ALGator

Avtor: Tomaž Dobravec (2013, 2014, 2105)

Verzija: 0.21

1 Opis sistema 3

1.1 Namen 3

1.2 Vloge v sistemu 3

1.2.1 Skrbnik sistema 3

1.2.2 Skrbnik projekta 3

1.2.3 Raziskovalec 4

1.2.4 Gost 4

1.3 Primer uporabe 4

2 Delovanje sistema 5

2.1 ALGator kot samostojna aplikacija 5

2.2 ALGator kot spletna aplikacija 5

2.2.1 AEE - Algator execution engine 5

2.2.2 ADE - Algator data engine 6

2.2.3 AWE - Algator Web Engine 7

3 The configuration of the system 8

3.1 The structure of the <algator\_root> folder 8

3.2 System configuration 9

3.2.1 Building blocks 9

3.2.2 Projects 12

3.2.3 Algorithms 13

3.2.4 Tests 14

3.2.5 Results 16

3.2.6 Queries 18

3.2.7 Graphs 23

3.2.8 Configuration of the ALGator system 24

4 Izvajanje algoritmov 27

4.1 Osnovni pojmi 27

4.2 Naloge (TestCases) in množice nalog (TestSets) 28

4.3 Algoritmi 31

4.4 Rešitev naloge in zapis rezultatov 33

4.4.1 Privzeti izhodni parametri 33

4.5 Meritve 34

4.5.1 Meritve parametrov izvajanja 34

4.5.2 Meritve s števci 35

4.5.3 Meritve s prirejenim JVM 36

4.6 Nov projekt 36

5 Izvajanje algoritmov v algator.Execute 37

# Opis sistema

ALGator je sistem za izvajanje algoritmov na podanih testnih podatkih ter analizo rezultatov izvajanja.

Sistem omogoča dodajanje in upravljanje s poljubnim številom projektov. V okviru enega projekta je definiran problem, testne množice vhodnih podatkov ter način reševanja nalog tega problema. Projekt lahko vsebuje poljubno število algoritmov, ki naloge rešujejo na predpisan način. Sistem omogoča analizo izvajanja posameznega algoritma ter primerjavo med algoritmi istega projekta.

## Namen

* Reševanje definiranih problemov z različnimi algoritmi.
* Analiza delovanja posameznega algoritma pri reševanju nalog danega problema.
* Primerjava različnih algoritmov za reševanje istega problema.

## Vloge v sistemu

Sistem lahko uporabljajo uporabniki z različnimi vlogami in pravicami. Neprijavljen uporabnik (gost) lahko izvaja le funkcije, ki ne spreminjajo podatkov sistema (podatke lahko le pregleduje). Za izvajanje ostalih funkcij mora biti uporabnik v sistem prijavljen. Natančni pomen posamezne vloge je opisan v nadaljevanju.

### Skrbnik sistema

Skrbnik sistema postavi celotni sistem in skrbi za strojno in programsko opremo in ima vse pravice in dostop do vseh virov sistema.

### Skrbnik projekta

Skrbnik projekta definira en projekt. K definiciji projekta spada

* definicija problema,
* definicija testnih množic vhodnih podatkov,
* opis formata vhodnih in izhodnih podatkov,
* definicija vseh javanskih razredov, ki so potrebni za izvajanje algoritmov tega projekta.

Skrbnik projekta ima dostop do vseh virov projekta. Če je projekt javen (ta atribut nastavi skrbnik projekta), lahko podatke o projektu vidijo vsi uporabniki. Privatne projeket vidi samo skrbnik sistema in skrbnik projekta.

### Raziskovalec

Raziskovalec definira en algoritem znotraj izbranega projekta. Javne algoritme (ta atribut nastavi raziskovalec) lahko vidijo vsi uporabniki, privatnega pa poleg skrbnikov le raziskovalec.

### Gost

Gost lahko vidi podatke o javnih algoritmih javnih projektov.

## Primer uporabe

V tem razdelku je opisan najpogostejši način uporabe sistema ALGator.

* Skrbnik sistema postavi celotni sistem in objavi povezavo do spletne strani sistema.
* Skrbnik projekta doda nov projekt; s tem definira problem in nabore testnih primerov ter predpiše način reševanja tega problema. Za vsak definiran projekt sistem avtomatsko ustvari spletno stran z razlago problema in navodili za pripravo algoritma.
* Skrbnik projekta v projekt doda nekaj znanih algoritmov za reševanje tega problema, ki bodo služili za osnovno primerjavo z novimi algoritmi, ki jih bodo dodajali raziskovalci.
* Raziskovalec skladno z navodili, objavljenimi na spletni strani projekta, pripravi nov algoritem in ga doda v sistem. ALGator dodani algoritem avtomatsko požene na testnih podatkih. Raziskovalec nato primerja učinkovitost tega algoritma z učinkovitostjo algoritmov, ki so že v projektu. Če želi, lahko algoritem odpre za javnost (algoritem postane javen) ali pa ga skrije (privaten algoritem). Privatne algoritme lahko raziskovalec kadarkoli naredi za javne.
* Gost sistema lahko pregleduje podatke o vseh javnih projektih in javnih algoritmih. Lahko izvaja primerjave in ostale operacije, ki ne spreminjajo konfiguracije sistema. Če želi, se lahko kadarkoli prijavi kot raziskovalec ali kot skrbnik projekta.

# Delovanje sistema

ALGator lahko uporabljamo na dva načina: kot samostojno ali kot spletno aplikacijo. Samostojna aplikacija je namenjena razvijalcem, ki želijo ALGator uporabljati na svojem računalniku, ločeno od ostalih uporabnikov. Spletna aplikacija omogoča uporabo sistema več uporabnikom hkrati. Rezultati izvajanja spletne aplikacije so vidni vsem uporabnikom.

## ALGator kot samostojna aplikacija

Za delovanje ALGatorja kot samostojne aplikacije potrebujemo naslednje:

* Izvajalni dela sistema (ALGator.jar + folder lib)
* Mapo s podatki o projektih (data\_root folder)
* Pravilno nastavljene sistemske spremenljivke
  + CLASSPATH (kazati mora na ALGator.jar)
  + ALGATOR\_ROOT (kazati mora na folder, kjer je ALGator.jar)
  + ALGATOR\_DATA\_ROOT (kazati mora na data\_root folder)

Programa ALGator se nahaja na https://github.com/ALGatorDevel/Algator. Tam je v datoteki README podrobneje opisan postopek namestitve ALGatorja.

## ALGator kot spletna aplikacija

Za pravilno delovanje ALGator spletne aplikacije potrebujemo tri podsisteme: **AEE** (ALGator execution engine), **ADE** (ALGator data engine) in **AWE** (ALGator Web engine). Na posameznem fizičnem računalniku lahko teče podsistemov (na primer: en fizični računalnik je lahko AEE ADE in AWE hkrati). Spletna aplikacija vsebuje natanko en ADE in en AWE ter enega ali več AEE.

### AEE - Algator execution engine

Namen: to je računalnik, na katerem se izvajajo testi (izvajalni stroj). Na tem računalniku teče **TaskClient**.

AEE konfiguracija (v local\_config):

- ComputerID ... enolična oznaka računalnika

- TaskServerName ... IP naslov ali ime ADE računalnika

- VMEP in VMEPClassPath, če na tem AEE lahko izvajamo jvm teste

Kaj mora biti nameščeno

* sistemski spremenljivki $ALGATOR\_ROOT in $ALGATOR\_DATA\_ROOT
* ALGator.jar + povezava do tega jara v CLASSPATHu
* Pravilno konfiguriran sistem, da se TaskClient požene ob rebootu

TaskClient

* TaskClient v zanki sprašuje TaskServer za nove naloge; ko dobi nalogo, požene algator.Execute, ta pa rezultate shranjuje direktno v data\_root na ADE; ko se algator.Execute konča, se zanka nadaljuje (glej ... <http://commons.apache.org/proper/commons-exec/tutorial.html>)

**Komunikacija med AEE in ADE preko datoteke v <tasks> folderju**

V datoteki <tasks>/status/idt so zapisani podatki oopravilu idt (idt je enolična oznaka opravila, ki jo določi TastServer) in sicer:

* v prvi vrstici je datum, ki je bilo opravilo ustvarjeno (ta podatek zapiše TastServer ob akciji addTask)
* v drugi vrstici je status in sicer
  + created (opravilo je bilo ustvarjeno in čaka na AEE; ta podatek zapiše TastServer ob akciji addTask)
  + scheduled (opravilo je bilo predano v izvajanje; ta podatek zapiše TaskServer ob akciji getNextTask)
  + processing x% (opravilo se izvaja in je končano do x%; ta podatek piše notifikator ob akciji notify)
  + completed (opravilo je končano; ta podatek zapiše TastServer ob akciji completeTask)
* v tretji vrstici je datum, ki je bilo opravilo dokončano (ta podatek zapiše TastServer ob akciji completeTask).

### ADE - Algator data engine

Namen: na tem računalniku so shranjeni vsi podatki (parametri in rezultati projektov ter konfiguracija sistema), na njem teče tudi **TaskServer**, ki upravlja z opravili.

ADE konfiguracija (v global\_config na <data\_root> folderju):

* podatki o družinah računalnikov in o računalnikih

Kaj mora biti nameščeno:

* Sistemska spremenljivka $ALGATOR\_DATA\_ROOT
* ALGator.jar + povezava do tega jara v CLASSPATHu
* pravilno konfiguriran sistem, da se TaskServer požene ob rebootu

TaskServer

* na ADE teče server (TaskServer) ki sprejema zahteve in vrača odgovore
* server vodi interno tabelo čakajočih opravil, iz katere po potrebi jemlje oziroma v katero po potrebi dodaja nova opravila
* ob vsaki spremembi tabele, se njena vsebina zapiše na disk (če bi se server ali računalnik sesul, čakajoča opravila tako ne bi bila izgubljena)
* ob zagonu serverja se prebere tabela opravil
* za shranjevanje podatkov in za komunikacijo z zunanjim svetom (npr. za sprejemanje sporočil od TaskClient-a) TastServer uporablja folder <comm\_folder>=<data\_root>/comm

- Seznam zahtev, na katere odgovarja server:

**addTask**

* parametri: project, algorithm, testset, measurementtype
* odgovor: idt
* opis: preveri, če to opravilo že obstaja; če ne obstaja, ga ustvari, mu določi zaporedno številko (idt) in ga uvrsti v čakalno vrsto; v vsakem primeru vrne idt opravila

**getNextTask**

* parametri: idc
* odgovor: idt, project, algorithm, testset, measurementtype
* opis: pregleda opravila in vrne "prvega", ki se lahko izvrši na računalniku idc; hkrati si server zabeleži, komu je dal kaj v izvajanje in kdaj;pri pregledovanju taskov preveri tudi, koliko časa se nek task že izvaja; če ima nek task predolgo status "In progress", ga da v izvajanje drugemu računalniku
* tehnično: task se prestavi iz seznama čakajočih\_opravil na seznam opravil\_v\_izvajanju; pri tem se opravilu pripne dodatne parametre (čas izvajanja, AEE, ki ga je prevzel, ...).

**getTaskStatus**

* parameter: idt **ALI** proj, alg, testset, mtype
* odgovor: status opravila (npr. Scheduled, In progress (xx%), ...)
* zahtevo pošlje, na primer, AWE, ko želi uporbnika obvestiti o statusu

**taskCompleted**

* parameter: idt, status (OK ali NOK + sporocilo)
* odgovor: OK
* opis: označi opravilo kot dokončano in ga odstrani iz seznama opravil\_v\_izvajanju;
* to zahtevo kliče AEE, ko zaključi z izvajanjem opravila

**getTaskList**

* parameters: void
* odgovor: seznam čakajočih opravil

### AWE - Algator Web Engine

Namen:

- na tem računalniku teče web page in vsi servisi, ki jih ta stran omogoča

Parametri:

- potrebujem $ALGATOR\_ROOT

# The configuration of the system

Vsi podatki, s katerimi upravlja sistem ALGator so zapisani v mapi <algator\_root> in njenih podmapah v tekstovnih konfiguracijskih datotekah tipa JSON ali CSV. Tip posamezne datoteke je razviden iz opisa datoteke v tem poglavju.

Sistem ALGator uporabi vrednosti sistemske spremenljivke $ALGATOR\_ROOT in predpostavi, da kaže na <algator\_root>. Če ta spremenljivka ni nastavljena, ALGator v ta namen uporabi trenutni direktorij. Podatki o projektih se nahajajo na $ALGATOR\_DATA\_ROOT. Če vrednost te sistemske spremenljivke ni nastavljena, se uporabi $ALGATOR\_ROOT/data\_root. Obe vrednosti lahko uporabnik nastavi tudi s pomočjo stikal –r in –d.

## The structure of the <algator\_root> folder

<algator\_root>

local\_config // configuration folder of this computer

config.atlc

data\_root // folder with information about projects

global\_config // configuration folder of the system

config.atgc

families.atcf // info. about families of computers

computers.atcc // registered computers

projects // root for the projects

tasks // folder for comm. between ADE and AEEs

app // folder with programs and services

ALGator // folder with ALGator.jar, ...

* in the <local\_config> folder contains the configuration of the current computer (i.e. computer ID, path for VMEP virtual machine, ....). See section 2.2.10 for datails.
* the <data\_root> folder contains informatin about the ALGator system. If a computer is a part of a group of computers composing ALGator system, this folder should be set as a link to this system's data\_root folder (that is mounted, for example, using the samba protocol).
* in the <global\_config> folder containes inforamtion about the ALGator system. See section 2.2.9 for datails.
* In the <tasks> folder serves as a comunication chanel between AED and AEEs of the system.

The <projects> folder contains the configuration information for all the projects that are defined in the system. For every project ProjName it contains a folder named PROJ-ProjName with the following structure:

PROJ-ProjName // project root folder

proj // folder for all project files

ProjName.atp // project configuration file

ProjName.atrd // result description file

src // folder for project source files

[Project]AbsAlgorithm.java

[Project]TestSetIterator.java

[Project]TestCase.java

bin // compiled classes of the project

[Project]AbsAlgorithm.class

[Project]TestSetIterator.class

[Project]TestCase.class

doc // html documentation for project

algs // folder for algorithms

ALG-AlgName // algorithm root folder (\*1)

AlgName.atal // algorithm configuration file

src // algorithm source files

[Algorithm][Project]AbsAlgorithm.java

bin // compiled algorithm classes

[Algorithm][Project]AbsAlgorithm.class

doc // html documentation for algorithm

reports // folder for algorithms reports

tests // project-specific test files folder (\*2)

TestSetName1.atts // one or more testset files

doc // html documentation for testsets

results // folder with results (attr files)

queries // folder for predefined queries

reports // folder for projects reports

\*1 - for each algotihm AlgName there is one folder with this structure

\*2 - files in this folder can be organized in sub-folders. The structure of this

folder is specific to the project and it is not defined on the ATSystem

level. Only the curent TestSetIterator knows for the structure of

this folder.

## System configuration

All the entities of the ALGator are defined by configuration files that are placed within the <algator\_root> folder. The type, location and the structure of these files is described in the following.

### Building blocks

#### NameAndAbrev

A String that identifies the name of an entity and its abreviation. A reserved word AS is used as delimiter: Example:

"BubbleSort AS BS".

If the second part of NameAndAbrev is missing (i.e. "BubbleSort"), the value is interpreted as "Name AS Name".

#### StatFunc

A statistical function used to calculate one scalar from a list of values. The functions are one of the following:

MIN the minimal element in the list

MAX the maximal element in the list

SUM the sum of all elements in the list

AVG the average of the elements in the list

VAR the variance of the elements in the list

FIRST the first element in the list

LAST the last element in the list

#### Sorting functions

To sort data, one of the following methods can be used:

* + numerical sort, ascending
* - numerical sort, descending
* > lexicogvvdraphic sort, ascending
* < lexicographic sort, descending

#### ParameterType

The parameters used in the program for several purposes (i.e. result parameters, testcase parameters, ...) are of the following types:

* timer a timer for one stage of algorithm execution (int value)
* counter counter of operations (int value); subtype defines StatFunc

to be used (see below)

* int integer value
* double double value; for decimal separator “.” is used;

subtype defines a number of decimal places to be used

* string a string inside quotes “”

Each type can have a subtype. A meaning of subtype is type-dependant. For timers a subtype defines a StatFunc to be used

|  |  |  |
| --- | --- | --- |
| Type | Subtype | Description |
| timer | # StatFunc | # ... the number of a timer (0 is default timer used to measure overall time; other timers may be activated by a algorithm-designer using the @SET\_PHASE(#) directives) or by project administrator by calling timer.next() method.    StatFunc ... a statistical function used to calculate the value of this timer.  Default: 0 MIN |

#### Parameter

A description for one parameter.

**File type**: JSON

**EntityID:** Parameter

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Name | String | Name of the parameter |
| Description | String | Description of the parameter |
| Type | ParameterType | A type of a parameter |
| Subtype | String | Optional. A space-delimited list of parameter-type specific arguments. This arguments are used by Executor to distinguish, for example, different timers. |
| Value | String | Optional. A value of a parameter (if known). |

**Example:**

{

"Name" : "T1",

"Description" : "Preprocessing timer",

"Type" : "timer",

"Subtype" : "1 AVG"

}

### Projects

Each project ProjName is defined in the folder named PROJ-ProjName within the projects folder.

Shortcut: <project\_root> = <data\_root>/projects/PROJ-ProjName

#### Project

The configuration of an individual project ProjName is defined in atp file as follows.

**File type**: JSON

**EntityID:** Project

**File extension**: atp

**File location**: <project\_root>/proj/ProjName.atp

**Fields:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | | Type | | Description |
| Description | | String | | HTML text to be shown on project website |
| HTMLDescFile | | Filename | | HTML file with detailed desc. of this project; filename is relative to the project folder |
| Author | | String | | Author of the project |
| Date | | String | | The date of the last change |
| TestSetDescHTML | | Filename | | HTML file with detailed **non-technical** desc. of the testsets used in the tests. |
| TestSetTechDescHTML | | Filename | | HTML file with detailed **technical** desc. of the testsets used in the tests. |
| TestSets | | String[] | | a list of TestSets |
| AlgDescHTML | | Filename | | HTML file with detailed **non-technical** desc. of the algorithms of this project, i.e., what is the input and what is the output of the algorithms |
| AlgTechDescHTML | | Filename | | HTML file with detailed **technical** desc. of the algorithms of this project. This file contains information for the authors of new algorithms: how the java code of the algorithm should look. |
| Algorithms | | String[] | | a list of Algorithms |
| ProjectJARs | Filename[] | | An array of jars that are available when compiling the project; names are relative to the folder <project\_root>/proj/lib | |
| AlgorithmJARs | Filename[] | | An array of jars that are available when compiling an algorithm; names are relative to the folder <project\_root>/proj/lib. | |
| EMExecFamily | String | | Identification of the family of computers that can execute em measurements for algorithms of this problem. | |
| CNTExecFamily | String | | Identification of the family of computers that can execute cnt measurements for algorithms of this problem. | |
| JVMExecFamily | String | | Identification of the family of computers that can execute jvm measurements for algorithms of this problem. | |
| ProjectReports | String [] | | An array of reports to be displayed on the project results description page. | |
| AlgorithmReports | String [] | | An array of reports to be displayed on the algorithm results description page. | |

Note1: project description implicitly includes three source files for each language (3 for java, 3 for C, ...) named <project\_name>AbsAlgorithm.<ext>, <project\_name>TestCase.<ext> and <project\_name>TestSetIterator.<ext> (where <ext> = java or cpp). These files should be placed in the src subfolder of a project.

Note2: AlgorithJARs are provided by project administrator. An author of the algorithm CAN NOT attach a JAR file to his algorithm (all sources must be evident to the public) .

Example:

{

"Project" : {

"Description" : "Testing several sorting methods",

"HTMLDescFile" : "sort.hmtl",

"Author" : "Tomaz",

"Date" : "30/07/2013",

"Algorithms" : ["QuickSort", "BubbleSort"],

"TestSets" : ["TestSet1", "TestSet2"]

}

}

### Algorithms

All the data of the algorithm <algorithm\_name>is placed in the folder algs/ALG-<algorithm\_name> inside the <project\_root> folder.

Shortcut: <algorithm\_root> = <project\_root>/algs/ALG-<algorithm\_name>

The <algorithm\_root> folder contains:

* AlgName.atal file with the description of the algorithm
* src folder with all source files. The name of main algorithm's class should be <algorithm\_name>Algorithm.<ext> (where <ext> = java or cpp). Note: if algorithm requires more than one source file, names of these files have to be listed in Classes property (names are relative to algorithms's src foleder). The main algorithm class MUST be in <algorithm\_root>/src folder (not in a subfolder!).

#### Algorithm

The configuration of an individual algorithm <algorithm\_name> is defined in atal file as follows.

**File type**: JSON

**EntityID:** Algorithm

**File extension**: atal

**File location**: <algorithm\_root>/AlgName.atal

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| ShortName | String | a project unique short name (at most 8 chars) |
| Description | String | HTML text to be shown on algorithm website |
| HTMLDescFile | Filename | HTML file with detailed description of this algorithm; filename is relative to the algorithm folder |
| Author | String | Author of the algorithm |
| Date | String | Creating date of the algorithm |
| Classes | Filename[] | an array of source files (other than <algorithm\_name>Algorithm.java |
| Language | String | Valid values: JAVA (default) or C |
| ExecuteSignature | String | Signiture for execute function of the algorithm. This property is used only for C++ algorithms. Example:    int \*tab, int size |

Example: QuickSort1.atal

{

"Algorithm" : {

"ShortName" : "BS",

"Description" : "The classical bubble sort algorithm",

"HTMLDescFile" : "bubblesort.hmtl",

"Author" : "BS and Tomaz",

"Date" : "07/30/2013"

}

}

### Tests

#### TestSet

A description of a collection of tests.

**File type**: JSON

**EntityID:** TestSet

**File extension**: atts

**File location**: <tests\_root> = <project\_root>/tests

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | | Description |
| ShortName | String | | A shortname of this test set. This description will be used in the result presentation forms. |
| Description | String | | A detailed description of this test set |
| HTMLDescFile | Filename | | HTML file with detailed description of this project; the filename is relative to the test folder |
| N | Integer | | The number of tests in this test set. When performing this test set, Executor must produce N outputs. This parameter is used to check if a tesset has already been completed (i.e. if the output file contains N results). |
| TestRepeat | | Integer | Number of times to execute a single test from this test set. |
| TimeLimit | | Integer | Maximum time allowed (in seconds) for execution of one **test**; if the algorithm does not finish in this time, the execution is killed. Default value: 10. |
| QuickTest | | Boolean | If true, the testset can be executed on AM, otherwise it must be executed on EM |
| TestSetFiles | | Filename[] | An array of masks of files (in <tests\_root> folder) that belong to this test set. The first file in this array is a project-specific file that contains detailed information of this test set. This file is an input to the <Project>TestSetIterator class. |
| DescriptionFile | | Filename | A name of the file with the project dependant description of this testset (in this file typically one line describes one test) |

**Example:**

"TestSet": {

"ShortName" : "SortQ",

"Description" : "This testset contains simple quick ...",

"HTMLDescFile" : "testset1.html",

"N" : 13,

"TestRepeat" : 10,

"QuickTest" : true,

"TestSetFiles" : ["quick\*"],

"DescriptionFile" : "testset1.txt"

}

### Results

#### ResultDescription

A description of a format of a result, containing a description of all parameters that are present in a single result.

**File type**: JSON

**EntityID:** ResultDescription

**File extension**: atrd

**File location**: <project\_root>/proj

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Format | String | A format of a file with results. Current implementation supports only CSV format, in the future JSON or other formats may be supported |
| Delimiter | String | A delimiter betwen CSV data. This filed is used only in CSV format. |
| TestParameters | String [] | The order of the test parameters. |
| ResultParameters | String [] | The order of the result parameters. |
| Parameters | Parameter[] | An array of parameters that are part of a result. |

The parameters TestParameters and ResultParameters define the order of parameters in CSV format. In the result file the parameters in TestParameters describe the properties of the test (i.e. the number of elements, group name, ...) and are used by analysing system to group results of different algoritmhs. The parameters in ResultParameters present the properties of the algorithm execution on a given test.

A default parameter type is INT. If a parameter is listed in TestParameters or ResultParameters and is not defined in Parameters, a default INT parameter is used.

**Example:**

{

"ResultDescription":

{

"Format" : "CSV",

"Delimiter" : ";",

"TestParameters" : ["Test", "Group", "N"],

"ResultParameters" : ["Tmax", "Check"],

"Parameters": [

{

"Name": "Test",

"Description": "The name of the test",

"Type": "string"

},

{

"Name": "Group",

"Description": "The name of the group",

"Type": "string"

},

{

"Name": "N",

"Description": "The size of the test",

"Type": "int"

},

{

"Name" : "Tmax",

"Description" : "Sorting MAX time",

"Type" : "timer",

"Subtype" : "0 MAX"

},

{

"Name" : "Check",

"Description" : "The correctness of the algorithm (OK/NOK)",

"Type" : "string"

}

]

}

}

#### Result

A result of a test can be presented in different formats. Currently only CSV format is supported (in the future also JSON and other formats may be added).

A result contains the following data

* algorithm … algorithm name
* testset … testset name
* test … unique id of test withi this testset
* status ... DONE if processes finished normally; KILLED if the test

exceeded the given amount of time; FAILED if other error

occured during the execution.

* parameters … parameters of a test

if status == DONE, all the parameters given in corresponding ResultDescription file are listed;

If status != DONE, an error message if given in the fifth column

The parameters are listed in the order that is defined in the ResultDescription file: first the TestParameters followed by the ResultParameters. All test-specific parameters (if they exist) should also be defined in the ResultDescription file. If a value of a parameter for a given test is not known, it is presented by an empty string.

If an error occurs during the execution of a test, result for this test contains the first 4 data (algorithm, testset, test and status), followed by the error description string. All such lines in result file will be omitted during the analyses of the results, but they must be there (for error tracking and because the result file must contain exactly N lines)!

Example of one line in CSV result file:

quicksort;testset1;Test-7;DONE;sorted;10;35;5;7;23.5

### Queries

A query to combine results from several result files into single array of data.

**File type**: JSON

**EntityID:** Query

**File extension**: atqd

**File location**: <project\_root>/queries/QueryName.atqd

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Description | String | Description of a query (HTML text) |
| Algorithms | NameAndAbrev[] | An array of algorithms to be included in this query |
| TestSets | NameAndAbrev[] | An array of testsets to be included in this query |
| TestParameters | NameAndAbrev[] | An array of test parameters |
| ResultParameters | NameAndAbrev[] | An array of result parameters |
| GroupBy | String[] | An array of groupby description |
| Filter | String[] | An array of filters |
| SortBy | String[] | An array of sort parameters |
| Count | String | This query is COUNT query?  »1« - true, other - false |

Query example:

{

"Description" = "Demo query",

"Algorithms" = ["QuickSort AS QS", "BubbleSort AS BS"],

"TestSets" = ["TestSet1 AS TS1", "TestSet2 AS TS2"],

"TestParameters" = ["TestSet", "Test AS T", "N"],

"ResultParameters" = ["TMin", "TMax"],

"GroupBy" = ["N; Tmax:SUM; MIN"],

"Filter" = ["N > 100", "N < 1000"],

"SortBy" = ["N:-"],

"Count" = ["1"]

}

Dodatna pojasnila:

* Polja Algorithms, Testsets, TestParameters, ResultParameters določajo obseg poizvedbe (kateri algoritmi in testne množice dobo vključeni v rezultat) in širino tabele (kateri stolpci bodo nastopali v rezultatu).

Podatki o izhodnih parametrih (ResultParameters) se črpajo iz vseh atrd datotek (glej poglavje 3.5). Podatki o parameterih testa se črpajo iz Project-em.atrd.

Izhodna tabela vsebuje tri privzete stolpce (se jih ne da izključiti): TestSet, TestID in Pass.

Če imamo v poizvedbi N algoritmov in M testnih množic ter T testnih parametrov (input parameters) in R paremetrov rezultata (output parameters), potem bo tabela vsebovala 3 + T + N \* R stolpcev.

* Polji GroupBy in Filter vplivata na število vrstic (lahko jih zmanjšata)

- Operacije se izvajajo v tem vrstnem redu: generate table, filter, group, sort.

* Polje Filter:
  + Filter iz tabele izloči vse vrstice, ki NE zadoščajo pogojem.
  + Filtri se lahko izvaja po vseh parametrih (tudi po izhodnih parametrih). Primer: če je CompRate izhodni parameter, se ob filtru "CompRate < 100" v tabeli ostanejo samo vrstice, pri katerih je vrednost CompRate < 100 pri VSEH algoritmih, ki so vključeni v poizvedbo.
  + Filter je tabela preprostih pogojev, za združevanje se uporabi operator and; z drugimi besedami, po filtru v tabeli ostanejo samo tiste vrstice, ki ustrezajo vsem pogojem filtra
    - Primer: Filter : ["N > 100", "N < 1000"]: med rezultati ostanejo samo tiste vrstice, v katerih je vrednost parametra N večja od 100 in manjša od 1000.
  + Posamezen filter je oblike: <ime\_parametra> <operator> <vrednost>
  + Operatorji filtra:
    - Za številske parametre: <, <=, >, >=, ==, !=
    - Za znakovne paramtere: ==, !=
  + Filter lahko vsebuje en parameter. Primer:

Tavg < $1 @(10,100,10)

Tak filter se bo pognal večkrat (v zgornjem primeru za vrednosti $1=10,20, ..., 100). Ker je rezultat večkratnega poganjanja filtra več tabel, je tak filter smiseln samo v kombinaciji s parametrom COUNT==1 (glej spodaj). Izhodna tabela za vsako vrednost parametra $1 vsebuje eno vrstico, ki pripadajo poizvedbam

COUNT Tavg < 10

COUNT Tavg < 20

...

COUNT Tavg < 100

Obseg (range) parametra $1 je podan s tremi celoštevilskimi parametri:

@(start,stop,step)

Obseg pišemo na koncu kateregakoli pogoja, ki sestavlja filter.

* Polje GroupBy:
  + Vse vrstice z enako vrednostjo podanega parametra se združijo v eno vrstico.

* + GroupBy se lahko izvaja samo po parametrih testa,
  + Če GroupBy vsebuje več opisov, se združevanja naredijo eno za drugim (najprej prvo združevanje; drugo združevanje se naredi na rezultatu prvega, ...)
  + Pred posameznim združevanjem se tabela uredi po polju združevanja; način urejanja je lahko podan ob imenu polja za dvopičjem (primer: "GroupBy" = ["N:>; Tmax:SUM; MIN"]) – glej 2.2.1.3. Če način urejanja ni podan, se uporabi numerično urejanje v naraščajočem vrstnem redu (+).

* + Format zapisa:
    - Primer: "N; Tmax:sum; min"
    - Eden ali več podatkov, ločenih s podpičjem
    - Prvi podatek je ime parametra (in opcijsko način urejanja), po katerem se združuje
    - Ostali podatki opisiujejo, katera statistična funkcija naj se uporabi za izračun vrednosti posameznega stolpca.
      * Podatek tipa polje:StatfFunc podaja statistično funkcijo za to polje
      * Podatek tipa StatfFunc podaja statistično funkcijo za vsa polja, za katera ni eksplicitno določeno drugače
      * Če za katero polje ni podana statistična funkcija, se uporabi funkcija FIRST.

Primeri: Recimo, da ima tabela stoplce N, Tmin in Tmax.

* + - * GroupBy = "N" ... za vse stolpce se uporabi AVG
      * GroupBy = "N; MIN" ... za vse stolpce se uporabi MIN
      * GroupBy = "N; Tmax:MAX; MIN" ... za Tmax se uporabi MAX, za osttale MIN
      * GroupBy = "N; Tmax:MAX" ... za Tmax se uporabi MAX, za osttale AVG
* Polje SortBy:
  + tabela se uredi po stolpch, ki so navedane v tem polju
  + način urejanja je naveden z uporabo dvopičja, ki mu sledi podatek o načinu urejanja (glej 2.2.1.3.). Primer: "N:-" uporaba numeričnega urejanja, podajoče.
  + privzeto urejanje: numerično, naraščajoče (+).
* Polje Count
  + Ob izbrani opciji Count=1, poizvedba vrne tabelo z eno vrstico in N stolpci, za vsak izbran algoritem po en stolpec. Vrednost v tem stolpcu število vrstic, ki jih za pripadajoči algoritem filter NE izloči.
  + Primer: če je filter Tavg < 30, dobim za vsak izbran algoritem podatek o tem, koliko meritev se je končalo prej kot v 30ms.
  + Če se filter uporablja v kombinaciji s parametrom v filtru, potem ima rezultat toliko vrstic, kolikor različnih vrednosti zavzame filter.

#### Primer uporabe filtra in polja count:

Imamo dva algoritma, A in B, ter izhodni parameter Tavg.

|  |  |  |
| --- | --- | --- |
| Test | A.Tavg | B.Tavg |
| test1 | 12 | 9 |
| test2 | 14 | 14 |
| test3 | 15 | 16 |
| test4 | 14 | 16 |
| test5 | 23 | 17 |

a) Filter: Tavg < 15 COUNT=0

|  |  |  |
| --- | --- | --- |
| Test | A.Tavg | B.Tavg |
| test1 | 12 | 9 |
| test2 | 14 | 14 |

(razlaga: samo v 1. in 2. Vrstici je Tavg < 15 pri obeh algoritmih)

b) Filter: Tavg < 15 COUNT=1

|  |  |  |
| --- | --- | --- |
| # | A.COUNT | B.COUNT |
| 1 | 3 | 2 |

(razlaga: pri A imajo trije testi (test1, test2 in test4) Tavg < 15,

pri B pa le dva (test1 in test2)).

b) Filter: Tavg < $1 @(10,20,5) COUNT=1

|  |  |  |
| --- | --- | --- |
| # | A.COUNT | B.COUNT |
| 10 | 0 | 1 |
| 15 | 3 | 2 |
| 20 | 4 | 5 |

(opomba: 2. vrstica je enaka edini vrstici v primeru b)

### Graphs

#### GraphType

A GraphType specifies a type of a graph to be ploted. ALGator supports the following graph types:

* line
* stair
* bar
* box
* pie
* area
* donut

#### Graph

A graph describes how data produced by a query are presented.

**File type**: JSON

**EntityID:** Graph

**File extension**: atgd

**File location**: <project\_root>/report/GraphName.atgd

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Name | String | Name of graph |
| Description | String | Description of a graph |
| Xaxis | String | The name of the column that contains x axis data |
| Yaxes | String[] | The names of the columns that contains y axis data |
| GraphTypes | GraphType[] | Types of a graph to be shown. |
| XaxisLabel | String | X axis label |
| YaxisLabel | String | Y axis label |

- graph contains Yaxes.length \* GraphTypes.length series

Example:

{

"Description" : "Average time for best three algorithms",

"Xaxis" : "N",

"Yaxis" : ["Java7.Tavg", "Hoare.Tavg"],

"GraphTypes" : ["Line", "Stair"],

"XaxisLabel" : "N",

"YaxisLabel" : ["Average time"]

}

### Tables

#### table

A graph describes how data produced by a query are presented.

**File type**: JSON

**EntityID:** Graph

**File extension**: attd

**File location**: <project\_root>/report/TableName.atgd

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Name | String | Name of table |
| Description | String | Description of table (caption) |
| Columns | String[] | Columns to be displayed. |

Example:

{

"Description" : "Average time of algorithms",

"Columns" : ["N", "\*.TAvg"]

}

### Reports

Reports are presentations of results used to present data produced by a query. Result can be presented on two different ways: as a table containing rows and columns as generated by query or as a graph.

#### ReportType

Currently two ReportTypes are supported:

* Graph
* Table

#### Report

**File type**: JSON

**EntityID:** Report

**File extension**: atrp

**File location**: <project\_root>/reports

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Name | String | Name of report |
| Title | String | Title of this report |
| Description | String | Detailed description of data presented by this report |
| Query | String | Filename of the query (or query itself if string starts with { ) that produces data for this report |
| Table | Table | Name of table description entity. Value of this field can be either filename or json description itself or empty (don't display table) or  Default (all columns will be displayed). |
| Graph | Graph | Name of graph description entity.  Value of this field can be either filename or json description itself or empty (no graph to present) |

Example 1:

{

"Title" : "Minimum time of execution",

"Description" : "This table presents minimum time of ...",

"Query" : "query1.atqd",

"Table" : "default",

"Graph" : "graph1.atgd",

}

Example 2:

{

"Title" : "Minimum time of execution",

"Description" : "This table presents minimum time of ...",

"Type" : "Graph",

"Query" : {

"Description" : "Demo query",

"Algorithms" : ["QuickSort AS QS"],

"TestSets" : ["TestSet1 AS TS1"],

"TestParameters" : ["N"],

"ResultParameters" : ["TMin", "TMax"

},

"Graph" : {

"Description" = "Average times ...",

"Xaxis" = "N",

"Yaxis" = ["Java7.Tavg", "Hoare.Tavg"],

"GraphTypes" = ["Line", "Stair"],

"XaxisLabel" = "N",

"YaxisLabel" = ["Average time"]

}

}

### Configuration of the ALGator system

#### CompCap (computer capability)

Computers in the system play different roles depending on their capabilities.

Capability Meaning

AEE\_EM this computer can execute measurements of time

AEE\_CNT this computer can execute measurements of counters

AEE\_JVM this computer can execute measurements of jvm bytecodes

AEE\_QUICK this computer can execute quick tests

WEB this computer is a web server (ALGator homepage)

#### Computer

Describes the computer.

**File type**: JSON

**EntityID:** Computer

**Computer:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| ComputerID | String | ID of this computer; if ALGator can not identify the ID, CO is assigned as generic id |
| Name | String | Name of the computer |
| Description | String | Description of the computer (including comp. specificities) |
| IP | String | IP address of this computer |
| Capabilities | CompCap[] | The capabilities of this computer |

Example

{

ComputerID : "C1",

FamilyID : "F1",

Name : "kepica",

Description : "My notebook",

CompIP : "212.235.189.120",

Capabilities : ["AEE\_EM", "AEE\_CNT"]

}

#### Computer Family

Describes the hardware and software information of computers. All computers in one family have the same characteristics (hardware, operating system and instaled software) .

Intent: each computer in the system belongs to a computer family. The results produced by two computers belonging to the same family are equivalent.

**File type**: JSON

**EntityID:** Family

**Family:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| FamilyID | String | ID of the family |
| Name | String | Name of the family |
| Description | String | Description of this family |
| Platform | String | The platform (i.e. Linux, ...) |
| Hardware | String | Info. about hardware |
| SystemType | String | Values: 32 or 64 |
| Computers | Computer[] | A list of all computers belonging to this family |

Example:

{

FamilyID : "F1",

Name : "Mac",

Description : "My Mac computer (kepica)",

Platform : "MacOS, 10.9.5",

Hardware : "2,66GHz Intel Core 2Duo, 8GB DDR3",

SystemType : "64",

Computers : [

{

ComputerID : "C1",

FamilyID : "F1",

Name : "kepica",

Description : "My notebook",

IP : "212.235.189.120",

Capabilities : ["AEE\_EM", "AEE\_CNT"]

}

  ]

}

#### Global configuration

The configuration of the ALGator system placed on (shared) data\_root folder. Every computer that shares data\_root folder can read/write configuration files. This configuration resides in <data\_root>/global\_config folder consists of

config.atgc // configuration of ALGator system

schema // folder with schema (see 5.2.1.3 for details)

##### config.atgc

**File type**: JSON

**EntityID:** Config

**File extension**: atgc

**File location**: <data\_root>/global\_config/config.atgc

**Fields in computer.atcf file:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Families | Family[] | An array of computer families |

#### Local configuration

A configuration of the this ALGator computer.

**File type**: JSON

**EntityID:** Config

**File extension**: atlc

**File location**: <algator\_root>/local\_config/config.atlc

**Fields:**

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| FamilyID | String | The identification of computer's family |
| ComputerID | String | The identification of computer |
| VMEP | String | A path to the vmep executable |
| VMEPClasspath | String | Classes (or JARs) that have to be added to the class when executing VMEP |
| TaskServerName | String | IP or computer name of a task server; defult: localhost |

# Izvajanje algoritmov

## Osnovni pojmi

* KONFIGURACIJSKE DATOTEKE

Vse entitete sistema ALGator so opisane v konfiguracijskih datotekah (ena datoteka opisuje eno entiteto). Format konfiguracijskih datotek je JSON.

Primer: izsek datoteke, ki opisuje projekt:

{

"Project" : {

"Name" : "Sorting",

"Description" : "Testing several sorting methods",

....

}

}

* PARAMETER

Spremenjlivka, ki jo uporabljamo za opis neke lastnosti. Vsak parameter ima svoje ime, tip in vrednost. Parameter je lahko zapisan v konfiguracijski datoteki ali pa »živi« znotraj javanskega razreda.

* PROBLEM

Problem, ki ga rešujemo. Primer: urejanje podatkov, trgovski potnik, krčenje datotek, linearno programiranje, …

* PROJEKT

Projekt je skupek konfiguracijskih datotek, s katerimi v sistemu ALGator opišemo vse komponente problema.

* NALOGA (Testcase)

Naloga danega problema. Primer za problem urejanja podatkov: ena tabela, ki jo je potrebno urediti

* MNOŽICA NALOG (Testset)

Naloge, ki predstavljajo najmanjšo enoto reševanja (ob enem zagonu sistema ALGator rešimo eno množico nalog).

* ALGORITEM

Javanski razred, ki reši eno nalogo danega problema.

* IZVRŠEVALEC (Executor)

Orodje, s katerim poženemo dani algoritem nad dano množico nalog. Gre za javanski razred, ki je zapakiran v JAR datoteko. Izvrševalec poženemo iz ukazne vrstice; pri tem podamo parametre, ki opisujejo projekt, algoritem in množico nalog.

* REŠITEV NALOGE

Skupek parametrov, ki opisujejo obnašanje algoritma pri reševanju dane naloge: poraba časa, pravilnost rezulata, poraba virov, ...

* REŠITEV MNOŽICE NALOG

Datoteka, ki vsebuje rešitve za vse naloge dane množice nalog v CSV obliki: rešitev vsake naloge je zapisana v svoji vrstici, parametri znotraj vrstice so ločeni s podpičjem.

* FAZA IZVAJANJA ALGORITMA

Posamezen algoritem je lahko razdeljen na faze, ki jih označujemo z zaporednimi številkami od 1 do 9. Primer: faza 1: predprocesiranje, faza 2: urejanje podatkov, faza 3: postprocesiranje. Faza 0 zavzema celotno izvajanje algoritma (vsota vseh definiranih faz). Če programer ne uporablja faz, se vse zgodi v fazi 0. Posamezno fazo programer vklopi z rezervirano besedo //@SET\_PHASE{phase\_num}. Uporaba faz se pozna pri izpisu končnega rezultata – vsi števci in časovniki se izpišejo večvrednostno – vrednost za fazo 0 (ki je vsota vrednosti ostalih faz) + vrednosti za posamezno definirano fazo.

## Naloge (TestCases) in množice nalog (TestSets)

* Izvrševalec z danim algoritmom reši vse naloge dane množice nalog.
* Za opis ene naloge v javi uporabljamo razred TestCase, ki vsebuje množico parametrov (generično ime problema, velikost problema, ...). Ker se opisi nalog posameznih problemov med seboj močno razlikujejo, mora skrbnik projekta napisati razred

[Project]TestCase extends TestCase

v katerem predvidi podatkovne strukture za hranjenje podatkov o nalogi.

Primer: pri projektu Sort (urejanje podatkov) za hranjenje tabele podatkov v SortTestCase definiram tabelo

public int [] arrayToSort;

* Ker je način zapisa naloge močno odvisen od problema (primer: naloga faktorizacije je podana z eno celoštevilsko vrednostjo, naloga urejanja s tabelo števil, naloga ZIP pa je podana z datoteko), je opis nalog prepuščen skrbniku projekta.

ALGator predpostavlja naslednje:

* + Opis množice nalog je podan z eno konfiguracijski datoteko tipa TestSet (glej poglavje 2.2.4.1). V tej datoteki je poleg imena in opisa ter števila nalog te množice podan tudi seznam datotek (TestSetFiles), ki se uporabljajo pri izvajanju testov.
  + Datoteka tipa TestSet ter vse datoteke s seznama TestSetFiles, se nahajajo v direktoriju tests danega projekta. Poti do datotek v TestSetFiles, so relativne glede na tests.
  + Polje DescriptionFile v TestSet datoteki opisuje ime datoteke, v kateri je opisan testset. Ker je opis testseta močno odvisen od projekta, je tudi format te datoteke povsem odvisen od projekta.

Skrbnik projekta mora predvideti, kako bo na podlagi podatkov, zapisanih v datoteki tipa TestSet pridobil podatke o vseh nalogah te testne množice.

* Za »sprehod« po dani množici nalog uporabljam razred AbstractTestSetIterator. Naloga tega razreda je, da uporabniku dostavi vse naloge dane množice. V ta namen razred AbstractTestSetIterator predvideva naslednje metode:
  + void initIterator()

Ko se pokliče ta metoda, je v atribut testSet že nastavljen. Naloga metode initIterator() je, da pripravi vse potrebno, da bo klic ostalih metod potekal nemoteno. Običajno ta metoda odpre konfiguracijsko datoteko (opisano v prejšnji alineji) in prebere (ali pa se pripravi za branje) podatkov o nalogah.

* + void hasNext();

Vrne true, če je v tej testni množici še kakšna naloga in false sicer.

* + void readNext();

Prebere in v internih strukturah shrani naslednjo nalogo.

* + TestCase getCurrent();

Vrne nalogo, ki je bila prebrana z zadnjim readNext().

Razred AbstractTestSetIterator je abstrakten (vse zgoraj naštete naloge so abstraktne). Naloga skrbnika projekta je, da napiše naslednika tega razred: [Project]TestSetIterator.

Najpogostejši primer zapisa testne monžice je tak, da v opisni datoteki (DescriptionFile) vsaka vrstica pripada natanko enemu testu. Za sprehod čez teste, zapisane na ta način, se ob vsaki zahtevi po novem testu (readNext()) prebere ena vrstica iz opisne datoteke, na podlagi podatkov, zapisanih v tej vrstici, pa se kasneje (metoda getCurrent()) generira testni primer. Obdelavo take opisne datoteke skoraj v celoti podpira razred LineTestSetIterator, ki implementira vse metode razreda AbstractTestSetIterator, razen metode getCurrent(). Torej: če skrbnik projekta pripravi testne množice na tak način, da v opisni datoteki vsaka vrstica opisuje natanko en test, potem lahko napiše naslednika razreda LineTestSetIterator in implementira le metodo getCurrent() (ki iz prebrane vrstice ustvari TestCase). V nasprotnem primeru (če je zapis testnih množic bolj kompleksen), pa mora napisati naslednika razreda AbstractTestSetIterator in implementirati vse metode.

Primer izvedbe metode getCurrent() v razredu SortTestSetIterator je prikazan na sliki Slika 1.



Slika 1: Primer zapisa nalog in implementacije TestSetIterator-ja za problem Sort.

## Algoritmi

* Za izvajanje algoritmov uporabljamo razred AbsAlgorithm, ki vsebuje naslednje metode:
  + ErrorStatus init(TestCase test)

Metoda poskrbi, da se iz podatkov, ki so zapisani v objektu test, ustvarijo podatki, na katerih se bo izvedel test. Pri tem se morajo opraviti vsi časovno zahtevni postopki v zvezi z obdelavo vhodnih podatkov. Po koncu te metode mora biti vse pripravljeno za dejansko izvajanje algoritma.

* + void run()

V tej metodi se kliče metoda execute(<parameters>). Parametri metode execute() so odvisni odvisni od problema in jih v času ustvarjanje projekta predvidi skrbnik. Čas izvajanja metode run() se šteje kot čas izvajanja algoritma, zato morajo biti parametri, ki se podajo v metodi execute(), že prej skrbno pripravljeni (v metodi init()). Priprava parametrov znotraj metode run() po krivici veča časovno zahtevnost algoritma!

* + ParameterSet done();

Metoda zbere vse parametre (vhodni parametri, parametri izvajanja in časovni parametri) in jih vrne v ParameterSet obliki.

Razred AbsAlgorithm je abstrakten. Naloga skrbnika projekta je, da napiše naslednika tega razred -[Project]AbsAlgorithm - in v njem napiše vse tri zgoraj omenjene metode. Poleg tega mora deklarirati atribute za shranjevanje podatkov o nalogi (v te atribute bo metoda init() prepisala podatke o nalogi iz podanega TestCase-a) ter abstraktno metodo execute(...) s problemu prirejenim naborom parametrov.

* Naloga raziskovalca je, da implementira naslednika razreda [Project]AbsAlgorithm. Da bi preprečili morebitno zmedo, se raziskovalcu posreduje poenostavljeno verzijo tega razreda – razred, ki je direkten naslednik razreda Object in vsebuje le abstraktno metodo execute(...). Razred, ki je naslednik tega poenostavljenega razreda, bo hkrati (brez sprememb kode) tudi naslednik pravega razreda [Project]AbsAlgorithm. Na ta način raziskovalcu ni treba vedeti za zapleteno strukturo pravega očeta in se lahko osredotoči samo na implementacijo algoritma, hkrati pa bo njegov razred brez sprememb uporaben v širšem kontekstu celotnega sistema.

Primer implementacija razreda AbsAlgorithm in njegovih naslednikov za konkreten problem je prikazan na sliki Slika 2.



Slika 2. Razred AbsAlgorithm in njegovi nasledniki

## Rešitev naloge in zapis rezultatov

Rešitev ene naloge je sestavljena iz parametrov. Te delimo na

* Vhodni parametri

Vhodni parametri opisujejo lastnosti naloge (na primer: velikost problema N, skupina problema, ...). Zapisani so v datoteki, ki opisuje test. Vhodne parametre prenesemo v datoteko z rešitvami za lažjo analizo rezultatov (primer: urejanje in združevanje rešitev po velikosti vhoda).

* Izhodni parametri

Izhodni parametri opisujejo lastnosti algoritma. Primer: numerična rešitev naloge, kakovost rešitve, pravilnost rešitve, ... (compresionRate, myResult/optResult, OK/NOK, ..). Izhodni parametri se izračunajo v metodi [Project]AbsAlgorithm.done().

* Časovni parametri.

Časovni parametri opisujejo čas(e) izvajanja. Osnovni časovni parameter meri čas izvajanj celotnega algoritma (čas izvajanja metode [Project]AbsAlgorithm.run()). Če uporabnik definira faze ( @SET\_PHASE), se izmerijo tudi časi trajanja posamezne faze.

* Števci

Števci povejo, koilkokrat se je med izvajanjem algoritma na danem testu izvedla vrstica, ki vsebuje ukaz //@COUNTER{counter\_name, 1}. Primer: če v algoritem za urejanje podatkov pred vsako vrstico, v kateri primerjamo dva podatka, zapišemo //@COUNTER{compare, 1}, bo na koncu izvajanja algoritma števec compare vseboval število vseh primerjav, ki so se izvedle.

* JVM parametri

Število posameznih java bytecode ukazov, ki so se izvedli med izvajanjem algoritma.

Algoritmi se izvajajo nad celotno množico nalog (ob enem izvajanju se izvedejo vsi testi podane množice). Rezultati izvajanja se zapišejo v izhodno datoteko, v kateri vsakemu testu pripada ena vrstica s parametri. Kateri parametri se nahajajo v tej vrstici in kakšen je njihov vrstni red, je določeno s pripadajočo atrd konfiguracijsko datoteko.

### Privzeti izhodni parametri

Vsaka vrstica z rešitvijo vsebuje 4 privzete izhodne parametre:

* ime algoritma,
* ime testne množice,
* ime testa,
* parametre uspešnosti testa.

Ime testa mora biti enolično za dano testno množico (dva testa znotraj iste testne množice ne smeta imeti istega imena). Če združimo več datotek z rezultati, prvi trije podatki enolično določajo test.

Parameter uspešnosti testa pove, ali se je izvajanje algoritma končalo v predvidenem času (parameter TimeLimit v atts datoteki). Parameter ima dve vrednosti: PASS (izvajanje se je končalo) ali KILLED (izvajanje je bilo ustavljeno).

## Meritve

Izračun parametrov se opravi z več meritvami (primer: z eno meritvijo izmerimo časovne, z drugo pa jvm paramtere). Ena meritev se izvede nad celotno množico nalog. Rezultati vsake meritve so zapisani v samostojni datoteki. Za par (algoritem, množica nalog) imamo zato več izhodnih datotek, za vsako meritev po eno.

Vsebina izhodnih datotek (katere parametre vsebuje) je opisana v pripadajoči atrd datoteki (proj/Project-[ext].atrd). Ime izhodne datoeke je [Algoritem]-[TestSetName].[ext]. Pri tem je [ext] končnica, specifična za to meritev (em, cnt in jvm).

### Meritve parametrov izvajanja

Ime atrd datoteke: proj/Project-em.atrd

Končnica: izhodne datoteke: .em

Glavni namen te meritve je izračun izhodnih ter časovnih parametrov.

Časovni parametri se izmerijo avtomatsko (čas izvajanja metode run()). Vsak test se požene večkrat (parameter TestRepeat v atts datoteki). V izhodno datoteko se zapiše ena številska vrednost, ki jo sistem izračuna na podlagi izmerjenih časov po različnih formulah. Katera formula se bo uporabila, določi avtor atrd datoteke. Primer: če bosta v atrd datoteki definirana parametra

{

"Name" : "Tmin",

"Description" : "Sorting MIN time",

"Type" : "timer",

"Subtype" : "0 MIN"

}

{

"Name" : "Tavg",

"Description" : "Sorting AVG time",

"Type" : "timer",

"Subtype" : "0 AVG"

}

bosta v izhodni datoteki zapisana minimalni in povprečni čas izvajanja algoritma pri TestRepeat ponovitvah tega testa. Osnovni namen ponavljanja je izničenje naželjenih vplivov iz okolja. Pri natančnem ocenjevanju izvajanja algoritma običajno uporabimo minimalno vrednost (pri tem izvajanju je prišlo do najmanj motenj iz okolja).

Izhodne parametre se izračuna v metodi [Project]AbsAlgorithm.done() na podlagi podatkov o izvajanju algoritma.

### Meritve s števci

Ime atrd datoteke: proj/Project-cnt.atrd

Končnica izhodne datoteke: .cnt

Koda algoritma lahko vsebuje ukaz //@COUNT{counter\_name, value} ki poveča vrednost števca counter\_name za value (primer: //@COUNT{SWAP, 1} poveča vrednost števca SWAP za 1). Po končanem izvajanju izvajalni sistem vrne vrednost vseh števcev, ki so navedeni v polju ResultParameters v datoteki proj/Project-cnt.atrd. (to polje določa tudi vrstni red zapisa števcev v izhodno datoteko). Polje TestParameters v atrd datoteki je pri tej meritvi opcijsko.

Meritev s števci vsak algoritem nad posameznim testom požene samo enkrat. Parameter TestRepeat v tej datoteki se ignorira.

Pred izvajanjem meritve s števci se izvorna koda XXXAlgorithm.java prepiše v XXXALgorithm\_COUNT.java. Potem se prevede in izvede XXXALgorithm\_COUNT.class.

Pri prepisu datoteke se

1) vse pojavitve

//@COUNT{cnt\_name, value}

nadomestijo z

Counters.addToCounter("cnt\_name", value);

2) vse vrstice, ki vsebujejo //@REMOVE\_LINE, odstranijo.

Primer: Izvorna koda Koda v \_COUNT datoteki

|  |  |
| --- | --- |
| /\*//@REMOVE\_LINE  //@COUNT{CMP, 1}  if (a[e1] != a[e2]) {  //@COUNT{CMP, 1}  }  \*///@REMOVE\_LINE    **if** (a[e1] != a[e2] && a[e2] != a[e3]) {  //@COUNT{CMP, 1}  **while** (a[++less] < pivot1) {  //@COUNT{CMP, 1}  }  } | Counters.addToCounter("CMP", 1);  **if** (a[e1] != a[e2]) {  Counters.addToCounter("CMP", 1);  }  **if** (a[e1] != a[e2] && a[e2] != a[e3]) {  Counters.addToCounter("CMP", 1);  **while** (a[++less] < pivot1) {  Counters.addToCounter("CMP", 1);  }  } |

### Meritve s prirejenim JVM

Ime atrd datoteke: proj/Project-jvm.atrd

Končnica izhodne datoteke: .jvm

Pri tej meritvi štejemo število posameznih bytecode ukazov ter velikost porabljenega pomnilnika (ukazi new, newarray, ...). Za to meritev uporabljamo drugačno izvajalno okolje (prirejen JVM, ki šteje ukaze).

## Nov projekt

Ko se naredi nov projekt, je treba biti posebej pozoren na nasledenje podrobnosti.

* V razredu [Project]TestSetIterator.java
  + V metodi getCurrent() je treba v TestCase, ki se vrne kot izhod, dodati vse vhodne parametre (parametri, ki so opisani v atrd datoteki pod TestParameters).
  + Metoda getCurrent()mora poleg ostalih OBVEZNO vrniti še parameter “Test\_ID” – enotna (zaporedna) številka testa znotraj danega testseta.
* V razredu [Project]AbsAlgorithm.java
  + Metoda done() mora poleg vhodnih parametrov v ParameterSet vrniti tudi vse izhodne parametre testa (parametri, ki so opisani v atrd datoteki pod ResultParameters). Vhodne parametre prebere iz testa, izhodne parametre pa doda na podlagi rezultatov metode execute(). Priporočljiva implementacija metode done():

public ParameterSet done() {

ParameterSet result = new ParameterSet(pppTestCase.getParameters());

EParameter par1 = new EParameter(...);

result.addParameter(par1, true);

...

return result;

}

# Izvajanje algoritmov v algator.Execute

**java algator.Execute Project –a algorithm –t testset –m measurementType**

- v main metodi določi parametre Project, Algorithms, TestSets, MeasurementType

- za vsak Algorithm iz Algorithms in TestSet iz TestSets kliče

**Executor.algorithmRun(project, algorithm, testset, measurementType)**

= preveri pravilnost in obstoj projekta, algoritma in testseta

= po potrebi prevede vse javanske datoteke projekta

- **projectMakeCompile(project)**

= po potrebi prevede vse javanske datoteke algoritma

- **algorithmMakeCompile(project, algorithm, measurementType)**

= ustvari instanco testSetIteratorja (tsIt)

= če (mType == EM) ali (mType==CNT):

**ExternalExecutor.iterateTestSetAndRunAlgorithm(project, alg, tsIt, resDesc, mType)**

+ za vsak testniPrimer (**tsIt.getCurrent()**)

o) generira **testCase**

o) generira novo instanco algoritma (**alg**) in mu doda testcase (**alg.init(testCase)**)

o) algoritem serializira in ga zapiše v datoteko **tmpFolder/alg.ser.test** in kliče

**errorStatus = ExternalExecutor.runWithLimitedTime(tmpFolder)**

x) s pomočjo metode **ExternalExecute.runWithExternalJVM(tmpFolder)** v

drugem JVM asinhrono požene

**algator.ExternalExecute tmpFolder**

- v main metodi prebere argument (tmpFolder) in kliče

**ExternalExecute.run()**

- za vsak i=1 ... **timesToExecute**

- prebere algoritem **alg** iz datoteke **tmpFolder/alg.ser.test**

- kliče **alg.run()** in meri čas

- shrani čase v interno tabelo

- poveča števec v komunikacijski datoteki (tmpFolder/alg.com)

- v zadnjo instanco algoritma shrani vse čase iz interne tabele

- algoritem zapiše v datoteko **tmpFolder/alg.ser.result**

x) med izvajanjem algoritma v VMEP VM preverja vsebino komunikacijske datoteke;

če ugotovi, da se je izvajanje enega testa ustavilo (test se ni končal v

**timeForOneExecution**), ubije drug JVM in vrneErrorStatus.PROCESS\_KILLED

x) če se izvajanje konča preveri izpis na stdout

- če obstaja izpis: vrne ErrorStatus. PROCESS\_CANT\_BE\_CREATED

- sicer vrne ErrorStatus.OK

o) če errorStatus == OK

- prebere algoritem iz datoteke **tmpFolder/alg.ser.result**

- glede na tip testa (**measurementType**) iz prebranega algoritma pridobi

parametre testa (časi, števci, jvm)

- zapiše eno vrstico v datoteko alg-ts-mType

o) sicer

- v datoteko alg-ts-mType zapiše 4 default parametre in sporočilo o napaki

= če (mType == JVM):

**VMEPExecutor.iterateTestSetAndRunAlgorithm(project, alg, tsIt, resDesc, mType)**

+ za vsak testniPrimer (**testID** je zaporedna številka testa) požene

**errorStatus = VMEPExecutor.runWithLimitedTime**

**(pName, algName, tsName, testID, tmpFolder, dataRoot, timeLimit)**

o) s pomočjo metode **VMPEExecute.** **runWithVMEP(proj,alg,testset,test,tmpFolder, ...)**

asinhrono požene

**jamvm algator.VMEPExecute proj alg testset test tmpFolder**

x) preveri pravilnost vhodnih podatkov in požene algoritem na danem testu

x) če se je test izvršil normalno

- prebere VMEP parametre ter rezultate zapiše v komunikacijsko datoteko

- program se konča z errorCode = VMEPErrorStatus.OK

x) sicer

- program se konča z errorCode = VMEPErrorStatus.\*

o) med izvajanjem algoritma v VMEP JVM se meri čas; ko se izvajanje VMEP JVM konča,

shrani exitCode procesa jamvm, nato

- **če je potekel čas timeLimit :** vrne VMEPErrorStatus.KILLED

- če je (exitCode != 0) : vrne VMEWErrorStatus(exitCode)

- sicer: : vrne VMEPErrorStatus.OK

+ v pripadajočo result datoteko zapiše eno vrstico in sicer:

- če **errorStatus == OK**: prvo vrstico iz komunikacijske datoteke

- če **errorStatus != OK:** 4 default parametre + sporočilo o napaki

# Beleženje zgodovine

## Ob ročnem zagonu ALGatorja:

- vsa logiranja se dogajajo v folderju <log\_folder>=data\_root/log

- mesto izpisa (0=NONE/1=STDOUT/2=FILE/3=BOTH) določimo s stikalom -log

- če so parametri P, A, T in EM pravilno podani, se status opravila v vsakem

primeru zapiše v datoteko <log\_folder>/tasks/<P\_A\_T\_E>.status

(status se med izvajanjem spreminja, na koncu postane 100%)

- če stikalo -log vsebuje FILE bit (2), potem se logira na dve mesti:

- <log\_folder>/algator.log

- <log\_folder>/tasks/<P\_A\_T\_E>.history

- če želiš beležiti zgodobino, obvezno dodaj -log 2 ali -log 3

## Zagon algatorja preko TaskClienta

... TODO: preglej stanje, popiši in po potrebi popravi tako, da bo smiselno