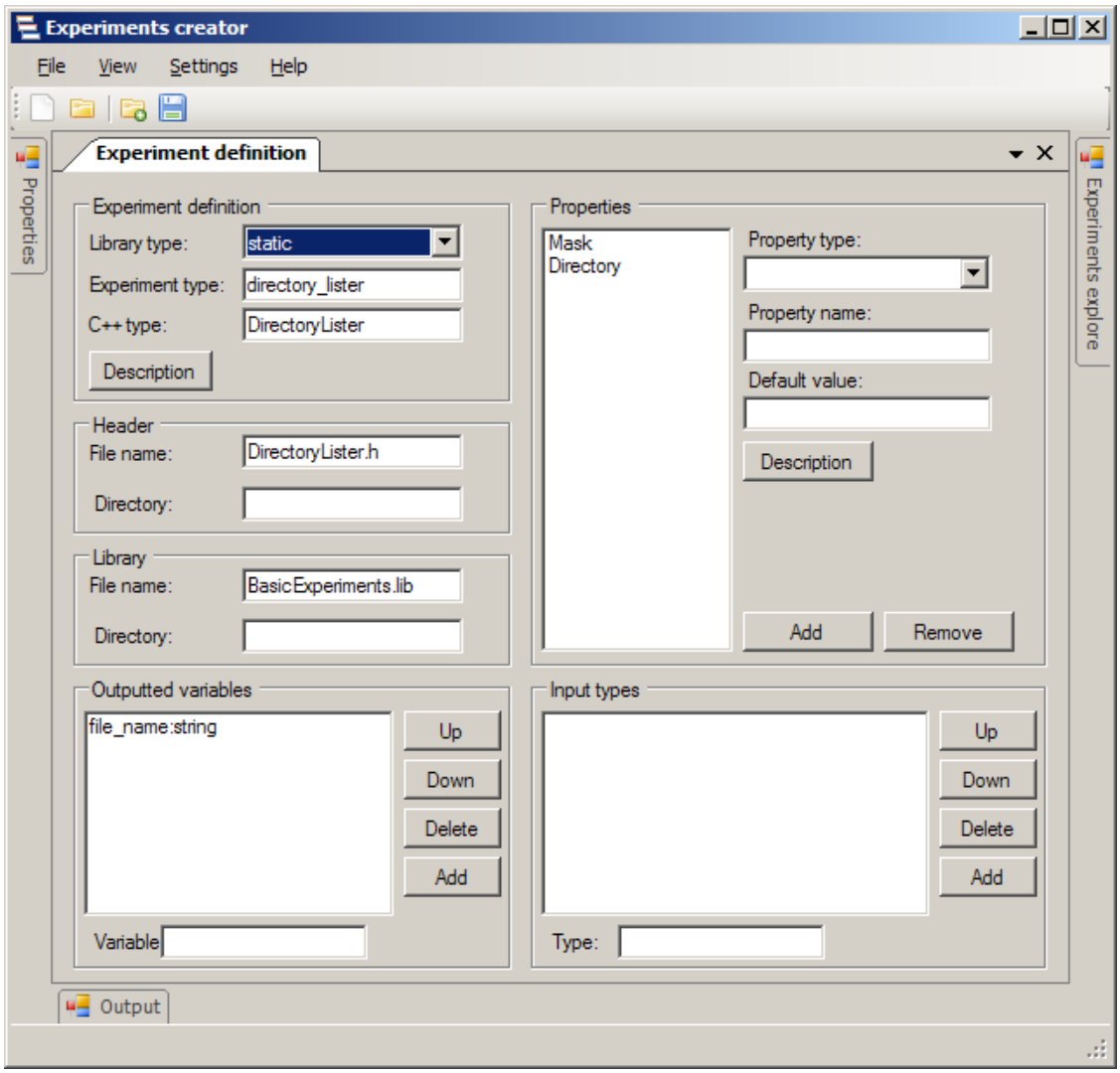
Pri zadávaní požiadaviek na vstupné premenné, vlastnosť *Input types*, postupujeme obdobne, ako pri odkazovaní sa na vstupy pomocou ich poradia. Určujeme, aké typy experiment potrebuje na vstupe. Pokiaľ chceme zadať prázdnu požiadavku, vyplníme hodnotu `<wild_card>`. Toto zadanie sa hodí v prípade, že sa experiment odkazuje napríklad na prvú a tretiu premennú vstupu. Druhú nemôžeme pri zadávaní vynechať a tak vložíme prázdnu požiadavku. Ak experiment nevyžaduje žiadny vstup, špecifikovanie tejto vlastnosti vynecháme. Je nutné podotknúť, že experiment môže na vstupe dostať viac parametrov, ako sme zadali. Tie však musia spĺňať zadané požiadavky.

Najdôležitejšou súčasťou experimentu je jeho výstup. Ten zadávame v skupine *Outputted variables*. Najprv pomenujeme výstupnú premennú a určíme jej typ, čo do poľa *Variable* zadávame v tvare:

`<meno výstupnej premennej>:<typ premennej>.`

Napokon ju pridáme do zoznamu výstupov tlačítkom *Add*.

Experimenty chceme chceme často parametrizovať. K tomu sú určené ich vlastnosti. V tomto prípade ich zadávame vyplnením skupiny *Properties*. Zvolíme typ vlastnosti (*PropertyType*), jej meno (*Property name*) a počiatočnú hodnotu (*Default value*). Ku každej premennej je vhodné vyplniť jej popis. Ten sa bude zobrazovať počas jej editácie v GUI. Pri vlastnostiach máme len predvolenú množinu typov, ktoré môžeme použiť. Jej rozšírenie o vlastné typy bude vysvetlené v nasledujúcej sekcii. Vo vlastnostiach nájdeme aj referenčné typy odvodené od základných. Tie umožňujú vytvárať vlastnosti schopné odkazovať sa na hodnoty vstupných premenných. Meno zadanej vlastnosti musí presne odpovedať (aj na veľkosť písmen)



Obrázok 7.1: Formulár určený k definovaniu používateľského experimentu.

set metóde vlastnosti. Pokiaľ chceme zobraziť v editore iné meno, vyplníme popis vlastnosti.

Vytvorený popis experimentu môžeme načítať a ukladať tlačítkami, ktoré sa počas otvorenia dialógu pridali do panelu nástrojov. Pri úpravách je potrebné dať pozor, aby sme nevytvorili dva experimenty s tým istým typom. Ten musí byť jednoznačný, pretože podľa neho program *ExperimentsCreator* rozlišuje serializované experimenty. Pokiaľ dôjde k tejto chybe pri načítaní konfliktného, experimentu sme upozornení chybou *„Creator with the type '<typ>' is already stored.“*. Chyba nespôsobí ukončenie programu, avšak nie je definované, ktorý z experimentov sa bude pod týmto typom používať. Nevieme predpovedať, ktorý z experimentov bude načítaný ako prvý.

Takto vytvorený experiment je v knižnici *ExperimentsCreator.Core* implemento-

Library type	Typ knižnice, ktorá bude použitá počas kompilovania testov používajúcich definovaný experiment.
Experiment type	Typ experimentu – reťazec, pod ktorým bude experiment evidovaný v prostredí *ExperimentsCreator*.
C++ type	Meno triedy v jazyku C++, ktorá obsahuje implementáciu experimentu.
Header	Hlavičkový súbor experimentu.
Library	Súbor s knižnicou – buď je to statická knižnica alebo `.cpp` súbor s implementáciou.
Outputted variables	Premenné, ktoré experiment vytvára.
Input types	Vstupné premenné – požiadavky na vstup.
Properties	Zoznam vlastností, ktorými môžeme experiment parametrizovať.

Tabuľka 7.2: Nastavenia používateľského experimentu.

vaný triedou *CustomExperiment*. Neumožňuje generovať vlastný kód. Odkazovanie na vstupné premenné je možné definíciou požiadaviek na vstupné parametre alebo použitím odkazujúcich sa vlastností. Jedná sa o vloženie jednoduchého, ale stále veľmi dobre parametrizovateľného experimentu, ktorý nevyžaduje zmeny jeho serializácie alebo generovanie vlastného C++ kódu. Základná inicializácia experimentu pozostáva z nastavení jeho vlastnosti. Väčšina experimentov nie je navrhovaná, aby generovala kód alebo požadovala špeciálne zmeny serializácie. Tento spôsob ušetrí veľa času, ktorý by sme strávili riešením podobných úloh pri tvorení nových experimentov. Vyhovuje akémusi štandardnému chápaniu experimentu, ktorý spracuje vstupy a vytvorí výstupy podľa vyplnených hodnôt jednotlivých vlastností.

7.3 Experiment programu *ExperimentsCreator*

Pokiaľ nám nevyhovujú obmedzenia spôsobené použitím štandardnej, „asistovanej“ integrácie do programu *ExperimentsCreator*, musíme vytvoriť vlastný popis experimentu. Tento postup používajú najmä experimenty, ktoré požadujú neštandardnú inicializáciu, inú serializáciu vlastností (*DataCollector*) alebo generujú vlastné experimenty – vlastný testovací kód (*CodeBlock*, *MethodCaller*, *ObjectCreator*). Všetky z nich sú implementované v projekte *ExperimentsCreator.Experiments* programu *ExperimentsCreator*.

Počas tvorby nového experimentu je výhodné túto knižnicu využiť. Obsahuje totiž triedy, ktoré generujú kód uložený v šablónach (*GeneratedCodeOutputter*). Ďalej umožňuje do vygenerovaného kódu vložiť potrebné knižnice, nastavené v jednej

z vlastností experimentu (*GeneratedLibrary*) alebo vkladanie odkazov na premenné (*ExpressionProperty*).

Pretože implementácia môže byť pomerne komplikovaná, vyžaduje znalosť aj menej využívaných častí jazyka C#[3]. Pokúsime sa objasniť len tvorbu základov. Zložitejšie konštrukcie a použitie častí knižnice *ExperimentsCreator.Experiments* nájdeme v tutoriáloch a príkladoch na priloženom médiu.

Atribúty experimentu nastavíme vlastnosťou `Properties` triedy *Experiment*. Musíme ju implementovať podľa vlastných potrieb. Každá vlastnosť experimentu je potomkom triedy *ExperimentProperty*. Je vhodné atribútmi[4] *DisplayName*, *Category* a *Description* nastaviť jej popis, ktorý sa zobrazuje v GUI počas jej editovania.

K uľahčeniu ich implementovania máme preddefinované triedy *SimpleProperty* a *WrappedProperty*, ktoré samé spravujú hodnotu alebo obaľujú premennú triedy tak, aby bola vlastnosťou experimentu. Kolekciu *ExperimentPropertyCollection* môžeme použiť k exportovaniu ich zoznamu. Pokiaľ náš experiment nemá žiadne vlastnosti, ako ich zoznam použijeme triedu `EmptyPropertyCollection.Collection`.

Pri ich definovaní však musíme dať pozor, aby sme vlastnosti po vytvorení experimentu už nemenili. Obálka experimentu v GUI ich totiž počas jej konštrukcie načíta len raz a ďalej sa odkazuje len na nájdené vlastnosti.

Overovanie správnosti experimentu rieši metóda `IsValid`. Jej preťažením ovplyvníme požiadavky na správne nastavený experiment. Vlastné objekty, implementácie rozhrania *IExperimentValidator*, ktoré sa postarajú o zistenie správnosti niektorej časti celku, nastavujeme zoznamom `Validators`. Do neho implicitne počítame aj všetky vlastnosti experimentu – k splneniu podmienok správnosti musia byť správne nastavené.

Ukladanie experimentu riešime implementovaním rozhrania *IExperimentSaver*. Počas generovania výstupu, rozhranie *IExperimentOutputter* môžeme zadefinovať spôsob, akým experiment v testovacom kóde (jazyk C++), vytvoríme, zapojíme do hierarchie a inicializujeme.

Načítanie experimentu realizuje trieda implementujúca rozhranie *IExperimentCreator*. Jej úlohou je deserializovať uložený experiment a vytvoriť jeho skutočnú reprezentáciu (metóda `CreateExperiment`). Registrovaním týchto objektov vzniká zoznam použiteľných experimentov. Každý z nich vracia typ experimentu, ktorý je schopný vytvoriť (vlastnosť `ExperimentType`).

Je zrejmé, že s každým novým typom experimentu musíme vytvoriť objekt, ktorý je schopný vracať jeho inštancie. Práve to je úlohou tohto rozhrania. Vie vytvoriť nový experiment žiadaného typu, napríklad pri vložení do diagramu, alebo deserializovať načítaný experiment. K poslednej úlohe stačí využiť, napríklad de-

[3]Jedná sa najmä o atribúty[21], komponent *PropertyGrid*[20] a s ním *UITypeEditor*[22].

[4]V tomto prípade ide o atribúty[21] jazyka C#.

dičnosťou, triedu *GenericExperimentCreator*. Pokiaľ serializujeme vlastnosti experimentu špeciálnym spôsobom, preťažujeme jej metódu `SetProperty`.

Registrácia nových typov experimentov prebieha automaticky použitím rozhrania *IExperimentExporter*. Jeho implementácie musia mať bezparametrický konštruktor. Vlastnosťou `Creators` počas inicializácie programu *ExperimentsCreator* vrátime implementované experimenty. Metódou `Initialize` môžeme dokončiť inicializáciu prostredia potrebnú k ich behu. Túto metódu je možné využiť aj k definovaniu nových typov vlastností pre *CustomExperiment* alebo zadaniu nových typov spracovania výstupu[5].

Pokiaľ sa rozhodneme behovú knižnicu rozšíriť, vytvárame nové projekty do adresára `Additional` v inštalačnom adresári. Má to tú výhodu, že ku adresárom hlavičiek môžeme pridať adresár `Additional`. Ďalej ich vypĺňame iba relatívne voči tomuto adresáru, či už vo vlastnostiach experimentov alebo počas jeho integrovania do prostredia. Je dobré nepoužívať v odkazoch na knižnice absolútne cesty (platí aj pre hlavičky). Vytvorené statické knižnice je lepšie prekopírovať do adresárov `DebugLibs` a `ReleaseLibs`. Následne sa odkážeme len na meno knižnice.

Tento postup si môžeme všimnúť na experimentoch *CounterExperiment* a agregácii *Concat*, nájdeme ich v zmienenom adresári `Additional`. Ich použitie sú hierarchické experimenty `counter_experiment_test.xml` a `concat.xml`.

[5]Nájdeme v príkladoch.

Kapitola 8

Záver

V práci sme popísali princíp testu založeného na hierarchickom experimente, jeho obmedzenia a prednosti. Na základe tohoto princípu, sme vytvorili program určený k testovaniu softvéru. Podobne, ako implementovaná aplikácia vytvára test, funguje napríklad spracovanie obrazu. Výstup jednej „krabičky“ je vstupom druhej[1], pričom jednotlivé kroky sú zapojené do hierarchie. Takýmto zapojením a jeho konfiguráciou ovplyvníme test a zbierané dáta.

Výsledkom však nie je zďaleka „dokonalá“ aplikácia, ktorá vie automaticky generovať testy, či vstup. Taktiež ani automaticky neskúma výkonnostné parametre, ako to robia napríklad profilery[23]. Na druhú stranu nevymedzujeme konkrétny účel testovania. Vo vytvorenej aplikácii môžeme testovať tak výkon ako aj korektnosť, napríklad uplatnením unit-testingu[24]. V práci sme sa zamerali na spôsob jej rozšírenia o nové typy experimentov.

Hlavné nevýhody a nedostatky sú:

- Z optimalizačných dôvodov by bolo výhodné doimplementovať úložisko malých objektov[14] spomínané v kapitole 6. Jeho implementácia by nebola obzvlášť náročná, avšak vyžadovala by zrejme dosť času na odladenie.

- Možnosť lepšej podpory zadania knižníc počas editácie, prípadne pohodlnejšie použitie menných priestorov v programe. Často sa treba odkazovať plne kvalifikovanými názvami – obzvlášť pri použití experimentu *ObjectCreator*.

- Odstránenie požiadavky korektnosti experimentu počas jeho ukladania. Nejedná sa o náročnú úlohu. Vytvoríme odlišný formát dát – zoznam experimentov a ich zapojení. Takto môžeme uložiť aj cykly, pričom experimenty môžeme serializovať rovnakým spôsobom ako doteraz. V tomto prípade nevyužijeme hierarchiu XML, do ktorej sa premieta definovaný strom, čo bolo prvotným zámerom.

[1] Podobne pracuje napríklad knižnica *JaGrLib*[25]

- Lokalizácia programu, taktiež typová kontrola v programe *ExperimentsCreator* pri odkazoch na premenné nie je „dokonalá“.

- Niekedy chýba príjemnejšie spracovanie diagramu, ako je označenie viacerých objektov. Automatická práca s celým podstromom, pri posunutí experimentu sa posunie celý jeho podstrom, podobné správanie je výhodné aj pri iných akciách.

- Podpora pre viac programovacích jazykov. Do generovania výstupu pridáme ešte jeden parameter – jazyk. V každom z podporovaných programovacích jazykov by sme museli vytvoriť behovú knižnicu podobnú *ExperimentsLibrary*.

- Podpora programu pri definovaní vstupov, budeme asistovať rôznymi sprievodcovami apod. Tu program môže poradiť napríklad generovanie náhodných dát, vyberanie hodnôt z extrémov, preddefinované sady a ich vzájomné kombinácie. Je však dôležité dať pozor na množstvo vstupných dát, musíme vybrať len relevantné. Pretože je nutné skúšať všetky variácie vstupov, celková veľkosť rastie ako karteziánsky súčin vstupov pre jednotlivé parametre.

- Podpora spustenia vygenerovaného kódu priamo v aplikácii *ExperimentsCreator*. Väčšia automatizácia.

- Chýba štatistická analýza dát, môžeme spolupracovať s projektom *R*. Výhodné môže byť aj naimplementovať viac formátov výstupu. Lepšia podpora pre testovanie korektnosti a unit testing.

Výhodou aplikácie je možnosť vizuálne editovať navrhnutý test a sledovať používateľom zvolené parametre. Z nich následne vytvoríme ich prezentáciu, štandardne graf alebo tabuľku. Editácia prebieha vizuálne, z toho plynú niektoré základné výhody, ako je podpora vracania zmien, ukladanie a načítanie testov. Nie sme nútení starať sa o rutinné programovanie, a plne sa sústredíme na vytváraný test. Problémom môže byť obmedzená sada experimentov, čo sa dá prekonať buď použítím experimentu *CodeBlock* alebo definovaním používateľského experimentu.

Aplikácia sa dá zlepšovať ešte inak ako navrhnutými zmenami. Môžeme sa vybrať aj tým smerom, že zadefinujeme zložitú hierarchiu komplikovaných testov systému. Následne budeme ovládať ich beh, kontrolovať a distribuovať ich po sieti.

S určitosťou však môžeme povedať, že program je možné z mnohých stránok jednoducho rozšíriť napríklad o nové experimenty, spracovanie dát alebo zlepšenia behovej knižnice. Snáď splnil základný účel. Ním je vytvorenie hierarchického experimentu na pohodlnejšie vytvorenie takéhoto testu, ako by sme ho mali programovať „ručne“.

Literatúra

[1] Pezzè M., Young M.: *Software Testing and Analysis-Process, Principles, and Techniques*, John Wiley & Sons, 2008.

[2] Nešetřil J., Matoušek J.: *Kapitoly z diskrétní matematiky*, Karolinum, 2003.

[3] Microsoft: *.NET Framework*, http://msdn.microsoft.com/en-us/netframework, júl 2008

[4] Kitware, Inc: *Cmake Cross Platform Make*, http://www.cmake.org, júl 2008

[5] Microsoft: *Project Files*, http://msdn.microsoft.com/en-us/library/2208a1f2(VS.80).aspx, júl 2008

[6] Wikipedia, The Free Encyclopedia: *Comma-separated values*, http://en.wikipedia.org/wiki/Comma-separated_values

[7] World Wide Web Consortium: *Extensible Markup Language (XML)*, http://www.w3.org/XML, júl 2008

[8] Eckel B.: *Thinking in C++, Second Edition*, Prentice Hall, 2000

[9] World Wide Web Consortium: *W3C HTML*, hhttp://www.w3.org/html, júl 2008

[10] LaTeX project team: *LaTeX – A document preparation system*, http://www.latex-project.org, júl 2008

[11] R Development Core Team: *The R Project for Statistical Computing*, http://www.r-project.org, júl 2008

[12] Wikipedia, The Free Encyclopedia: *Portable Network Graphics*, http://en.wikipedia.org/wiki/Portable_Network_Graphics, júl 2008

[13] Sutter H., Alexandrescu A.: *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley, 2005

[14] Alexandrescu A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001

[15] Wikipedia, The Free Encyclopedia: *Run-time type information*, http://en.wikipedia.org/wiki/RTTI, júl 2008

[16] World Wide Web Consortium: *W3C XML Schema*, http://www.w3.org/XML/Schema, júl 2008

[17] Luo W.: *SourceForge.net: DockPanel Suite*, http://sourceforge.net/projects/dockpanelsuite, júl 2008

[18] Gamma E., Helm R., Johnson R., Vlissides J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998

[19] Microsoft: *Assemblies Overview*, http://msdn.microsoft.com/en-us/library/k3677y81.aspx, júl 2008

[20] Microsoft: *Getting the Most Out of the PropertyGrid Control*, http://msdn.microsoft.com/en-us/library/aa302326.aspx, júl 2008

[21] Microsoft: *Extending Metadata Using Attributes*, http://msdn.microsoft.com/en-us/library/5x6cd29c.aspx, júl 2008

[22] Microsoft: *Implementing a UI Type Editor*, http://msdn.microsoft.com/en-us/library/53c49eck(VS.71).aspx, júl 2008

[23] Wikipedia, The Free Encyclopedia: *Performance analysis*, http://en.wikipedia.org/wiki/Profiler_%28computer_science%29, júl 2008

[24] Wikipedia, The Free Encyclopedia: *Unit testing*, http://en.wikipedia.org/wiki/Unit_testing, júl 2008

[25] Josef Pelikán: *JaGrLib - library for computer graphics*, http://cgg.ms.mff.cuni.cz/JaGrLib, júl 2008

Dodatok A

Obsah priloženého média

Na priloženom CD nájdeme:

- `Install` Inštalačné súbory.
  - `install.zip` Inštalačné súbory v archíve. K ich spusteniu si užívateľ musí byť istý, že má správne nainštalovanú knižnicu *.NET Framework* vo verzii 3.5.
  - `setup.exe` Inštalátor programu. Pokiaľ chýba zmienená knižnica, k jej prevzatiu je nutné pripojenie na internet.
- `Text` Tento text vo formáte *PDF*.
- `Examples` Videá a príklady určené k vysvetleniu použitia vytvorenej aplikácie.
- `Utils` Nástroje umožňujúce pohodlné generovanie vlastných experimentov.
- `Doc` Vygenerovaná dokumentácia k programu.
  - `ExperimentsLibrary` Dokumentácia knižnice *ExperimentsLibrary*.
  - `ExperimentsCreator` Dokumentácia programu *ExperimentsCreator*.
- `Src` Zdrojové kódy programu. Ich kompiláciu nájdeme v nasledujúcom dodatku.
- `readme.txt` Potrebné poznámky, kontakt.

Dodatok B

Kompilácia zdrojových kódov

Úplné zdrojové kódy nájdeme v adresári `Src` na priloženom médiu. K ich preloženiu potrebujeme *Microsoft Visual Studio 2008* prípadne aj systém *CMake*.

Dodali sme aj testovacie projekty. Nesmieme zmeniť ich relatívne pozície voči ostatným projektom, odkazujú sa ne. Program *MS VS* určený k testovaniu používateľských komponentov nespracúva s cesty obsahujúce medzery alebo diakritiku. Počas testovania knižnice `Diagram` musí byť celý projekt vo „vhodnom“ umiestnení.

Experiments library

CMake:

```
cd ExperimentsLibrary/cmake
cmake .
make
```

V aktuálnom adresári nájdeme potrebné statické knižnice pre projekt *Experiments library*. Tie je potrebné umiestniť tak, aby súbory `CMakeLists.txt` generované programom *ExperimentsCreator* k nim mali prístup.

Microsoft Visual Studio:

V adresári `ExperimentsLibrary\vsproject` nájdeme súbor obsahujúci projekty knižnice, `Experiments.sln`. Celé riešenie necháme preložiť v oboch módoch, *Debug* aj *Release*. Vytvorené statické knižnice potom nájdeme v adresároch `Debug` a `Release`, tie sú v zložke, kde je aj riešenie `Experiments.sln`. Knižnice prekopírujeme do adresárov `DebugLibs` a `ReleaseLibs`.

ExperimentsCreator

V dodaných zdrojových kódoch nájdeme aj zdrojové kódy tejto aplikácie. Zvolíme vhodný typ archívu a rozbalíme ho. V adresári `ExperimentsCreator` nájdeme súbor `ExperimentsCreator.sln`. Ten použijeme k preloženiu projektu.

Dodatok C

Problémy

Počas používania programu a kompilovania výstupov sa môže vyskytnúť niekoľko problémov, ktoré majú pôvod v rôznych konfiguráciách systémov. Ide najmä o kompilačné problémy. Preto pred použitím programu a každou úpravou behového prostredia odporúčame dodané statické knižnice *ExperimentsLibrary* nahradiť novými, čerstvo skompilovanými (adresáre `DebugLibs` a `ReleaseLibs`).

Kompilačné problémy vytvorených projektov.

- *Warning LNK4204: '...\vc90.pdb' is missing.* K odstráneniu tohoto varovania je nutné prekompilovať knižnicu *ExperimentsLibrary* a vytvorené statické knižnice prekopírovať do adresárov `DebugLibs` a `ReleaseLibs`. Je možné, že bude potrebné preložiť aj všetky behové rozšírenia zmienenej knižnice.

- *Unresolved external reference ...*, *Can not open include file ...* Nesprávne nastavené knižnice alebo adresáre k nim. Buď nastavujeme experimenty zvlášť alebo nastavenie vykonáme globálne pre prostredie. Cesty ku každému novému rozšíreniu je výhodné pridať do globálnej konfigurácie programu. Pokiaľ sme vytvorili vlastné rozšírenie, uistíme sa, že všetky metódy sú implementované.

- V prípade, že chceme generovať projekt pre *Microsoft Visual Studio 2005* upravíme šablónu `vcproj.tpl` (adresár `Templates`). Pôvodnú šablónu nahradíme súborom `vcproj2005.tpl`.

Behové problémy vytvorených projektov:

- *The wanted operation is not available upon this type.* Chyba vzniká v dvoch prípadoch. Keď do `TreeVariable` umiestnime typ, ktorého „wrapper“ nepodporuje porovnanie. Alebo použijeme agregáciu s nevhodným typom. Nemôžeme agregovať napríklad reťazce pomocou priemeru. Pre typ, ktorý má podporovať zvolené operácie upravíme behové prostredie. Vytvoríme novú špecializáciu šablóny `ValueWrapper` a upravíme súbor `Values.h` tak, aby obsahoval odkaz na vytvorenú zmenu. Skompilujeme projekt *ExperimentsLibrary*.

Problémy s programom ExperimentsCreator:

- *Could not find a part of the path '...'.* Objaví sa zvyčajne, keď konzolovú aplikáciu, ale aj GUI spustíme v inom pracovnom adresári, ako je inštalačný adresár. Následne zvyčajne chýbajú šablóny potrebné ku generovaniu výstupu. Môže sa objaviť, ak generujeme súbor do neexistujúcich adresárov.

- *Could not find the schema for experiment.* XML schémy neboli nájdené v adresári `Schemas`. Potom nemôžeme otvoriť žiadny experiment. Uistíme sa, že adresár existuje a program otvárame z inštalačného adresára.

- Niekedy sa stane, že experiment je zobrazený s červeným pruhom, aj keď už je správne nadefinovaný. Je to spôsobené tým, že nemáme zapnutú voľbu *Validity check*. Kontrolu správnosti je potrebné vyvolať manuálne.

Dodatok D

Policy classes a *SmartPointer*

Objekty typu *SmartPointer*[14] slúžia na poloautomatickú správu pamäti. Kedykoľvek na halde vytvoríme objekt, musíme ho po jeho použití uvoľniť. Cieľom smart pointra je rozoznať, kedy už objekt nie je potrebný a zmazať ho. V [14] je kvalitná ukážka ako naprogramovať vlastný smart pointer, ktorej sme sa pri implementácii a písaní tohoto dodatku držali. *SmartPointer* je rozumná náhrada za bežný ukazovateľ, ktorá rieši problém jeho mazania. Z hľadiska používania sa správa rovnako ako bežný ukazovateľ, poskytuje to isté rozhranie, čo v C++ implementujeme preťažovaním vhodných operátorov.

Hlavnou výhodou takéhoto ukazovateľa je jeho hodnotová sémantika. To znamená, že objekt je správne zmazaný, keď už nie je používaný. Napríklad lokálna premenná zaniká pri odchode z bloku jej definície. V prípade smart pointra túto sémantiku chápeme tak, že pri zániku všetkých referencií, je objekt, na ktorý sme sa odkazovali, zmazaný z pamäti.

Vo všeobecnosti existuje niekoľko možností ako tento cieľ dosiahnuť. V knižnici *Experiments* používame techniku zvanú *reference counting*. Pre rôzne účely sa hodia iné techniky, nám tento spôsob vyhovuje. Taktiež môžeme smart pointer obohatiť o rôzne spôsoby správy pamäti – mazania a vytvárania objektov, ich zdieľania, prípadne ho vybaviť podporou pre viac-vláknové aplikácie. Často je výhodné tieto schopnosti vypínať, či zapínať alebo kombinovať rôzne vlastnosti. K tomu, nielen v tomto prípade, môžeme použiť návrh využívajúci policy classes[14].

Policy classes alebo politiky nám umožnia jednoduchým spôsobom kombinovať rôzne schopnosti do jedného objektu. Napríklad v prípade smart pointra môžeme nájsť niekoľko ortogonálnych vlastností[1], ktoré chceme ovplyvniť. Ak máme ukazovateľ na pole, je nutné ho mazať pomocou operátora `delete []` a nie klasickým operátorom `delete`. Niekedy zase chceme povoliť alebo zakázať použitie `nil` uka-

[1] Do politík je vhodné zahŕňať len málo sa prekrývajúce vlastnosti, aby boli od seba navzájom nezávislé.

zovateľa. Vypnutie alebo zapnutie podpory viacerých vlákien môže byť ďalším z atribútov, na ktorý si chceme zachovať vplyv. Tieto vlastnosti sa menia v závislosti od použitia ukazovateľa a pomocou policy classes ich môžeme v dobe kompilácie[2] jednoducho „prepínať“.

Politiky v jazyku C++ implementujeme pomocou šablón. Každá politika – vlastnosť je typicky jeden argument šablóny triedy, ktorú parametrizujeme. Napríklad *SmartPointer* prijíma ako parameter šablóny dve politiky – overenie správnosti ukazovateľa a spôsob jeho mazania.

Samotné politiky sú implementované ako triedy, od ktorých môžeme našu triedu oddediť. Bežne nie sú určené k tomu, aby sme tvorili ich inštancie. V programoch jazyka C++ vo všeobecnosti platí, že pokiaľ získame ukazovateľ na predka, v tomto prípade na niektorú z politík, môžeme na ňu aplikovať operátor `delete`. Z dôvodu korektného vymazania celého objektu je nutné, aby politiky mali virtuálny deštruktor. Ak používame verejnú dedičnosť, je dobré si uvedomiť, že trieda ktorá používa politiky, môže s rôznymi politikami získať rôzne rozhranie.

```
struct Policy1A {
        virtual ~Policy1A(void) {
                ...
        }
        void method(void) const {
                ...
        }
};
struct Policy1B {
        virtual ~Policy1B(void) {
                ...
        }
        void method(void) const {
                ...
        }
};

template<class Policy1, class Policy2>
struct PolicyBasedClass : public Policy1, public Policy2 {
        ...
}
```

Druhou možnosťou je využitie metód, ktoré nájdeme v dodaných politikách. V tomto prípade predpokladáme, že všetky politiky budú obsahovať potrebné metódy. Niektorú funkcionalitu potom budeme delegovať na tieto metódy a tým dosiahneme cielený efekt – rôzne vlastnosti pre rozličné politiky. V knižnici *Experiments* uprednostňujeme tento spôsob použitia politík.

[2]Podobný efekt za behu dosiahneme použitím návrhového vzoru *Strategy*[18]. Policy classes sú narozdiel od *Strategy* statické.

```
struct CreateStrategyStatic {
        static void * create(void) {
                return &instance;
        }
};
struct CreateStrategyHeap {
        static void * create(void) {
                return new MyObject();
        }
};

struct CheckValidityNotNull {
        static bool isValid(void * ptr) {
                return ptr != 0;
        }
};
struct CheckValidityOk {
        static bool isValid(void * ptr) {
                return true;
        }
};

template<class CreateStrategy, class ValidityCheck>
struct PolicyBasedClass :
        public CreateStrategy,
        public ValidityCheck
{
        function bool create(void) {
                this->member = CreateStrategy::create();
                return ValidityCheck::isValid(this->member);
        }
        ...
};
```

Pomocou policy classes môžeme jednoduchým spôsobom vytvoriť triedu *SmartPointer*. Vlastnosti, ktoré overujú správnosť a vykonávajú mazanie objektu budú tvoriť rôzne politiky, tie predáme ako parametre `DeletePolicy` a `CheckPolicy`.

SmartPointer obsahuje:

- Odkaz na skutočný objekt.
- Odkaz na počet referencií. Počet odkazov sa nachádza na halde a je zdieľaný všetkými inštanciami odkazujúcimi na ten istý objekt. V prípade vytvorenia nového ukazovateľa sa počet odkazov zvýši, v prípade zániku zníži. Posledný zanikajúci *SmartPointer* vymaže objekt a počet z haldy.

- Metódy a operátory, ktoré napodobnia správanie sa bežného ukazovateľa, napríklad: `!`, `==`, `<`, `>`, `*`.

- Kopírovací konštruktor, operátor priradenia a deštruktor určené k správnej implementácii hodnotovej sémantiky.

Keď dochádza k mazaniu objektu, využijeme dodanú politiku `DeletePolicy`, ktorá sa postará o správne zdeštruovanie ukazovateľa. Počas práce s referovaným objektom, overujeme jeho správnosť politikou `CheckPolicy` – ovplyvňuje povolenie alebo zakázanie `nil`.

Pre úplnosť je nutné doplniť, že bežné implementácie triedy *SmartPointer* často umožňujú parametrizovať aj zdieľanie – kopírovanie objektov. Nemusia vykonávať len *reference counting*, ale v kopírovacom konštruktore a operátore priradenia môžu vytvárať hlboké kópie, či zanechávať len jediného výhradného vlastníka objektu tak, ako to robí napríklad štandardný `std::auto_ptr`.

V projekte *Experiments SmartPointer* využívame najmä pri kopírovaní hodnôt (*Value*) alebo ich popisov (*ValueDescriptor*, *ValueDescriptorPointer*). Pri skopírovaní smart pointra nedochádza ani k vytvoreniu kópie, ani k neželanému predčasnému zániku. Taktiež nezabudneme objekt vymazať v prípade, že naň stratíme referenciu. Pretože hodnoty môžu byť vytvorené kedykoľvek počas behu, bez použitia tejto techniky by bolo náročné sledovať ich existenciu. Správanie smart pointra nám úplne vyhovuje a preto sme sa ho rozhodli použiť.

Dodatok E

Knižnica *Diagram*

Program *ExperimentsCreator* svojou povahou viac-menej vyžaduje editáciu hierarchického experimentu v diagrame. Jeho použitím sa vytvorí pohodlné užívateľské prostredie na konfiguráciu experimentu a jeho zapojenie do hierarchie. K tomuto účelu bola vytvorená knižnica *Diagram*. S jej pomocou môžeme editovať v komponente *DiagramControl* ľubovoľný graf[1].

Knižnica *Diagram* obsahuje dve úrovne objektov:

- *Logický objekt* Vrchol, hrana alebo port, vytvárajú model editovaného grafu.
- *Reprezentácia logického objektu* Obsahuje spôsob, akým sa má objekt vykresliť a reagovať na vstupy od používateľa, polohu na osi z. Pamätá si referenciu na reprezentovaný logický objekt.

Všetky logické objekty sú odvodené od interfacu *IGraphObject*. Jeho rozšírením vznikli rozhrania pre vrchol *IVertex*, hranu *IConnection* a port *IPort*. Každá inštancia odvodená od *IVertex* obsahuje zoznam portov, vlastnosť (`Ports`), ktorými môže byť spojená s portmi ostatných vrcholov.

Spájanie portov funguje podobne ako spájanie experimentov – máme metódy, ktorými môže port spojenie odmietnuť, vytvoriť alebo zrušiť. V diagrame dochádza k spájaniu portov, nie vrcholov. Pokiaľ chceme porty spojiť alebo rozpojiť, najjednoduchšie to urobíme tak, že medzi nimi vytvoríme hranu (trieda *Connection*) a zavoláme jej metódu `Connect`, prípadne `Disconnect`.

Logické objekty obsahujú reprezentáciu grafu, ktorý sme v diagrame vytvorili. Diagram priamo nedefinuje spôsob jeho reprezentácie, ani nevykonáva grafové algoritmy. Implementáciou rozhraní však sami rozhodneme o spôsobe práce logických objektov. Môžeme si spraviť vlastné zoznamy uložených vrcholov alebo dokonca celú reprezentáciu grafu, napríklad maticou susednosti, a implementovať vlastné algoritmy.

[1] Knižnica nie je navrhnutá len k editácii hierarchických experimentov.

Je možné, že do diagramu potrebujeme vkladať objekty, ktoré neboli na prácu s ním navrhnuté a uspôsobené. V tomto prípade im jednoducho vytvoríme obálku – adaptér[18], ktorý splní požadované rozhranie a všetky metódy bude delegovať na obsiahnutý objekt. Napríklad spojením portov v diagrame sa ihneď spoja logické objekty, ktoré sú týmito portmi reprezentované. Výhodou takéhoto prístupu je, že diagram spája a rozpája logické objekty, udržuje graf synchronizovaný s diagramom. Takto sa synchronizujú aj objekty používateľskej aplikácie, nie je potrebné zaistiť nič navyše.

Často je dôležité vytvoriť si vlastný „skin“, spôsob akým majú byť reprezentácie logických objektov v diagrame vykreslené. K tomuto účelu v knižnici *Diagram* slúži trieda *IDiagramObjectFactory*. Obsahuje metódy, ktoré pre ľubovoľný typ logického objektu vrátia jeho reprezentáciu v diagrame. Napríklad, keď do diagramu vkladáme vrchol, komponent diagram využije túto továreň k získaniu jeho reprezentácie.

Základom pre reprezentáciu objektu v diagrame je interface *IDiagramObject*. Definuje reakcie na užívateľské akcie – klik, pohyb myši, stlačenie klávesu, ale aj na vymazanie objektu z diagramu. Objekt v diagrame môže byť vo vybratom stave, potom ho je nutné vykresliť odlišne. Bežne sa takýto objekt vykreslí so sivým rámčekom. Z tohoto dôvodu máme dve metódy určené k vykresleniu reprezentácie.

Každá reprezentácia vracia jej obalový obdĺžnik, vlastnosť `BoundingArea`. Využijeme ju počas prekresľovania diagramu a pri reagovaní na udalosti myši. Metódu `ContainsPoint` použijeme k zisteniu, či konkrétny bod z vrátenej oblasti patrí objektu. Napríklad, čiara spájajúca porty nie je rozložená v celej `BoundingArea`, iba v jej časti, ktorú si sami určíme. Len v tejto časti bude objekt informovaný o udalostiach napríklad myši. Počas prekresľovania komponentu *DiagramControl* je vlastnosť `BoundingArea` použitá k určeniu, či je nutné objekt znova vykresliť alebo nie.

Knižnica *Diagram* v sebe obsahuje štandardné reprezentácie vrcholu, hrany a portu, ktoré môžeme použiť, prípadne pozmeniť. Obsahuje aj triedy, ktoré menia ich správanie a vykreslenie, jedná so o objekty odvodené od *DecoratorObject*. Vlastnú reprezentáciu môžeme zabaliť do objektu, ktorý ňou bude hýbať – *MoveDecorator*, meniť jej šírku – *WidthDecorator* alebo ju vykreslí s rámčekom – *SelectedDecorator*.

V diagrame môžeme zmeniť aj mechanizmus zapájania objektov – trieda *ConnectionManager*. Nemusíme editovať len grafy – binárne relácie, ale aj n-árne relácie. Potom však musíme doplniť vhodné reprezentácie spojení a implementovať porty tak, aby bolo ich správanie intuitívne. O vytvorení nového spojenia sme informovaní udalosťou `ConnectionMade` rozhrania *IConnectionManager*.

Selected	Práve vybraný objekt v diagrame, `null` ak neexistuje.
SelectedOnTop	Ovplyvňuje to, či vybraný objekt bude na vrchole osi z.
Antialias	Reprezentácie sa pri kreslení riadia touto vlastnosťou.
Zoom	Aktuálne zväčšenie, 1 znamená identitu.
BackColor	Farba pozadia diagramu.
ShowGrid	Zobrazovanie mriežky.
MinXDiff, MinYDiff	Odstupy medzi bodmi mriežky na jednotlivých osiach.
ObjectsFactory	Továreň reprezentácií určujúca „skin“.
ConnectionManager	Objekt synchronizujúci spájanie portov.

Tabuľka E.1: Zoznam vlastností komponentu *DiagramControl*.

Zoznam obrázkov

2.1 Hierarchický experiment 8

3.1 Hlavné okno programu ExperimentsCreator 14
3.2 Práca s diagramom 15
3.3 Nastavenia diagramu 18
3.4 Menu edit 19
3.5 Dialóg určený k nastaveniu výstupu 20

4.1 Hierarchický experiment – príklad odkázania sa na premenné. 24
4.2 Klauzula `output_types` experimentu *Code block* 30
4.3 Klauzula `output` experimentu *Code block* 30
4.4 Definícia sady dát experimentu *Data collector* 31

5.1 Formulár určený k spracovaniu výstupných dát. 36
5.2 Príklad tabuľky vygenerovanej programom *ExperimentsCreator*. . . . 37
5.3 Príklad grafu vygenerovaného programom *ExperimentsCreator*. . . . 38

6.1 Mapovanie hodnôt uložených v *TreeVariable*. 46
6.2 Stavy akcie (IExperimentAction) vzhľadom k operáciám Undo a Redo. 53

7.1 Formulár určený k definovaniu používateľského experimentu. 60

Zoznam tabuliek

4.1 Príklady odkazovania sa na premenné cestou k nim. 24
4.2 Príklady zadania požiadaviek na vstupné premenné. 25
4.3 Prehľad vlastností experimentu *Composite* 26
4.4 Prehľad vlastností experimentu *Loop* 27
4.5 Prehľad vlastností experimentu *Collector* 27
4.6 Prehľad vlastností experimentu *Object creator* 29
4.7 Prehľad vlastností experimentu *Method caller* 29
4.8 Prehľad vlastností experimentu *Code block* 30
4.9 Prehľad vlastností experimentu *Data collector* 32
4.10 Prehľad vlastností experimentu *Directory lister* 33
4.11 Prehľad vlastností experimentu *Line enumerator* 33
4.12 Prehľad vlastností experimentu typu *Exponential loop* 34

6.1 Triedy implementujúce rozhranie *IExperimentSaver*. 51

7.1 Požiadavky triedy *CustomExperiment* na experiment. 59
7.2 Nastavenia používateľského experimentu. 61

E.1 Zoznam vlastností komponentu *DiagramControl*. 78