Semestrální práce z předmětu X33TSW

Miroslav Hrúz, Ondřej Kuželka {hruzm1, kuzelo1}@fel.cvut.cz

1. Úvod

V tomto dokumentu popisujeme průběh testování prováděného v rámci cvičení k předmětu X33TSW Testování a diagnostika softwaru. Jako testovaný program používáme jednoduchou aplikaci vytvořenou v programovacím jazyce Java s názvem Kalkulačka.

Nejprve shrnujeme systémové požadavky, případy užití a test cases a naznačujeme, jakým způsobem je možné tyto spravovat v programu Rational Requisite Pro. Následně popisujeme strukturu testované aplikace prostřednictvím class diagramů a sequence diagramů, které rovněž komentujeme v textu. Dále ilustrujeme tvorbu Cause-and-Effect diagramu na jednoduchém problému týkajícím se naší testované aplikace. Pomocí Netbeans Profileru následně analyzujeme kritická místa aplikace, co se týče její výkonnosti i paměťových nároků. Pomocí open-source programu CodeCover analyzujeme pokrytí kódu testované aplikace několika testy a diskutujeme aplikovatelnost získaných výsledků pro náš konkrétní případ. V programu Rational Manual Test vytváříme sadu testů. V programu Rational Robot vytváříme několik automatizovaných testů, které následně použijeme pro testování naší aplikace. Pro vyzkoušení si spravování výsledků testů provedených s pomocí programů Rational Manual Test a Rational Robot používáme program Rational Test Manager. Nakonec na uměle vytvořeném problému odvozujeme propagaci variance podle Haralicka.

2. Požadavky na program kalkulačka

Funkční požadavky

SR1: Vkládání výrazů

Program umožní uživateli vkládat matematické výrazy prostřednictvím textového pole v horní části okna programu. Za korektní je považován jakýkoliv matematický výraz obsahující čísla, proměnné, funkce *sin*, *asin*, *cos*, *acos*, *tan*, *atan*, *log*, *exp* a konstanty *pi*, *e*.

SR2: Zjednodušování výrazů

Po stisknutí tlačítka „Zjednodušit “, případně po stisknutí klávesy Enter v textovém poli pro zadávání výrazů, se program pokusí zjednodušit výraz zadaný uživatelem. Pokud je zadaný výraz syntakticky nesprávný, program to uživateli oznámí prostřednictvím dialogového okna. V případě, že je zadaný výraz syntakticky korektní, je výsledek zjednodušení tohoto výrazu vypsán v textovém poli ve spodní části okna programu a je uložen do proměnné, jejíž jméno je vypsáno společně se zjednodušeným výrazem.

SR3: Roznásobování výrazů

Po stisknutí tlačítka „Expandovat “ program roznásobí všechny závorky přítomné ve výrazu zadaném uživatelem a pokusí se výsledný výraz následně zjednodušit. Pokud je zadaný výraz syntakticky

nesprávný, program to uživateli oznámí prostřednictvím dialogového okna. V případě, že je zadaný výraz syntakticky korektní, je výsledek roznásobení závorek a zjednodušení tohoto výrazu vypsán v textovém poli ve spodní části okna programu a je uložen do proměnné, jejíž jméno je vypsáno společně se zjednodušeným výrazem.

SR4: Derivování výrazů

Po stisknutí tlačítka „Derivovat podle proměnné " program zderivuje výraz zadaný uživatelem podle proměnné zadané v daném textovém poli a pokusí se výsledný výraz následně zjednodušit. Pokud je zadaný výraz syntakticky nesprávný, program to uživateli oznámí prostřednictvím dialogového okna. V případě, že je zadaný výraz syntakticky korektní, je výsledek derivování a následného zjednodušení tohoto výrazu vypsán v textovém poli ve spodní části okna programu a je uložen do proměnné, jejíž jméno je vypsáno společně se zjednodušeným výrazem.

SR5: Využívání předchozích výsledků – funkce paměť

Program umožní přistupovat k výsledkům předchozích operací prostřednictvím názvů proměnných vypisovaných společně s těmito výsledky. Pokud se v nějakém výrazu vyskytne název takovéto proměnné, bude tato proměnné nahrazena odpovídajícím obsahem a na takto vzniklém výrazu bude provedena uživatelem požadovaná operace.

SR6: Nastavení počtu zobrazovaných desetinných míst

Program umožní nastavit počet zobrazovaných desetinných míst až do počtu daného způsobem uložení datového typu double. Po výběru položky „Desetinná místa" z menu „Soubor", se uživateli zobrazí dialogové okno s textovým polem, do nějž uživatel vepíše požadovaný počet zobrazovaných desetinných míst. Nově zadaný počet zobrazovaných desetinných míst se projeví až ve výpisu nějaké nové operace.

SR7: Ukončení programu

Program bude možné ukončit jednak standardně kliknutím na křížek v pravém horním rohu, a jednak zvolením položky „Konec" v nabídce „Soubor".

Případy užití

UC1: Zjednodušování výrazů

Uživatel zadá do textového pole v horní části okna programu výraz, který má být zjednodušen. a klikne na tlačítko „Zjednodušit". V hlavním textovém poli umístěném v dolní části okna programu se zobrazí korektní výsledek, který je rovněž přiřazen do nějaké nové proměnné.

UC2: Derivování výrazu

Uživatel zadá do textového pole v horní části okna programu výraz, který má být derivován. Do textového pole umístěného vpravo vedle tlačítka „Derivovat podle proměnné" zadá proměnnou, podle níž se má derivovat, a klikne na tlačítko „Derivovat podle proměnné". V hlavním textovém poli umístěném v dolní části okna programu se zobrazí korektní výsledek, který je rovněž přiřazen do nějaké nové proměnné.

UC3: Využití funkce paměť

Předpokladem využití této funkce je to, že uživatel již provedl alespoň jeden výpočet. Je-li tomu skutečně tak, pak uživatel do vytvářeného vzorce vloží název proměnné, která je vypsána u požadovaného zpracovaného výrazu v hlavním textovém poli v dolní části okna aplikace, a klikne na jedno z tlačítek „Zjednodušit“, „Expandovat“ nebo „Derivovat podle proměnné“ v závislosti na tom, jakou operaci chce provádět. Výsledný výraz vypsaný v hlavním textovém poli pak je výsledkem dosazení odpovídajícího výrazu za danou proměnnou a aplikací požadované operace na takto vzniklý výraz.

Test cases

TC1: Zjednodušení výrazu

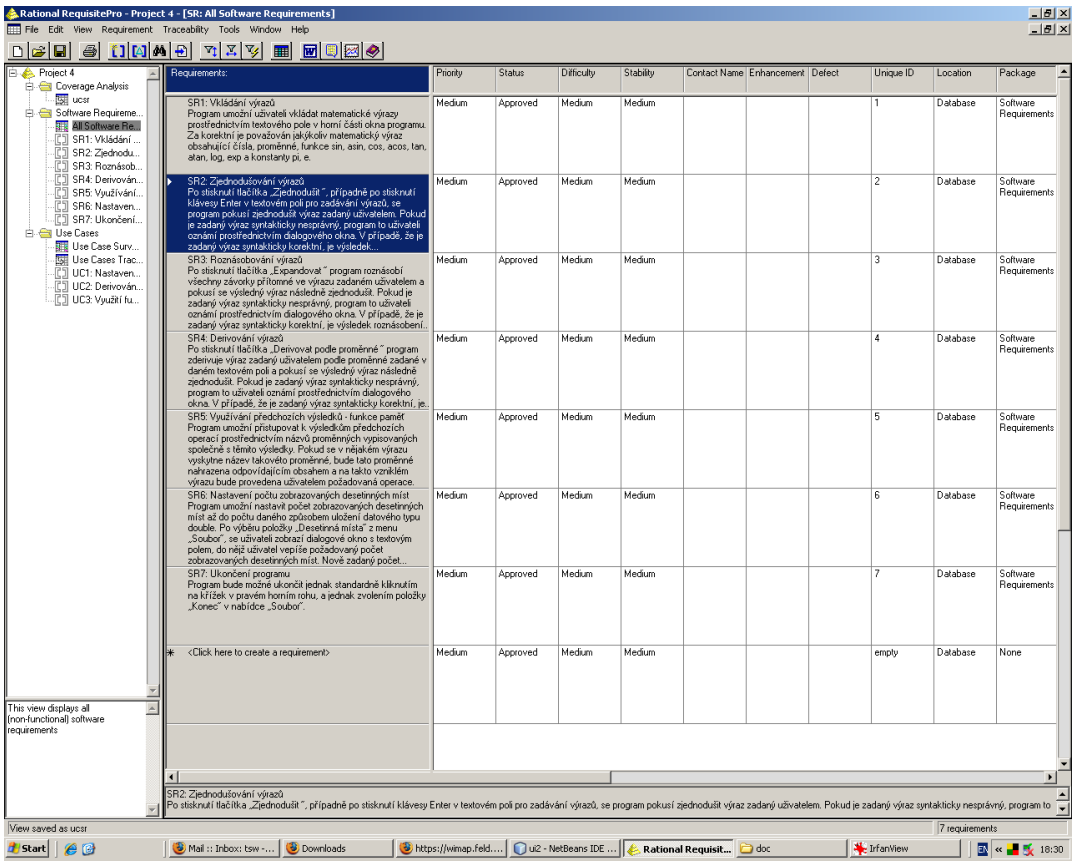
Uživatel zadá do textového pole v horní části okna programu výraz „1+1“, který má být zjednodušen. a klikne na tlačítko „Zjednodušit“. V hlavním textovém poli umístěném v dolní části okna programu se zobrazí korektní výsledek „2“, který je rovněž přiřazen do nějaké nové proměnné.

TC2: Derivování výrazu

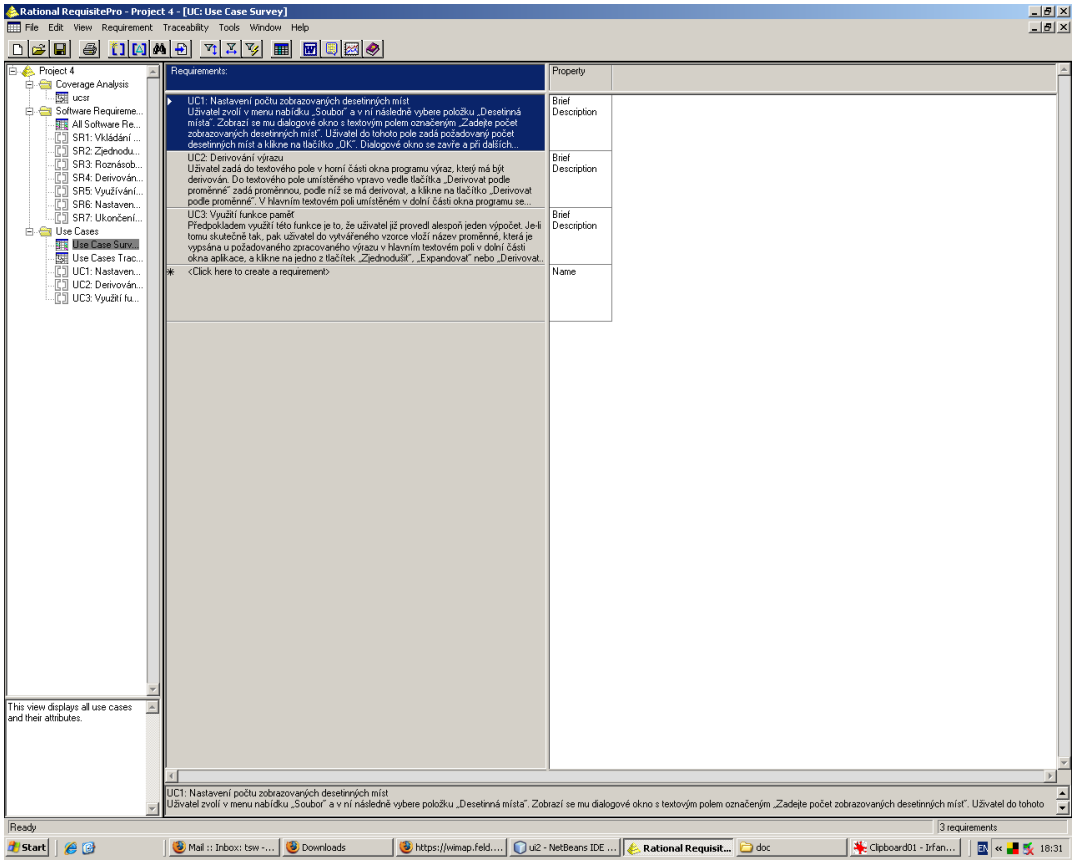
Uživatel zadá do textového pole v horní části okna programu výraz „x+sin(x)“, který má být derivován. Do textového pole umístěného vpravo vedle tlačítka „Derivovat podle proměnné“ zadá „x“ a klikne na tlačítko „Derivovat podle proměnné“. V hlavním textovém poli umístěném v dolní části okna programu se zobrazí výsledek „1+cos(x)“, který je rovněž přiřazen do nějaké nové proměnné.

TC3: Využití funkce paměť

Uživatel provede postup popsaný v TC1. Následně do textového pole v horní části okna napíše „ansX+1“, kde X nahradí číslem proměnné, do níž program uložil výsledek předchozího výpočtu. Program zobrazí v hlavním okně výsledek „3“.



Obrázek 1: Softwarové požavky v programu RequisitePro



Obrázek 2: Případy užití programu RequisitePro

UML model testovného sw

UML je stavebním kamenem analýzy sw projektu, pomocí něhož se modelují procesy, děje a činnosti, které bude sw vykonávat. Pokud nepoužíváme jako metodiku Extrémní programování, měla by vlastní programovací činnosti předcházet analýza. Všechny druhy diagramů v UML specifikaci můžeme rozdělit do následujících 2 druhů, a to na

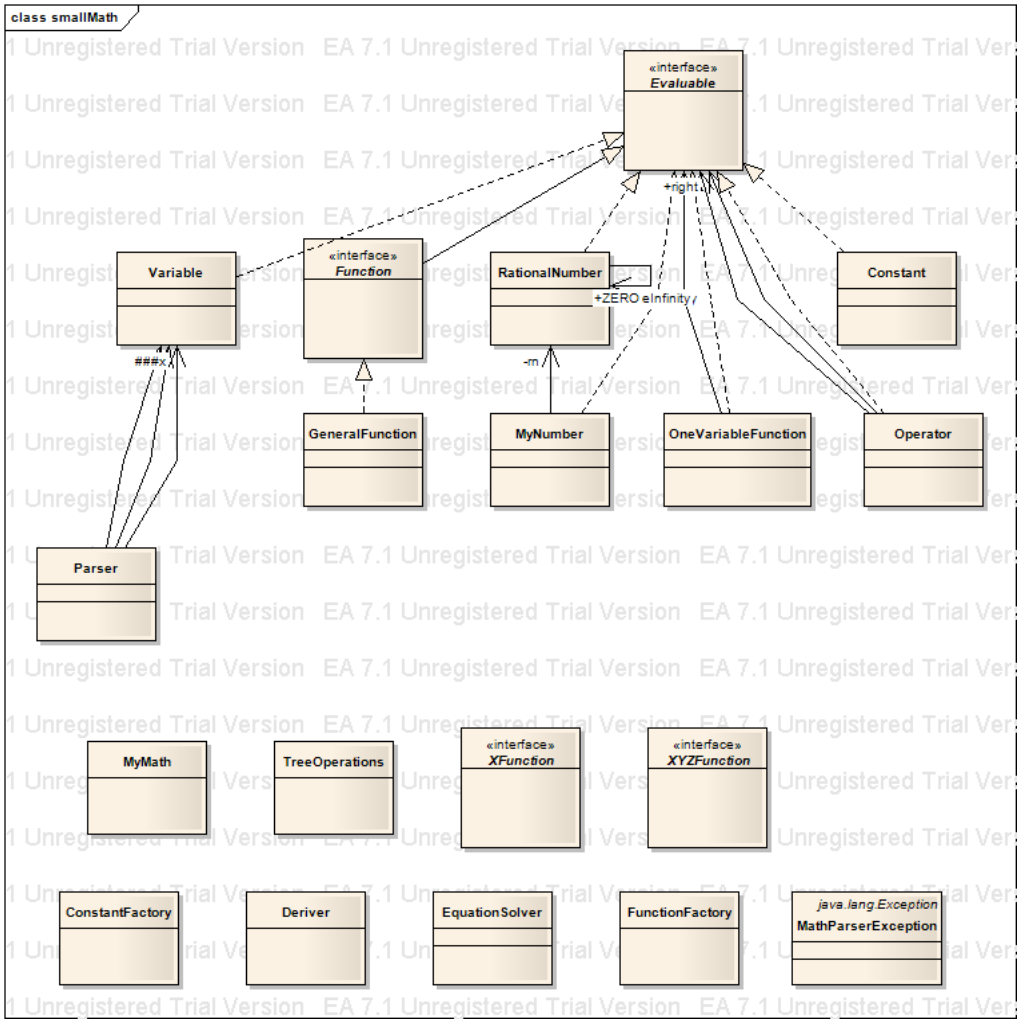
- diagramy modelu chování – behavioral modelling diagrams a
- diagramy strukturního modelu – structural modelling diagrams

Všechny diagramy zde obsažené jsou vytvořeny pomocí nástroje Enterprise Architect od společnosti Sparx System z 30denní trial verze. Program je možné stáhnout na adrese http://www.sparxsystems.com/products/ea.html

Class diagram

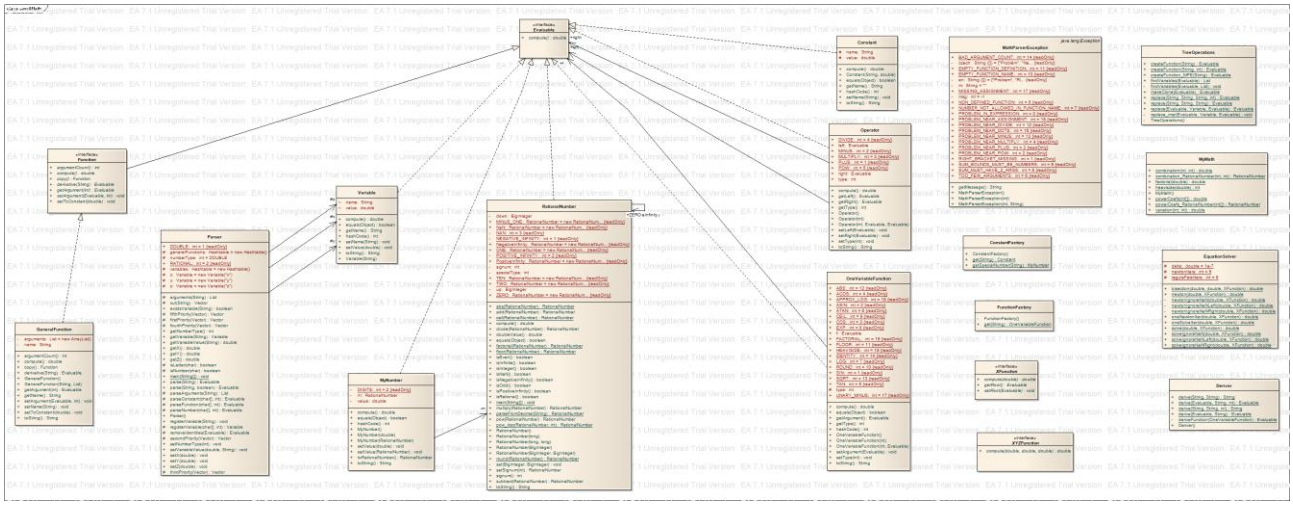
Class diagram spadá do druhé kategorie. Často pomocí něj definujeme svět tzv. Bussiness Objectů, tj. entit, které nás v obchodě zajímají především (můžeme setkat také s pojmy Domain Model, ER model). Class diagram je abstrakce realizace tříd a vztahů mezi nimi, které jsou v OOP/OOD důležité, jejich implementace je v tomto pohledu nepodstatná.

Jádro package smallMath



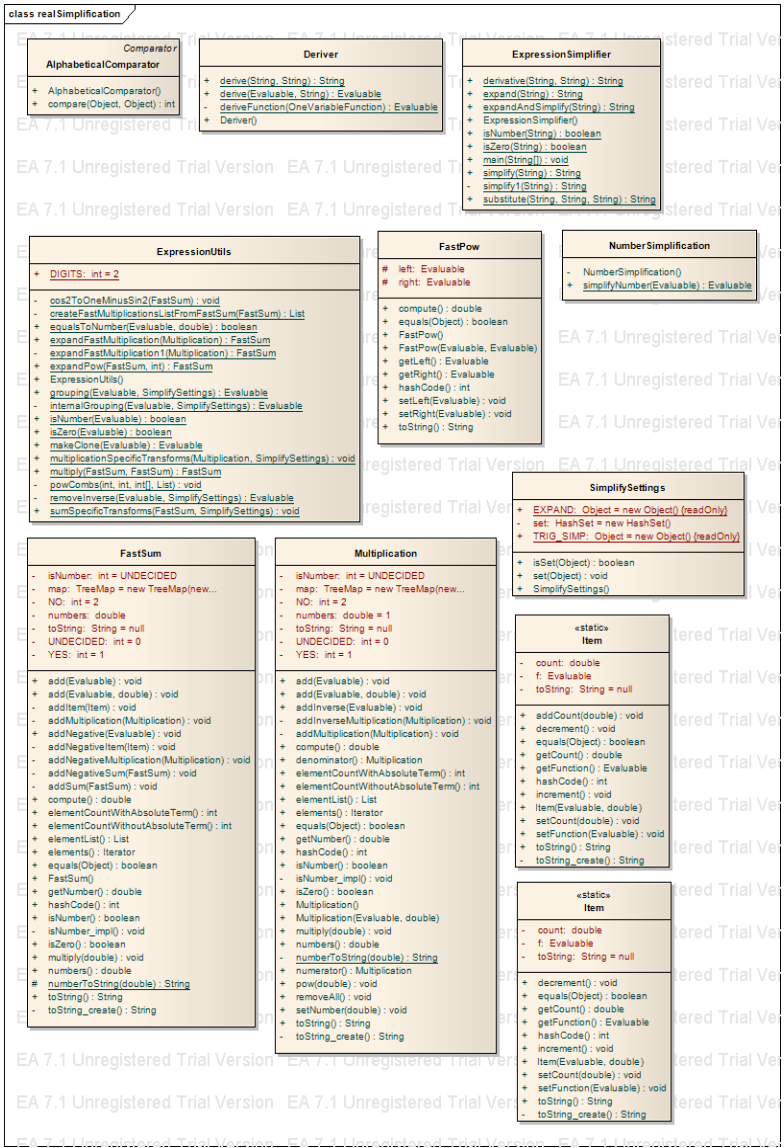
Package smallMath tvoří jádro aplikace Tento diagram abstrahuje i od property fieldů (atributů) a metod tříd, které v jsou v tomto kontextu nepodstatné. Jedná se o AST (Abstract syntax tree) překladače vlastního jazyka stroje typu kalkulačka, který využívá při překladu (parsování) třída Parser. Vidíme zde **dědičnost** (opak generalizace), od abstraktního předka (v našem případě rozhraní) Evaluable, **asociaci** (kompozici) např. u třídy Operator.

Pro úplnost uvádíme také úplný class diagram package smallMath, kde jsou kromě základního modelu tříd a závislostí uvedeny také třídní atributy s obory viditelností, metody, statické konstanty a podobně.



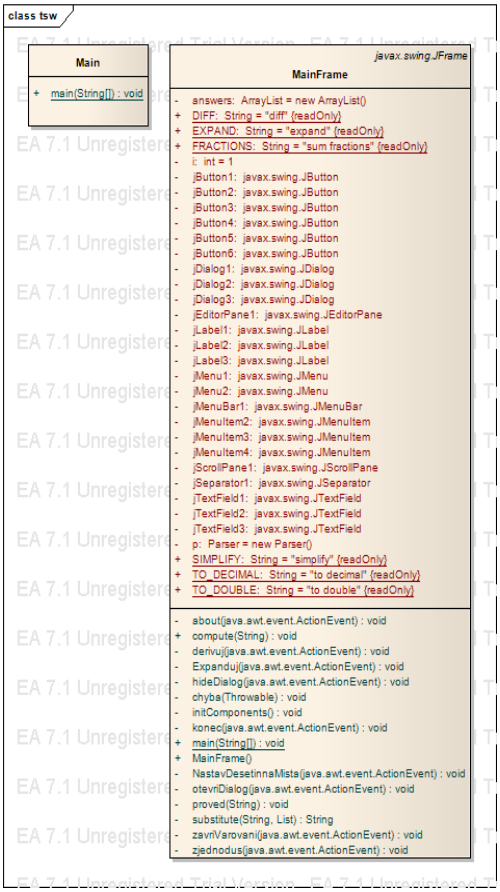
package realSimplification

V tomto package se nachází servisní vrstva aplikace, chcete-li aplikační logika. Jsou zde třídy reprezentující realizaci jednotlivých user stories (requirementů) jako jsou: derivate, simplify, evaluate. Package je tvořen převážně třídami podle návrhového vzoru Helper, třídami, jež mají jen statické metody.



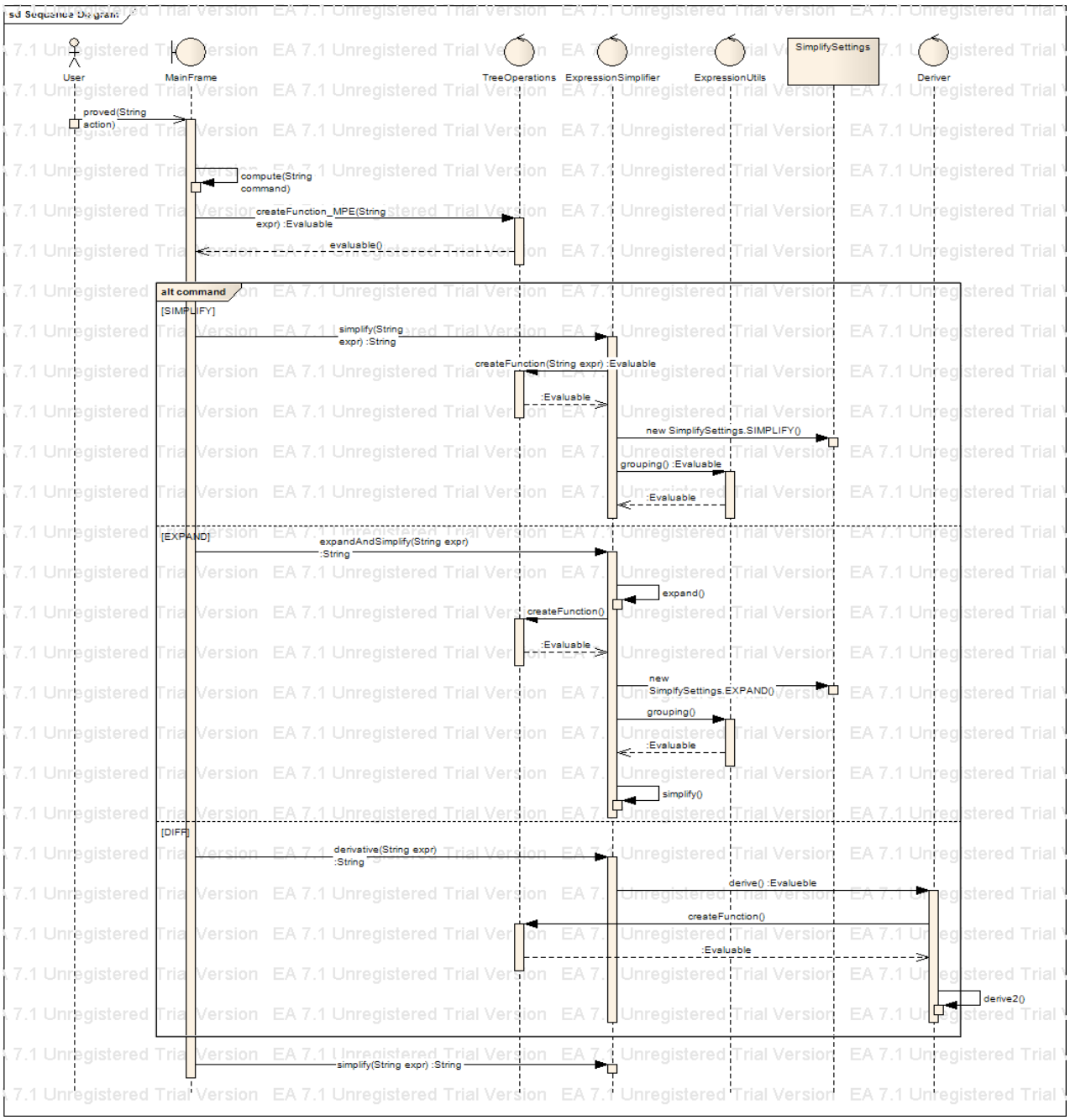
package tsw

V tomto package je naimplementováno GUI pro interakci s programovým vybavením zapsaným v předchozích 2 class diagramech. Jedná se o hlavní třídu MainFrame, která jak je vidět z diagramu extenduje od javax.swing.JFrame, tudíž jde o Swingovskou aplikaci.



Sequence diagram

Sekvenční diagram patří do první výše zmíněné kategorie. Pomocí tohoto dobře modelujeme interakci mezi jednotlivými entitami (třídami). Jednotlivé třídy spolu komunikují pomocí předávání zpráv, jednak synchronního, druhak asynchronního. U synchronních zpráv zde mluvíme o volání (call), zprávy můžeme posílat rekurzivně. Tento diagram ukazuje průběh zpracování user requirementů, kdy uživatel, aktor z Use case diagramu (modelu jednání), klikne na button expanduj, zjednoduš, derivuj a očekává správný výsledek svého dotazu. „První na ráně“ je vstupní bránou do aplikace, v terminologii sekvenčního diagramu říkáme Boundary element. Dál zde vidíme kontroléry (Controll elements), v našem případě třídy odpovídající návrhovému vzoru Helper a dále třídu SimplifySettings, která nespadá do žádné

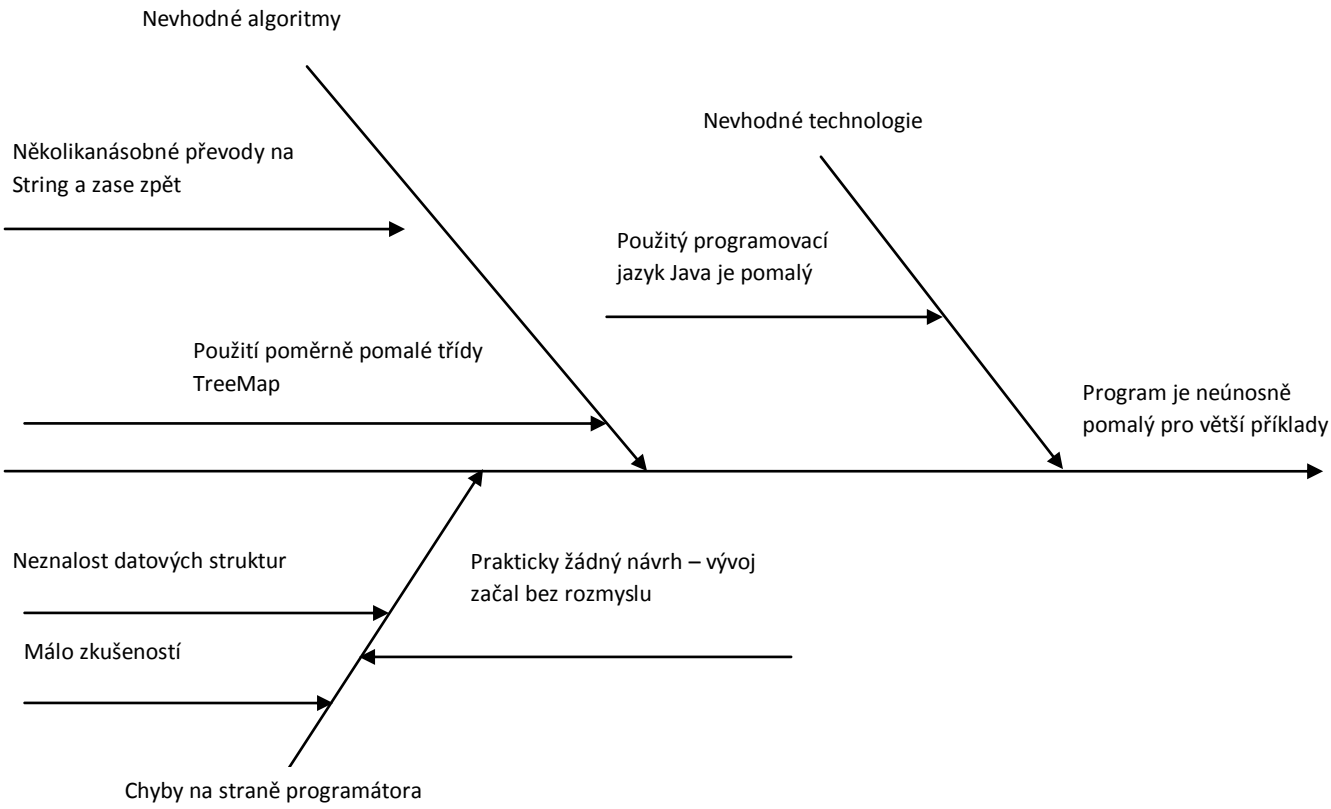


kategorie, je použita pro zaobalení parametrů použitách při zjednodušení výrazu.

3. CE-diagram

CE-diagram (Cause-and-Effect-Diagram) je grafická pomůcka umožňující zobrazit různé příčiny nějakého pozorovaného jevu (v našem případě špatné výkonnosti testovaného programu). CE-diagram se typicky vytváří tak, že se nejprve nakreslí horizontální šipka, na jejímž konci je jev, jehož příčiny, chceme najít, a k této šipce se nakreslí další šipky, které označují poměrně obecné příčiny (v našem případě například „nevhodné algoritmy“). K těmto šipkám označujícím obecné příčiny se pak přidají šipky s konkrétnějšími příčinami a tak se postupuje, dokud není diagram dostatečně detailní. CE-diagramu se kvůli jeho tvaru také někdy říká „rybí kost“.

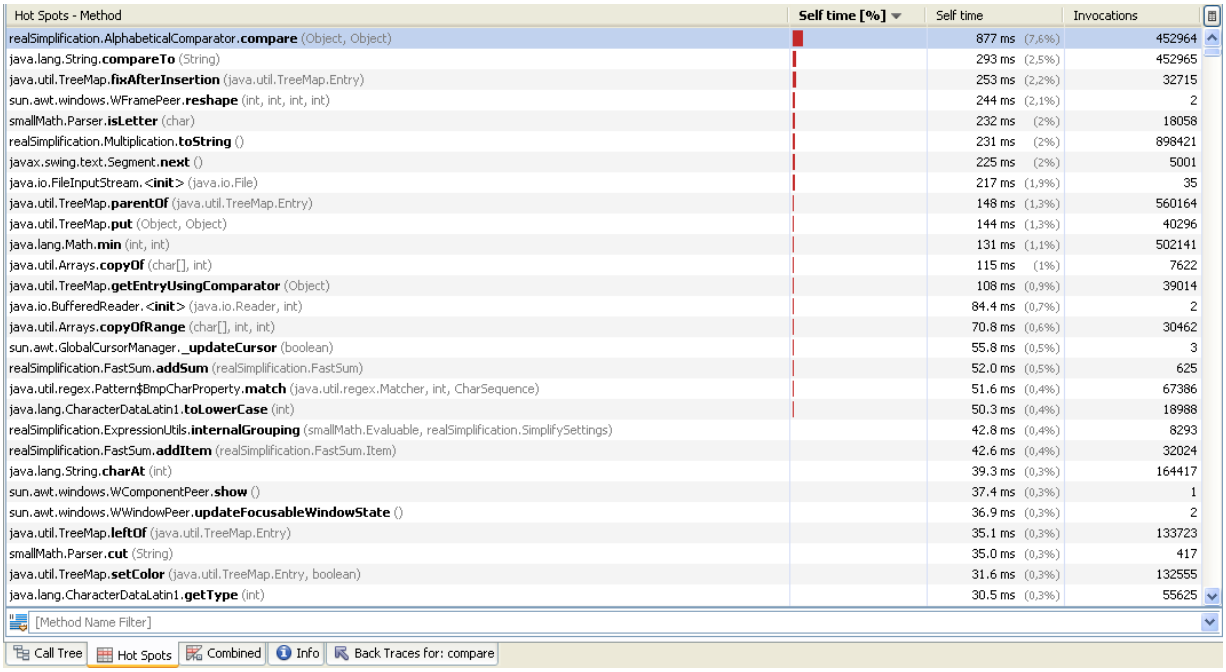
Pomocí CE-diagramu jsme se pokusili zachytit možné příčiny špatné výkonnosti testovaného programu Kalkulačka. Možné příčiny jsme rozdělili do několika skupin: Nevhodné algoritmy, Chyby na straně programátora a nevhodně použité technologie.



4. Analýza výkonnosti programu pomocí profileru

Pro analýzu výkonnosti programu Kalkulačka jsme využili profiler dodávaný s integrovaným vývojovým prostředím Netbeans 6. Bohužel vzhledem k tomu, že uživatelské rozhraní programu Kalkulačka bylo vytvořeno v Javě 1.6, nebylo možné využít doporučeného profilovacího programu IBM Quantify. Nicméně i s tímto profilerem poskytovaným společně s Netbeans se nám podařilo určit části aplikace kritické pro výkon.

Ukázalo se, že nejvíce procesorového času je spotřebováno na udržování výrazů ve stromových strukturách – konkrétně nejvíce strojového času spotřebovala podle výsledků profileru metoda *realSimplification.AlphabeticalComparator.compare(Object, Object)*. Ta byla nejčastěji volána při vkládání prvků do instancí třídy *java.util.TreeMap* a při volání metody téže třídy *java.util.TreeMap.containsKey()*, o čemž se můžeme přesvědčit ve výpisu stack-traces zobrazených na Obrázku 2. Pokud bychom tedy měli v úmyslu aplikaci zrychlit, bylo by vhodné buď zcela odstranit použití třídy java.util.TreeMap, anebo alespoň snížit četnost jejího použití na nutné minimum. Co je však nejdůležitější, je poznání, že bez použití profileru bychom byli pouze velice těžko schopni zjistit, která část aplikace by měla být optimalizována s ohledem na výkon.

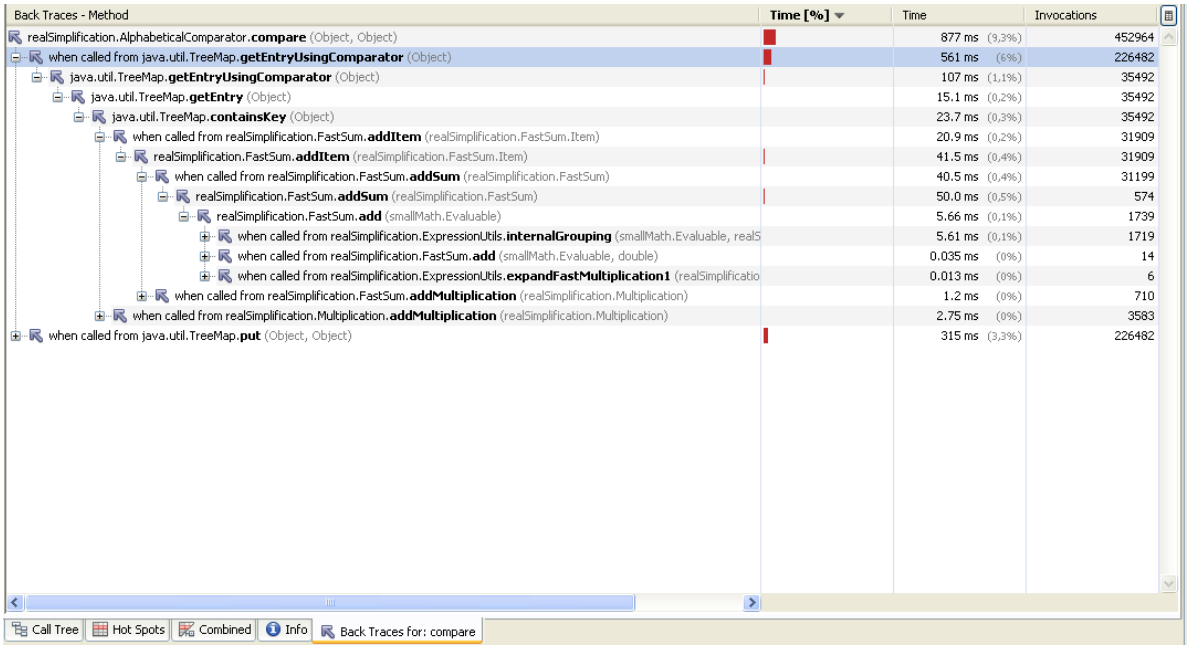


Hot Spots - Method	Self time [%]	Self time	Invocations
realSimplification.AlphabeticalComparator.**compare** (Object, Object)		877 ms (7,6%)	452964
java.lang.String.**compareTo** (String)		293 ms (2,5%)	452965
java.util.TreeMap.**fixAfterInsertion** (java.util.TreeMap.Entry)		253 ms (2,2%)	32715
sun.awt.windows.WFramePeer.**reshape** (int, int, int, int)		244 ms (2,1%)	2
smallMath.Parser.**isLetter** (char)		232 ms (2%)	18058
realSimplification.Multiplication.**toString** ()		231 ms (2%)	898421
javax.swing.text.Segment.**next** ()		225 ms (2%)	5001
java.io.FileInputStream.**<init>** (java.io.File)		217 ms (1,9%)	35
java.util.TreeMap.**parentOf** (java.util.TreeMap.Entry)		148 ms (1,3%)	560164
java.util.TreeMap.**put** (Object, Object)		144 ms (1,3%)	40296
java.lang.Math.**min** (int, int)		131 ms (1,1%)	502141
java.util.Arrays.**copyOf** (char[], int)		115 ms (1%)	7622
java.util.TreeMap.**getEntryUsingComparator** (Object)		108 ms (0,9%)	39014
java.io.BufferedReader.**<init>** (java.io.Reader, int)		84.4 ms (0,7%)	2
java.util.Arrays.**copyOfRange** (char[], int, int)		70.8 ms (0,6%)	30462
sun.awt.GlobalCursorManager.**_updateCursor** (boolean)		55.8 ms (0,5%)	3
realSimplification.FastSum.**addSum** (realSimplification.FastSum)		52.0 ms (0,5%)	625
java.util.regex.Pattern$BmpCharProperty.**match** (java.util.regex.Matcher, int, CharSequence)		51.6 ms (0,4%)	67386
java.lang.CharacterDataLatin1.**toLowerCase** (int)		50.3 ms (0,4%)	18988
realSimplification.ExpressionUtils.**internalGrouping** (smallMath.Evaluable, realSimplification.SimplifySettings)		42.8 ms (0,4%)	8293
realSimplification.FastSum.**addItem** (realSimplification.FastSum.Item)		42.6 ms (0,4%)	32024
java.lang.String.**charAt** (int)		39.3 ms (0,3%)	164417
sun.awt.windows.WComponentPeer.**show** ()		37.4 ms (0,3%)	1
sun.awt.windows.WWindowPeer.**updateFocusableWindowState** ()		36.9 ms (0,3%)	2
java.util.TreeMap.**leftOf** (java.util.TreeMap.Entry)		35.1 ms (0,3%)	133723
smallMath.Parser.**cut** (String)		35.0 ms (0,3%)	417
java.util.TreeMap.**setColor** (java.util.TreeMap.Entry, boolean)		31.6 ms (0,3%)	132555
java.lang.CharacterDataLatin1.**getType** (int)		30.5 ms (0,3%)	55625

[Method Name Filter]

Call Tree | Hot Spots | Combined | Info | Back Traces for: compare

Obrázek 3: "Hot Spots"



Obrázek 4: "Stack traces "

5. Test chyb ve správě paměti

V Javě k memory leakům nemůže dojít, resp. pokud dojde, je to čistě v implementaci Garbage Collectoru a tu na aplikační úrovni nejde ovlivnit. Co se týče chyb v nesprávném používání paměti, ta byla profilována pomocí nástroje Netbeans Profiler v předchozím bodu a žádné chyby nalezeny nebyly. Netbeans Profiler jsme použili z toho důvodu, že stejně jako v předchozím bodě nebylo možné použít program z balíku IBM Rational kvůli použití Javy verze 1.6.

6. Test částí kódu, které nejsou nikdy volány

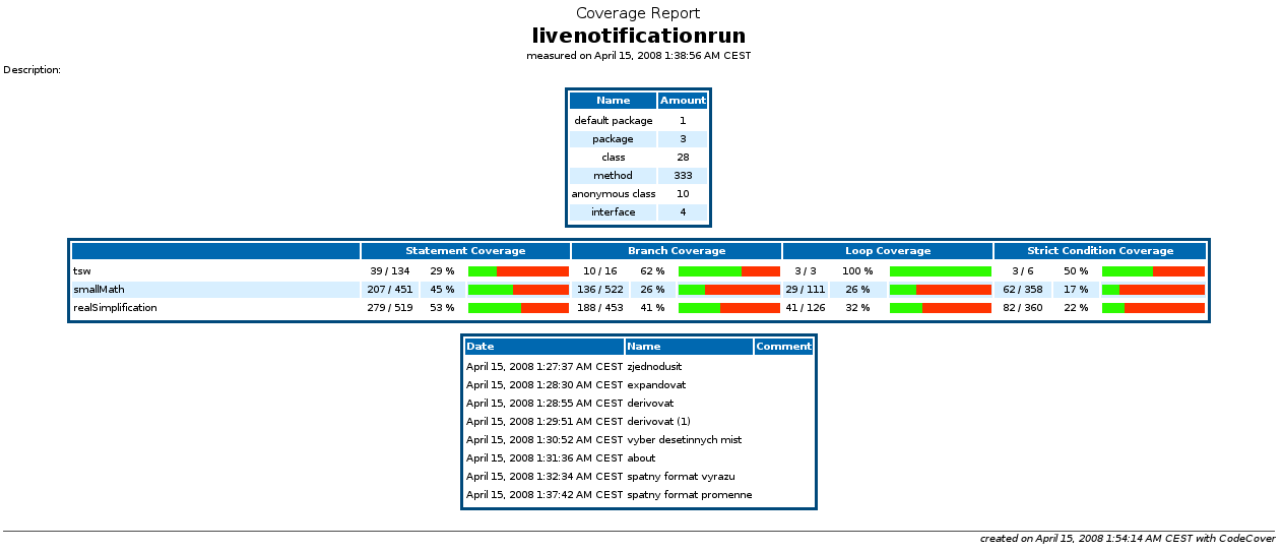
Předně pod pojmem code coverage rozumíme poměr mezi kódem, který je pokryt jednotkovými testy a celým kódem. Jednotkový test (neboli Unit test či chcete-li white-box test) by měl psán vývojář sám. Existuje dokonce novodobá agilní metodika TDD (Test-driven developement), která říká dokonce, aby se testy psaly nejdříve (Test-first approach). Tento přístup se nedá bohužel použít všude, resp. otestování někdy zabere více úsilí než samotná implementace a v některých případech to není možné vůbec. V praxi navíc se mnohdy unit testy nepíší ani po implementaci. Spolu s unit testy jde ruku v ruce refaktorování, kdy pokud jsou napsané testy, snadno (čti rychle) se dá ověřit, zda je pořád program po změně návrhu stejně funkční.

V této aplikace se unit testy nepsaly test-first, dokonce nejsou psané ani po implementaci, čili zde tato metrika nemá význam.

V našem případě pod pojmem code coverage rozumíme test na kusy kódu, které nejsou nikdy volány. Pokud se jedná o knihovnu nebo framework, tak tam to na škodu samozřejmě není.

Pro následující report byl použit open-sourceový nástroj Code cover, který je

spustitelný jednak standalone, druhak jako plugin do nejoblíbenějšího vývojového prostředí Eclipse IDE. Tool je dostupný na adrese http://codecover.org/



Coverage Report

livenotificationrun

measured on April 15, 2008 1:38:56 AM CEST

Description:

Name	Amount
default package	1
package	3
class	28
method	333
anonymous class	10
interface	4

	Statement Coverage		Branch Coverage		Loop Coverage		Strict Condition Coverage	
tsw	39 / 134	29 %	10 / 16	62 %	3 / 3	100 %	3 / 6	50 %
smallMath	207 / 451	45 %	136 / 522	26 %	29 / 111	26 %	62 / 358	17 %
realSimplification	279 / 519	53 %	188 / 453	41 %	41 / 126	32 %	82 / 360	22 %

Date	Name	Comment
April 15, 2008 1:27:37 AM CEST	zjednodusit	
April 15, 2008 1:28:30 AM CEST	expandovat	
April 15, 2008 1:28:55 AM CEST	derivovat	
April 15, 2008 1:29:51 AM CEST	derivovat (1)	
April 15, 2008 1:30:52 AM CEST	vyber desetinnych mist	
April 15, 2008 1:31:36 AM CEST	about	
April 15, 2008 1:32:34 AM CEST	spatny format vyrazu	
April 15, 2008 1:37:42 AM CEST	spatny format promenne	

created on April 15, 2008 1:54:14 AM CEST with CodeCover

Zadali jsme základní uživatelské scénáře, jako kliknout na button expandovat, na button zjednodušit, derivovat, zadali jinou proměnnou, změnili počet desetinných míst, zapsali výraz ve špatném formátu. Výsledek je možné vidět na tomto shrnujícím obrázku. Bohužel se nám s programem nepodařilo zaznamenávat hned od začátku a tak se iniciační části tříd jako konstruktory a jimi volané metody jeví jako nevyužité. Na následujícím obrázku jsou zeleně části, které se během testu volali a červeně části, které ne.

TreeOperations.java

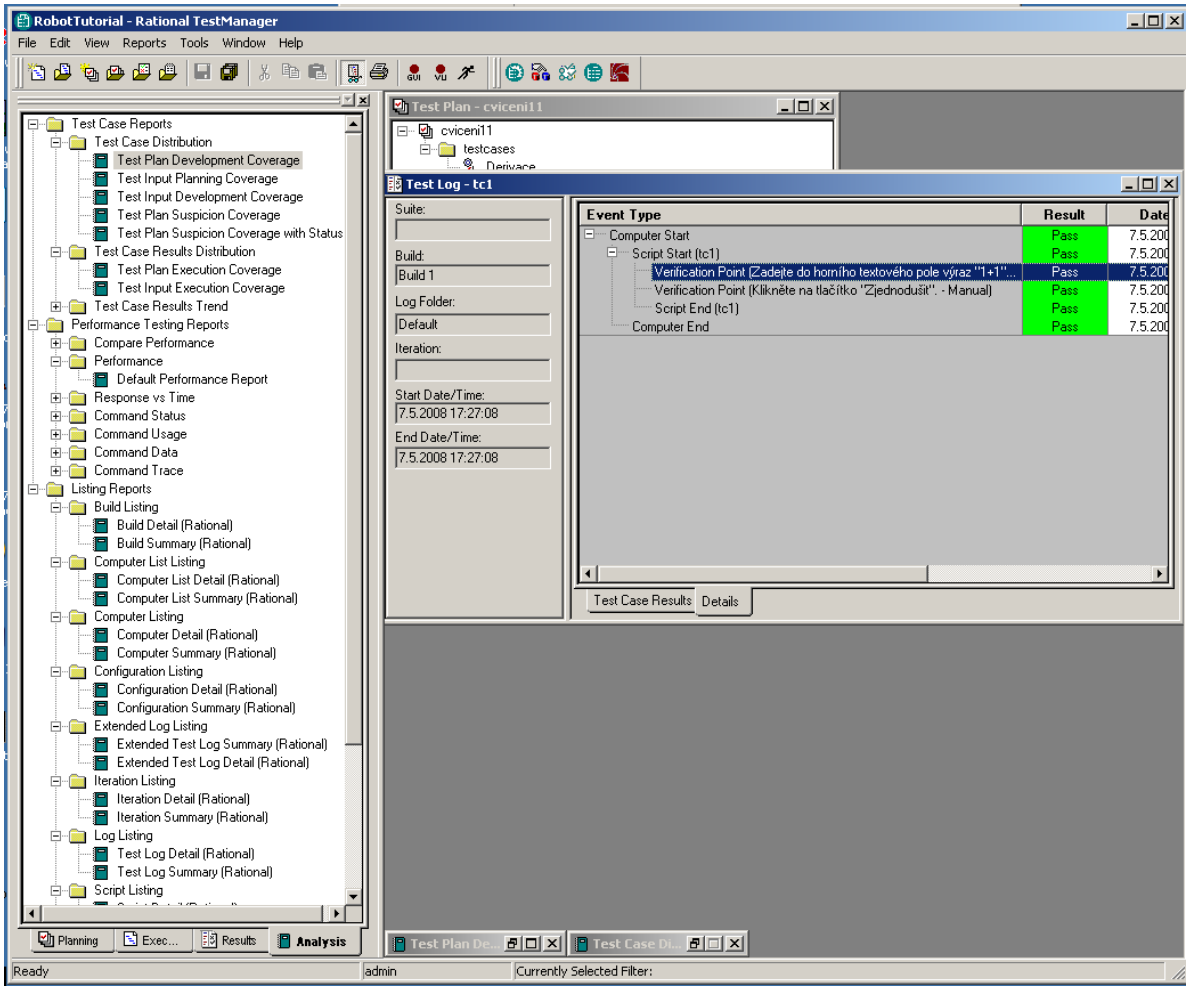
```
35      private static void replace_impl(Evaluable f, Variable var, Evaluable replaceBy){
36          if (f instanceof Operator){
37              Operator o = (Operator)f;
38              if (o.getLeft().equals(var))
39                  o.setLeft(replaceBy);
40              else
41                  replace_impl(o.getLeft(), var, replaceBy);
42              if (o.getRight().equals(var))
43                  o.setRight(replaceBy);
44              else
45                  replace_impl(o.getRight(), var, replaceBy);
46          }
47          else if (f instanceof OneVariableFunction){
48              OneVariableFunction ovf = (OneVariableFunction)f;
49              if (ovf.getArgument().equals(var))
50                  ovf.setArgument(replaceBy);
51              else
52                  replace_impl(ovf.getArgument(), var, replaceBy);
53          }
54          else if (f instanceof Function){
55              Function sf = (Function)f;
56              for (int i = 0; i < sf.argumentCount(); i++){
57                  if (sf.getArgument(i).equals(var))
58                      sf.setArgument(replaceBy, i);
59                  else
60                      replace_impl(sf.getArgument(i), var, replaceBy);
61              }
62          }
63      }
```

Podle mého názoru je tento postup použitelný za jistých situací, spíše jako doplněk k jednotkovému testování a debugování. Pro úplný výčet částí kódu, které se nevolají využijeme některého stroje pro generování testovacích scénářů, např. Ration Robot.

7. Manuální testování pomocí programu Rational Manual Test

S pomocí programu Rational Manual Test jsme provedli manuální testování programu Kalkulačka. Kromě základních funkcí spočívajících ve vytváření testovacího plánu a zaškrtávacích formulářů pro zaznamenávání výsledků testů poskytuje program Rational Manual Test ve spojení s programem Rational Test Manager možnost vytvářet souhrnné zprávy o výsledcích testů – například o počtu pokrytých test cases apod.

Výsledky jednoho ze tří manuálních testů (konkrétně TC1), které jsme prováděli na programu Kalkulačka, jsou zobrazeny na následujícím obrázku. Jak je patrné z tohoto obrázku, tester, jímž jsme v tomto případě byli my, provedl úspěšně všechny požadované kroky, což je indikováno zeleně podbarvenými nápisy „Pass“.

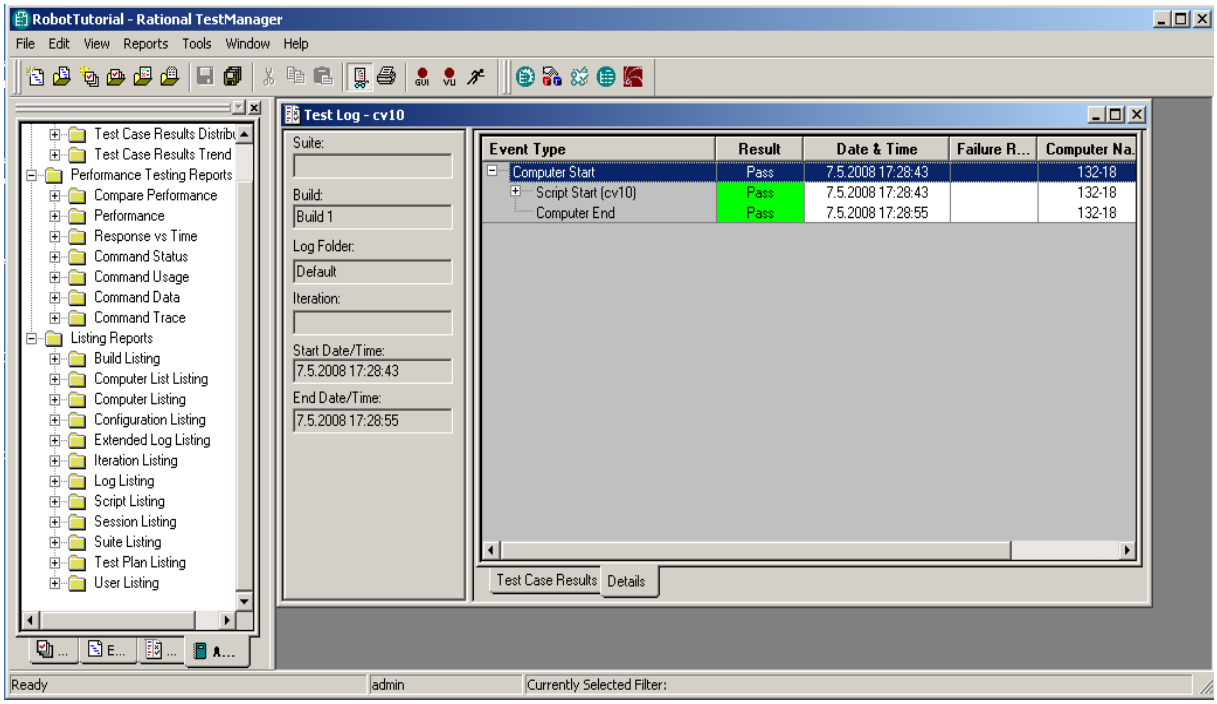


Obrázek 5: Výsledek manuálních testů v programu Rational Manual Test

8. Automatické testování pomocí programu Rational Robot

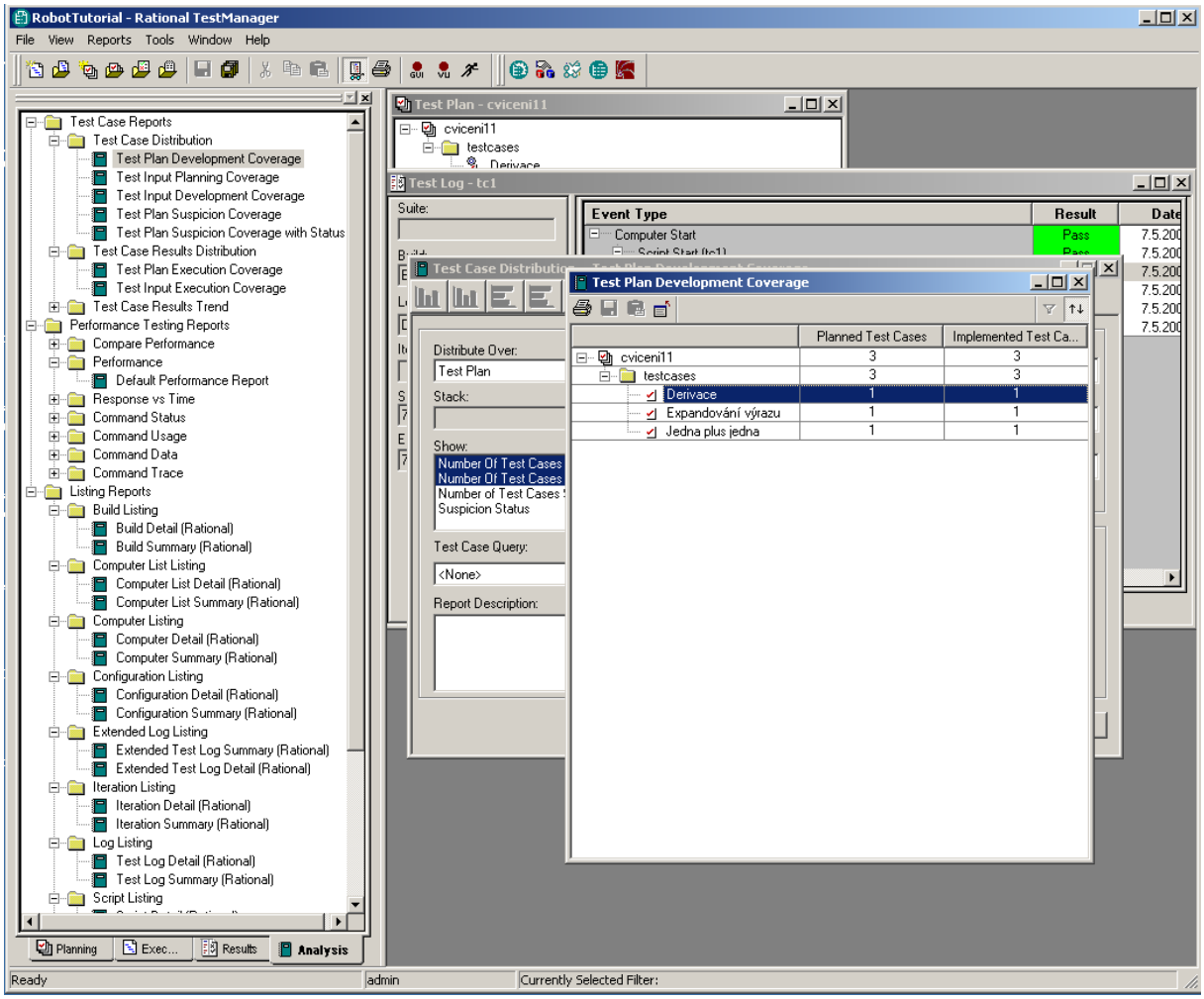
Na rozdíl od programu použitého v předchozí sekci umožňuje program Rational Robot automatické testování aplikací s grafickým uživatelským rozhraním. Rational Robot umožňuje automatické provádění posloupností uživatelských akcí a testování stavu komponent uživatelského rozhraní (textových polí atd.). Používá se především pro regresní testování, jímž se zjišťuje, jestli změny provedené v kódu aplikace nezpůsobily chyby v již otestovaných částech aplikace.

My jsme program Rational Robot použili pro nahrání sekvencí uživatelských akcí pro test cases TC1, TC2 a TC3. Vzhledem k tomu, že naše testovaná aplikace byla napsána v Javě 1.6, se vyskytly problémy, které způsobily nemožnost použít některé testy založené na čtení textu z textových polí aplikace. Tento problém jsme vyřešili pomocí testování obsahu systémové schránky, do níž byl tento obsah vložen rovněž pomocí programu Rational Robot. Obrázek 6 zobrazuje výsledek testu pokrývajícího test case TC 1 provedený na programu Kalkulačka. Obrázek 7 zobrazuje v programu Rational Test Manager souhrnné informace o pokrytí jednotlivých test cases.



Obrázek 6: Výsledek testu v programu Rational Robot

Kromě módu nahrávání akcí umožňuje program Rational Robot vytvářet testy ve skriptovacím jazyku podobném Visual Basicu, jenž jsme nicméně nevyužili.



Obrázek 7: Statistiky testů v programu Rational Test Manager

9. Propagace kovarianční matice

1 Propagace kovariancni matice

$$\mathbf{Z} = \mathbf{X} \cdot \mathbf{Y}, \text{ kde } \mathbf{Z} = \mathbf{F}(\mathbf{X}, \mathbf{Y})$$

$$\mathbf{Y} = \mathbf{Y_0} + \frac{\partial \mathbf{F}}{\partial \mathbf{X}}|_{\mathbf{X_0}}(\mathbf{X} - \mathbf{X_0}) + \mathbf{R_{Y0}} \Rightarrow_{\mathbf{sub}} |\sigma\mathbf{X} = \mathbf{X} - \mathbf{X_0}, \sigma\mathbf{Y} = \mathbf{Y} - \mathbf{Y_0}, \mathbf{J_0} = \frac{\partial \mathbf{F}}{\partial \mathbf{X}}|_{\mathbf{X_0}}|$$

$$\sigma\mathbf{Y} = \mathbf{J_0} \cdot \sigma\mathbf{X} + \mathbf{R_{Y0}}$$

$$\sigma\mathbf{Y} \cdot \sigma\mathbf{Y}^{\mathbf{T}} = (\mathbf{J_0} \cdot \sigma\mathbf{X} + \mathbf{R_{Y0}}) \cdot (\mathbf{J_0} \cdot \sigma\mathbf{X} + \mathbf{R_{Y0}})^{\mathbf{T}} = \mathbf{J_0} \cdot \sigma\mathbf{X} \cdot \sigma\mathbf{X}^{\mathbf{T}} \cdot {\mathbf{J_0}}^{\mathbf{T}} + \mathbf{R_Y Y_0}$$

$$\sum_{\mathbf{YY_0}} = \frac{1}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} (\sigma\mathbf{Y_i} \cdot \sigma{\mathbf{Y_i}}^{\mathbf{T}}) = \frac{1}{\mathbf{n}} \sum_{\mathbf{i=0}}^{\mathbf{n}} (\mathbf{J_0} \sigma\mathbf{X_i} \sigma{\mathbf{X_i}}^{\mathbf{T}} {\mathbf{J_0}}^{\mathbf{T}}) =$$

$$= \mathbf{J_0} \sum_{\mathbf{X}} \mathbf{X_0} {\mathbf{J_0}}^{\mathbf{T}} = \underline{\underline{\sum_{\mathbf{Y}} = \mathbf{J_0} \sum_{\mathbf{X}} {\mathbf{J_0}}^{\mathbf{T}}}}$$

1.2 Implicitni propagace

$$\mathbf{F(X,Y)} = \mathbf{(XY-Z)^2}$$
$$\mathbf{g} = \frac{\partial \mathbf{F}}{\partial \mathbf{Z}} = \mathbf{-2(XY-Z)} \Rightarrow \frac{\partial \mathbf{g}}{\partial \mathbf{X}} = \mathbf{-2Y}, \frac{\partial \mathbf{g}}{\partial \mathbf{Y}} = \mathbf{-2X}, \frac{\partial \mathbf{g}}{\partial \mathbf{Z}} = \mathbf{2}$$
$$\sum_{\mathbf{qq}} = |\sigma_{\mathbf{Z}^2}| = |\mathbf{2}|^{-1} \cdot |\mathbf{-2Y-2X}| \cdot \begin{vmatrix} {\sigma_X}^2 & \sigma_{XY} \\ \sigma_{XY} & {\sigma_Y}^2 \end{vmatrix} \cdot \begin{vmatrix} -2Y \\ -2X \end{vmatrix} \cdot |\mathbf{2}|^{-1} =$$
$$= |\mathbf{-Y-X}| \cdot |\cdots| \cdot \begin{vmatrix} -Y \\ -X \end{vmatrix} = |\mathbf{-Y\sigma_{X^2} - X\sigma_{XY}}, \mathbf{-y\sigma_{XY} - X\sigma_{Y^2}}| \cdot \begin{vmatrix} -Y \\ -X \end{vmatrix} =$$
$$= \mathbf{Y^2\sigma_{X^2}} + \underbrace{2XY\sigma_{XY}}_{\doteq \mathbf{0}} + \mathbf{X^2\sigma_{Y^2}} = \underline{\underline{\mathbf{{Y_0}^2\sigma_{X^2} + {X_0}^2\sigma_{Y^2}}}}$$

1.1 Explicitni propagace

$$\mathbf{Z} = \mathbf{F(X,Y)} = \mathbf{X \cdot Y}$$
$$\mathbf{J} = \left|\frac{\partial \mathbf{Z}}{\partial \mathbf{X}} \frac{\partial \mathbf{Z}}{\partial \mathbf{Y}}\right| = |\mathbf{YX}|$$
$$\sum_{\mathbf{ZZ}} = |\sigma_{\mathbf{Z}^2}| = |\mathbf{YX}| \cdot \begin{vmatrix} {\sigma_X}^2 & \sigma_{XY} \\ \sigma_{XY} & {\sigma_Y}^2 \end{vmatrix} \cdot \begin{vmatrix} Y \\ X \end{vmatrix} =$$
$$= |\mathbf{Y\sigma_{X^2} + X\sigma_{XY}}, \mathbf{Y\sigma_{XY} + X\sigma_{Y^2}}| \cdot \begin{vmatrix} Y \\ X \end{vmatrix} = (\mathbf{Y^2\sigma_{X^2} + XY\sigma_{XY}}) + (\mathbf{XY\sigma_{XY} + X^2\sigma_{Y^2}}) =$$
$$= \mathbf{{Y_0}^2\sigma_{X^2}} + \underbrace{2XY\sigma_{XY}}_{\doteq \mathbf{0}} + \mathbf{{X_0}^2\sigma_{Y^2}} = \underline{\underline{\mathbf{{Y_0}^2\sigma_{X^2} + {X_0}^2\sigma_{Y^2}}}}$$

10. Závěr

V rámci předmětu X33TSW jsme si vyzkoušeli práci s moderními nástroji pro testování softwaru. V programu Rational Rose jsme vytvořili systémové požadavky a případy užití pro program Kalkulačka. V programu Enterprise Architect jsme využili funkci reverzního inženýrství pro vytvoření UML class diagramů popisujících strukturu tříd testovaného programu. Dále jsme vytvořili diagram příčin a následků snažící se postihnout příčiny nedostatečné rychlosti testovaného programu. Pomocí programů Netbeans Profiler jsme analyzovali možnosti zvýšení rychlosti programu a snížení jeho paměťové náročnosti. Naši aplikaci jsme analyzovali pomocí programu Code Cover. Dále jsme použili programy Rational Manual Tester a Rational Robot pro manuální a automatické testování uživatelského rozhraní. Nakonec jsme na uměle vytvořeném problému odvodili propagaci variance.