ALGator

Avtor: Tomaž Dobravec (2013, 2014)

Verzija: 0.1

1 Opis sistema 3

1.1 Namen 3

1.2 Vloge v sistemu 3

1.2.1 Sistemski administrator 3

1.2.2 Administrator projekta 3

1.2.3 Raziskovalec 4

1.2.4 Gost 4

1.3 Primer uporabe 4

1.4 Podsistemi sistema ALGator 4

2 Zgradba sistema 6

2.1 The structure of the <algator\_data\_root> folder 6

2.1.1 The conf folder 6

2.1.2 The schema folder 6

2.1.3 The projects folder 6

2.2 System configuration 7

2.2.1 Building blocks 7

2.2.2 Projects 9

2.2.3 Algorithms 10

2.2.4 Tests 11

2.2.5 Results 12

2.2.6 Queries 14

3 Izvajanje algoritmov 17

3.1 Osnovni pojmi 17

3.2 Naloge (TestCases) in množice nalog (TestSets) 18

3.3 Algoritmi 22

3.4 Rešitev naloge in zapis rezultatov 24

3.4.1 Privzeti izhodni parametri 24

3.5 Meritve 25

3.5.1 Meritve parametrov izvajanja 25

3.5.2 Meritve s števci 26

3.5.3 Meritve s prirejenim JVM 26

3.6 Nov projekt 26

4 Podsistemi sistema ALGator 30

4.1 Konfigurator 30

4.1.1 JSONWebEditor (JWE) 30

4.2 Izvajalnik 36

4.3 AlgoTestSystem 37

4.3.1 Datotečni sistem 37

4.3.2 Opravila 38

4.3.3 Ukazi 38

# 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 med različnimi algoritmi 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.

### Sistemski administrator

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

### Administrator projekta

Administrator 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.

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

### Raziskovalec

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

### Gost

Gost lahko vidi podatke o javnih algoritmih javnih projektov.

## Primer uporabe

V tem razdelku je opisan najpogostejši način uporabe sistema ALGator.

* Administrator sistema postavi celotni sistem in objavi povezavo do spletne strani sistema.
* Administrator 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.
* Administrator 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. Algoritem se 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 (private 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 administrator projekta.

## Podsistemi sistema ALGator

Sistem ALGator je sestavljen iz več podsistemov, ki so medseboj povezani s konfiguracijskimi datotekami. Nekateri podsistemi te datoteke pišejo in berejo, nekateri pa jih samo berejo in obdelujejo.

**Konfigurator (Configurator).** Konfigurator skrbi za urejanje konfiguracije celotnega sistema, ki je zapisana v tekstovnih datotekah različnih formatov (JSON, CSV, …). Konfigurator pozna uporabnike sistema in vsakemu dovoli opravljati naloge, skladno z njegovimi pravicami.

Konfigurator je podrobneje opisan v poglavju 0.

**Izvajalnik (Execution engine, EE).** Izvajalnik skrbi za izvajanje algoritmov in za zapis rezultatov izvajanja. Ob klicu izvajalnika se požene podani algoritem na vseh testih danega nabora testov, rezultati izvajanja pa se zapišejo v datoteko z vnaprej določeno lokacijo in imenom (obstoj te datoteke in pravilno število vrstic v njej potrjuje uspešnost izvajanja posameznega testa).

Izvajalnik je podrobneje opisan v poglavju **Error! Reference source not found.**.

**Analizator.** Analizator je orodje za analizo podatkov, pridobljenih a pomočjo izvajalnika. Analizator skrbi za izvajanje poizvedb (iz vseh podatkov izlušči le tiste, ki so za dano analizo zanimivi) ter za analizo (npr. iz tabele, ki vsebuje dva stolpca – N in čas izvajanja za nek algoritem – zna izračunati (emipirično) časovno zahtevnost algoritma).

**Prikazovalnik.** Prikazovalnik skrbi za prikaz podatkov o sistemu, projektih in o algoritmih. Ker so vsi podatki za prikaz zapisani v konfiguracijskih datotekah, prikazovalnik poskrbi le za pravilno razporditev teh podatkov po spletnih straneh sistema. Med prikazanimi podatki prikazovalnik smiselno umesti tudi povezave s podsistemi (npr. na strani, kjer prikaže raziskovalčev algoritem, prikaže tudi gumb »Izvedi«, s katerim se kliče izvajalnik; na strani z rezultati izvajanja prikaže gumb »Analiza časa izvajanja«, s katerim se kliče analizator, ...).

# Zgradba sistema

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

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

<algator\_data\_root>

conf // the configuration folder

schema // schema for all the entities of the system

projects // root for the projects

### The conf folder

The configuration of the ALGator system.

### The schema folder

This folder contains the atjs files that are used by the configurator (i.e. jwe) to properly edit the entity files.

### The projects folder

This 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

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

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

TestSetName1.atts // one or more testset files

results // folder with results (attr files)

queries // folder for predefined queries

\*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\_data\_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
* > lexicographic 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)
* int integer value
* double double value; for decimal separator “.” is used
* string a string inside quotes “”

Each type can have a subtype.

|  |  |  |
| --- | --- | --- |
| 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)    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 | ResultParameterType | 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> = <algator\_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 |
| Author | String | Author of the project |
| Date | String | The date of the last change |
| Algorithms | String[] | a list of Algorithms |
| TestSets | String[] | a list of TestSets |
| AlgorithmClass | String | the name of java template file for the algorithms of this project; |
| TestCaseClass | String | the name of the TestCase class for this project |
| TestSetIteratorClass | String | a name of the TestSetIterator |

Note: the three source files (AlgorithmTPL, TestSetClass and TestSetIteratorClass) are placed in the src subfolder of a project.

{

"Project" : {

"Name" : "Sorting",

"Description" : "Testing several sorting methods",

"Author" : "Tomaz",

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

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

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

"AlgorithmTPL" : "SortAbsAlgorithm",

"TestCaseClass" : "SortTestCase",

"TestSetIteratorClass" : "SortTestSetIterator"

}

}

### Algorithms

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

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

The <algorithm\_root> folder contains:

* AlgName.atal file with the description of the algorithm
* src folder with all source files.

#### Algorithm

The configuration of an individual algorithm AlgName 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 |
| Author | String | Author of the algorithm |
| Date | String | Creating date of the algorithm |
| Classes | String[] | an array of source files |

Example: QuickSort1.atal

{

"Algorithm" : {

"Name" : "BubbleSort",

"ShortName" : "BS",

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

"Author" : "BS and Tomaz",

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

"Classes" : "BubblesortSortAlgorithm"

}

}

### 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 |
| Description | String | A detailed description of this test set |
| ShortName | String | A shortname of this test set. This description will be used in the result presentation forms. |
| 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). |
| DescriptionFile | String | A name of the file with the project dependant description of this testset. |
| TestSetFiles | String[] | 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. |
| TestRepeat | Integer | Number of times to execute a single test from this test set. |

**Example:**

"TestSet": {

"Name" : "SortAlgorithmsQuickTest",

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

"ShortName" : "SortQ",

"N" : 13,

"DescriptionFile" : "testset1.txt",

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

"TestRepeat" : 10

}

### 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.

**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; error if other error

occurs during the execution.

* parameters … parameters of a test

The parameters are listed in the order that is defined in the ResultDescription file. If there is any test-specific parameter presented in the result, it must also be defined in the ResultDescription.

Example:

quicksort;testset1;test7;done;sorted;10;35;5;7;23.5

NOTE: if an error occurs, result contains the first 4 data (algorithm, testset, test and status), following by the error description string. All such lines in result file will be omitted, but they must be there (for error tracking and because the result file must contain exactly N lines)!

### Queries

**File type**: JSON

**EntityID:** Query

**File extension**: atqd

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

**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 |

Primer poizvedbe:

{

"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:-"]

}

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.

Če imamo v poizvedbi N algoritmov in M testnih množic ter T tesnih parametrov in R paremetrov razultata, potem bo tabela vsebovala T + N \* R stolpcev.

* Polji GroupBy in Filter vplivata na število vrstic (lahko jih zmanjšata)
* 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 Filter:
  + Filter iz tabele izloči vse vrstice, ki NE zadoščajo pogojem.
  + Filtri se lahko izvaja le po parametrih testa.
  + Filter je tabela preprostih pogojev, za združevanje se uporabi operator in (&&) – 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: ==, !=
* 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 (+).

# 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 administrator 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 administratorju 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.

Administrator 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 administratorja 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 administrator 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 administrator. 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 administratorja 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). Primer: če v algoritem za urejanje podatkov pred vsako vrstico, v kateri primerjamo dva podatka, zapišemo //@COUNTER(compare), 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 MAX\_TIME 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 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.

### 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;

}

# Podsistemi sistema ALGator

## Konfigurator

Konfigurator skrbi za urejanje konfiguracije celotnega sistema, ki je zapisana v tekstovnih datotekah različnih formatov (JSON, CSV, …). Konfigurator pozna uporabnike sistema in vsakemu dovoli opravljati naloge, skladno z njegovimi pravicami.

Konfigurator sistemskemu administratorju omogoča nastavitev osnovnih parametrov sistema (podatki o izvajalnikih, direktorij projektov, ...) ter upravljanje z vsemi uporabniki sistema. Administratorju projekta omogoča spreminjanje podatkov o njegovem projektu.Raziskovalce omogoča spreminjanje podatkov o njegovem algoritmu.

Konfigurator je lahko implementiran kot spletna aplikacija ali kot program, ki se izvaja lokalno.

### JSONWebEditor (JWE)

JSONWebEditor je implementacija ALGator konfiguratorja. Je spletni program, ki je namenje urejanju JSON konfiguracijskih datotek. Za potrebe te dokumentacije program JSONWebEditor kličem z <http://localhost/jwe>.

#### JWE projekti

S programom lahko urejamo več ločenih JWE projektov. Vsak projekt ima svoje ime ter svojo $root mapo (vse datoteke enega JWE projekta so zapisane v mapi $root oziroma v njenih podmapah). Ob izbiri projekta se avtomatsko izbere tudi mapa, vsa imena datotek izbranega projekta so relativna gleden na $root (to pomeni, da, na primer, ime datoteke PROJ-Sorting/proj/Sorting.atp jwe avtomatsko razume kot $root/PROJ-Sorting/proj/Sorting.atp.

Imena JWE projektov in pripadajočih map so zapisana v tekstovni konfiguracijski datoteki, ki se bo urejala ročno.

JWE projekt izberemo ob klicu programa jwe s parametrom pName.

Primer: http://localhost/jwe?pName=AT

#### Entitete

JSON datoteke v našem sistemu uporabljamo za opis *entitet* različnih tipov, na primer, Project, Algorithm, TestSet, ResultDescription, ... Entitete različnih tipov imajo različne lastnosti. (Opomba: za lažje razumevanje pojmov entiteta, lasnost entitete, tip entitete, ... glej sliko JWE1).

JSONWebEditor zna urejati le entitete tistih tipov, za katere ima podano shemo. Sheme so zapisane v mapi $jsRoot=$root/schema. Vse datoteke v tej mapi imajo ime po tipu entitete, ki ga opisujejo, in končnico atjs (primer: Algorithm.atjs).

Urejanje točno določene entitete sprožimo tako, da programu jwe podamo tip (eType) in ime (eName) entitete (s tipom pokažemo na shemo, z imenom pa na konkretno entiteto).

**Primer**: s klicem

http://localhost/jwe?pName=AT&eType=Project&eName=Sorting

začnemo z urejanjem json datoteke Sorting.atp; shemo za to datoteko najdemo v datoteki $jsRoot/Project.atjs. V tej atjs datoteki med drugim piše tudi:

"**Filename**" : "PROJ-{}/proj/{}.atp",

s čimer je določeno mesto, datoteke za to entiteto:

$root/PROJ-Sorting/proj/Sorting.atp.

Pri odpiranju entitete lahko ob klicu programa jwe podamu tudi parametre v obliki »$i=vrednost\_i«, kjer je i zaporedna številka parametra. Pri branju atjs datoteke program jwe avtomatsko vse pojavitve niza $i zamenja z vrednost\_i.

**Primer:** v projektu Sorting odpremo algoritem BubbleSort z naslednjim klicem:

http://localhost/jwe?pName=AT&eType=Algorithm&eName=BubbleSort&$1=PROJ-Sorting

Opomba: Za boljše razumevanje glej opis tipa Entity spodaj in vsebino datoteke Algorithm.atjs.

#### Sheme

Shema opisuje tip entitete (primer: shema Algorithm.atjs opisuje tip Algorithm). Struktura sheme je povzeta po http://json-schema.org. Primer sheme je v datoteki schema/Project.atjs.

Shema je JSON datoteka, z naslednjimi lastnostmi:

* Name … ime tipa entitete, ki ga ta shema opisuje
* Filename … opis imena datoteke, ki vsebuje entiteto tipa Name
* properties … JSON objekt, ki opisuje lastnosti pridružene entitete. Za vsako lastnost entitete obstaja v properties istoimenska lastnost. Za podrobnosti glej podpoglavje Lastnosti spodaj.

Vsebina datoteke lahko vsebuje naslednje parametre:

* {} … to se ob branju atjs datoteke nadomesti z imenom entitete
* $i … to se ob branju atjs datoteke nadomesti z i-tim POST parametrom

##### JSHTML datoteke

Vsaki shemi je pridružena tudi istoimenska datoteka s končnico jshtml (datoteki Project.atjs je pridružena datoteka Project.jshtml). jshtml datoteke se nahajajo v isti mapi kot atjs datoteke. Datoteke jshtml opisujejo način izpisa entitete, ki ji pripadajo. Gre da običajen html dokument, v katerem so lahko naslednji parametri (ki jih je treba pred prikazom zamenjati s pravo vrednostjo):

* {\_ENTITY\_NAME\_} ... se nadomesti z imenom (niz) entitete
* {\_ALL\_PROPERTIES\_} ... se nadomesti s seznamom kontrol za vse lastnosti lastnosti
* {\_PROPERTY\_IME\_} ... se nadomesti s kontrolo za entiteto z imenom IME.
* {\_OTHER\_PROPERTIES\_} ... se nadomesti s kontrolami za vse še neizpisane lastnosti (tiste, ki še niso bile izpisane s pomočjo {\_PROPERTY\_IME\_}).

##### Lastnosti objekta properties

Vsaka lastnosti je podana z JSON objektom. Ime lastnosti se ujema z imenom lastnosti entitete, ki jo opisuje. Vsaka lastnost objekta properties je opisana z naslednjimi lastnostmi:

* description … opis te lastnosti; ta opis se pojavi na html strani pred kontrolo (opis za uporabnika, da ve, kaj točno vpisuje v kontrolo)
* type … tip lastnosti (natančneje opisano v naslednjem poglavju)

Poleg tega ima lahko lastnost še dodatne lastnosti, ki pa so odvisni od tipa. Nekatere med njimi so obvezne, drugi opcijski (kar je prav tako razvidno iz spodnjega opisa).

##### Tipi lastnosti

1) String

**Izpis**: TextField s preverjanjem vsebine

**Dodatni parametri**:

* pattern (optional): regularni izraz, ki opisuje dovoljene znake, ki se lahko pojavijo v tem nizu; če tega parametra ni, je vsebina lahko poljubna.

2) Integer

**Izpis**: TextField s preverjanjem vsebine

**Dodatni parametri**:

* minimum (optional; default: INT\_MIN): najmanjša vrednost, ki je lahko vpisana v tem polju
* maximum (optional; default: INT\_MAX): največja vrednost, ki je lahko vpisana v tem polju

3) Double

**Izpis**: TextField s preverjanjem vsebine

**Dodatni parametri**:

* minimum (optional; default: INT\_MIN): najmanjša vrednost, ki je lahko vpisana v tem polju
* maximum (optional; default: INT\_MAX): največja vrednost, ki je lahko vpisana v tem polju

4) File

**Izpis**: TextField (readonly) + gumb Browse + gumb Edit

**Dodatni parametri**:

* root (optional; default: "/"):

**Opis:** Gumb Browse odpre OpenDialog in uporabniku omogoči izbiro ene datoteke. Po zaprtju dialoga se pojavi gumb Upload, ki poskrbi za nalaganje izbrane datoteke v mapo root. Po nalaganju se v readonly kontrolo zapiše ime naložene datoteke.

Gumb Edit odpre naloženo datoteko v oknu in omogoči urejanje. V tem oknu sta gumba OK (shrani in zapri) ter Cancel (prekliči brez shranjevanja).

5) Files

**Izpis**: TextField[] + gumb OpenFileManager

**Dodatni parametri**:

* root (optional; default: "/"):

**Opis:** Namen te kontrole je, da uporabniku na čimbolj eleganten način omogoči upload več datotek hkrati. Rezultat (zapis v json datoteki) je tabela nizov, ki opisujejo naložene datoteke. Te nizi so lahko tudi regularni izrazi (recimo “quick\*”). Z uporabo kontrole Files moramo doseči popolnoma enak rezultat kot z uporabo kontrole File [], razlika je le v načinu dela (pri velikem številu datotek bo zelo zamudno, če bo moral uporabnik vsako posebej naložiti; poleg tega bo tudi število zapisov v tabeli zelo veliko – z uporabo regularnega izraza lahko število teh zapisov precej zmanjšamo).

Gumb OpenFileManager odpre dialog z levim in desni oknom; v levem oknu je prikazana vsebina direktorija root na serverju, v desnem pa lokalni file sistem na clientu. Okno naj omogoča prenašanje datoteke iz desne na levo stran (upload).

Ob kliku na OK v tem oknu, se dejansko izvrši Upload, polja pa se napolni z vrednostmi prenešenih datotek. Če je bila prenešena celotna mapa, se v eno od polj zapiše “ime\_mape/\*” (in ne vsako ime datoteke posebej). Ob kliku na Cancel pa se ne zgodi nič (datoteke se ne prenesejo).

Implementacija te kontrole je po moji oceni precej zahtevna, zato predlagam, da se tega lotite na koncu, k obo vse ostalo že delalo. Do takrat bomo nakako shajali s pomočjo kontrole tipa File []. Pred implementacijo bi se o tem še izdatno pogovorila.

6) Entity

**Izpis**: TextField (readonly) + gumb Edit + gumb Delete

**Dodatni parametri**:

* eType (required) … tip entitete, ki je vezana na to kontrolo.
* parameters (optional) ... tabela parametrov, ki jih podamo ob klicu te entitete;

Parametri rešujejo morebitnih odvisnosti med entitetami. S pomočjo parametrov v podrejene entitete prenesemo nastavitve iz nadrejene entitete. Ob klicu entitete se med POST (oziroma GET) paramtere dodajo vsi parametri iz tabele parameters in sicer takole: »$1=parameter\_1&...&$i=parameter\_i«.

**Primer uporabe:** entiteta Algorithm kot taka ne obstaja, saj je vedno vezana na entiteto Project. Pri urejanju entitete Algorithm je na nek način treba povedati, kateremu Projectu pripada (to je pomembno recimo pri razreševanju direktorijske strukture). Ob klicu entitete Algorithm kot prvi parameter podamo »PROJ-{}« in tako v Algorithm pripeljemo manjkajoči podatek o direktoriju. Ker v Algorithm.atjs piše

"**Filename**" : "$1/algs/ALG-{}/{}.atal",

se datoteka, ki opisuje to entiteto, se nahaja v $root/$1/algs/ALG-{}/{}.atal.

Ob kliku na gumb Edit se preveri, ali ime entitete obstaja (če obstaja, potem obstaja tudi ta entiteta, sicer jo je treba še ustvariti).

* Če ime ne obstaja, v pogovornem oknu vprašamo po imenu in ustvarimo novo (prazno) entiteto, potem nadaljujemo enako, kot če bi ime obstajalo.
* Če ime obstaja, odpremo entiteto in omogočimo urejanje.

7) Tabele []

**Opis:** Če imenu tipa sledita znaka [], imamo opravka s tabelo tega tipa. Rezultat (zapis v json datoteki) je tabela (recimo [podatek1 , podatek2]). Kontrola mora omogočati brisanje in dodajanje elementov tabele. Posamezen element izpiše tako, kot se izpišejo elementi tega tipa.

Želja: gumba GOR in DOL, s katerima se spreminja vrstni red elementov v tabeli.

**Primer:** Kontrola File izpiše en TextField z gumboma Browse in Edit. Kontrola File [] prvotno izpiše le gumb Add; ob vsakem kliku na ta gumb se pojavi vrstica tipa File (TextField + gumba Browse in Edit) z dodanim gumbom Delete (ki to vrstico zbriše).



Slika JWE1. Entiteta, lasnost entitete, tip entitete

## Izvajalnik

Izvajalnik je nameščen na dveh fizično ločenih računalnikih: na spletnem strežniku (administrative machine, AM) in na ločenem (izoliranem) računalniku, ki je namenjen izključno izvajanju ALGatorjevih algoritmov (execution machine, EM). Merjenje porabljenega časa na AM zaradi motenj (npr. zahteve strežniku) ni zanesljivo. Poleg tega lahko izvajanje večjega števila zahtevnih algoritmov na AM bistveno upočasni delovanje spletnega strežnika. Zato se na AM izvajajo le hitri testi, ki so namenjeni osnovnemu preverjenju delovanja algoritma. Vsi zahtevnejši testi se obvezno izvajajo na EM. Podatek o tem, ali se nabor testov lahko izvaja na AM, je zapisan v konfiguracijski datoteki nabora testov. Uporabnik lahko »ročno« (t.j. z ukazom na spletni strani) poganja le hitre teste, za ostale teste pa lahko uporabnik zahteva uvrstitev v čakalno vrsto za izvajanje na EM. Ti testi se bodo izvedli, ko bodo prišli na vrsto. V vmesnem času je na spletni strani prikazan status (»v čakalni vrsti« + podatek o predvidenem čakalnem času).

* !!!razmisli!!!: namesto o AM in EM lahko govorimo o nivojih zanesljivosti; računalnik nivoja 0 je nezanesljiv (AM), nivo 1 je zanesljivejši (EM), ...; pri testsetu bi bilo navedeno, najmanj koliko mora biti zanesljiv računalnik, da se ta testset lahko izvede na njem; v konfiguraciji sistema bi bilo zapisano, s katerimi računalniki razpolagamo, kakšna je njihova zanesljivost in kako se poganjajo algoritmi na njih; ob akciji »izvedi« bi se poiskal računalnik z dovolj veliko stopnjo zanesljivosti in algoritem bi se izvedel na njem; datoteke z rezultati bi imele na koncu poleg imena še podatek o zanesljivosti računalnika, na katerem se je test izvedel (.0, .1, ...).

ATSystem zna za dani Project:

1. Izvesti en nabor problemov z enim algoritmom;

* project.executor … Executor, ki bo izvršil algoritem na testni množici
* testSet = project.testSets[i] … testna množica
* algorithm = project.algorithm[i] … algoritem
* testDir = testFilesDir (projectDir + “testFiles”)
* if (testSet.copyTestFiles)

testDir = tmpFolder

copyTestFiles (project.filesToCopy) to testDir

copy testSet.testDescFile to testDir

- outputDir = projectDir + results

* executor.execute(algorithm, testSet, testDir, outputDir, notificator)
  + executor izvede algoritem na vseh testih iz TestSet-a
  + ob vsakem končanem testu kliče notifikator (da ga obvesti, da je bil test izvršen; notifikator, na primer, prikazuje napredek)
  + rezultate zapiše v datoteko outputDir/alg + test.res

b) pregledati celoten projekt in ugotovi, kateri nabori testov še niso

bili izvedeni (oziroma se zastareli) s posameznim algoritmom

c) izvesti točko b) in ob tem za vsak najden neizveden (oziroma

zastarel) test pognati točko a)

d) analizirati in prikazati rezultate, ki so zbrani v attrd datotekah po

različnih kriterijih

## AlgoTestSystem

AlgoTestSystem je sistem za izvajanje algoritmov. Sestavljen je iz izvajalnega in iz upravljalnega dela.

Izvajalni del sistema je sestavljen iz računalnika (execution machine, EM), na katerem se izvajajo algoritmi. Na EM je nameščen posebej v ta namen prilagojen operacijski sistem, ki omogoča nemoteno izvajanje in natančno merjenje časa izvajanja. V času izvajanja algoritmov na sistemu ne teče noben drug program ali servis, preprečena je tudi komunikacija preko omrežja.

Upravljalni del sistema je sestavljen iz računalnika (administration machine, AM), na katerem so shranjeni vsi podatki o algoritmih, testnih podatkih in rezultatih izvajanja. Za upravljenje s temi podatki je predviden dostop preko spletne strani.

Na računalniku EM se izvajajo algoritmi in izvaja natančno merjenje časa, zato mora biti računalnik izoliran od sveta. Edina komunikacija z zunanjim svetom je komunikacija do računalnika AM, pa še ta komunikacija poteka samo v eni smeri (EM kontaktira AM in od njega pridobi oziroma mu pošlje podatke). Na AM zato teče strežnik, na EM pa odjemalec.

Računalnik AM predstavlja komunikacijsko okno do računalnika EM. Vsa navodila, namenjena računalniku EM, se posredujejo računalniku AM. Ta jih hrani v obliki opravil (tasks), dokler EM ne vpraša zanje.

### Datotečni sistem

Na AM so shranjeni vsi podatki o projektih (algoritmi, testni podatki, rezultati, ...) v datotekah znotraj mape amProjectsRoot. Pred izvajanjem posameznega algoritma je treba nekatere datoteke iz te mape prenesti v mapo emProjectsRoot na računalniku EM, pri tem pa se na EM ustvari popolnoma enaka direktorijska struktura kot je na AM. (Ko so prenešene vse datoteke, je vsebina mape emProjectsRoot na EM identična vsebini mape amProjectsRoot na AM).

V komunikaciji med AM in EM so vsa imena datotek relativna glede na amProjectsRoot oziroma emProjectsRoot.

### Opravila

Opravilo je seznam ukazov, ki jih mora EM izvesti. Posamezno opravilo je zapisno v tekstovni datoteki v mapi amTasksFolder. Imena datotek z opravili vsebujejo časovni žig (timestamp) na tak način, da so imena starešjih datotek leksikografsko pred imeni mlajših datotek. (Posledično strežnik na AM naslednje opravilo izbere tako, da leksikografsko uredi imena datotek z opravili in izbere najmanjšega).

Ko strežnik opravilo posreduje računalniku EM, mora datoteko tega opravila prestaviti iz mape amTasksFolder v mapo amDoneTasksFolder.

### Ukazi

Sistem trenutno podpira naslednje ukaze: copy, run in wait, v prihodnosti se seznam podprtih ukazov lahko razširi.

#### Ukaz copy

**Sintaksa:** copy(ime\_datoteke, smer\_kopiranja)

**Opis:** Ukaz copy prenese datoteko iz AM na EM ali obratno. Smer kopiranja je določena z drugim parametrom, ki je lahko AM2EM ali EM2AM. Vsa imena datotek so relativna glede na amProjectsRoot oziroma emProjectsRoot.

**Dodatne zahteve:** da bi preprečili nepotrebno kopiranje, si pred kopiranjem datoteke strežnik in odjemalec izmenjata kontrolno vsoto (na primer MD5); do dejanskega prenosa datoteke pride le v primeru, da datoteka na ciljni strani ne obstaja ali da imata datoteki različni kontrolni vsoti.

**Primer:** Ukaz copy(»Sort/Sorting.atp«, AM2EM) prenese datoteko amProjectsRoot/Sort/Sorting.atp iz AM v datoteko emProjectsRoot/Sort/Sorting.atp na EM).

#### Ukaz run

**Sintaksa:** run(ime\_skripte, parametri)

**Opis:** Ukaz run požene podano skripto ime\_skripte.

(Opomba: prej izvajanjem ukaza run mora biti skripta s pomočjo ukaza copy prenešena na EM).

Parametri, ki se podajo ob klicu skripte, se predhodno prilagodijo in sicer:

* vse pojavitve $emProjectsRoot se zamenjajo s pravo vrednostijo emProjectsRoot
* vse pojavitve $null se zamenjajo z imenom null naprave na sistemu

**Primer:** ukaz run(“Sort/RunSort.sh”, “$emProjectsRoot/test1 >$null”) se izvede kot

emProjectsRoot/Sort/RunSort.sh emProjectsRoot/test1 >/dev/null

#### Ukaz wait

**Sintaksa:** wait(msec)

**Opis:** Ukaz wait povzroči nedejavnost sistema (stanje IDLE) za msec milisekund.