Project: Genome mapping.
Submitted by: Shadi Ibrahem.
MAGIC's site: http://MAGICmapping.sourceforge.net

Project's Summary:
MAGIC (reorder of Integrative and Accurate Comparative Genome Mapping) is a tool for comparing between genomes. The comparison is done in two phases: a pre-processing phase and a mapping phase. The pre-processing phase calculates a comprehensive table of all similar segments shared between the two genomes. The mapping phase operates on this comprehensive table in an attempt to determine the true evolutionary relations between the different segments in the two genomes. The pre-processing phase is implemented in C++ and integrates external code (for global and local alignments). The mapping phase, on the other hand, is implemented in C++ and has no external dependencies. For more information, we refer the reader to the documentations of the pre-processing and the mapping phases, as well as to the original manuscript (Swidan et. al., PLoS CB, 2006).

Task's Summary:
MAGIC's graphical user interface (GUI) is implemented in Java and provides a user friendly environment for interacting with the underlying C/C++ implementation. The GUI uses eclipse SWT (**Standard Widget Toolkit)** as an external package, since SWT provides a better control over the GUI functionality. To simplify MAGIC's installation, this external package is integrated and distributed with the software. Below we summarize the main features of the GUI, which enable the user to easily use MAGIC for interacting with both the pre-processing and the mapping functionalities. In addition, the GUI provides a friendly view of the results obtained in both phases. It provides as well a "history" mechanism for keeping track of the runs that were performed and of their parameter values. Finally, the GUI enables editing the organisms' database by allowing the addition of new genomes and the deletion of existing ones.

Ease of Use:

1. The GUI presents the history (results of previous runs) as editable (SWT) tables, and enables the user to view and to manipulate them. To facilitate distinguishing between the different runs, the table shows the names of the genomes that were compared, as well as the date, the time, and the parameters used in that comparison.
2. The history is divided into two subdivisions (each presented as a table): A division for the pre-processing runs, and a division for the mapping runs. The information associated with each run (as described in the previous bullet) is given in a separate line. Through the tables, the user can view and manipulate the history results, e.g., by running the mapping phase with different parameters on an existing pre-processing phase's result or by deleting a specific row.
3. The "add genome" form allows the user to add a new genome to the organisms' database by inserting the organism's properties and the

required files (e.g., the organism's genome, the anchor file, etc…). Once the addition is confirmed, the user can use the new organism as input to both phases. Furthermore, a user can remove organisms from the database.

4. MAGIC contains many parameters. To present these in a friendly way, a parameters' tab is supplied. Through this tab, the user can change the parameters' default values; these new values are saved and become the default values in future comparisons. Still, a user can always restore MAGIC's factory settings. Furthermore, the user can change the parameter values for a specific run, without altering the default values. This can be done through the parameter tables that are shown for each specific run (for both pre-processing and mapping phases).
5. MAGIC's setting dialog enables the user to alter MAGIC's input and output paths. It also enables the user to choose his preferred language from a set of available ones.
6. Tool tips are displayed each time a user hovers over visible components in the parameters' tab, the options' dialog and the "add genome" form. The tool tips facilitate the usage of the different buttons and text fields and explain the scientific meaning of related labels.

Implementation:

1. MAGIC's GUI is entirely built using the SWT package. This makes it (almost) platform-independent. Original Java methods were used in order to manage files, integrate C/C++ native methods, and for internationalization (i.e. the support of different languages). This enables MAGIC to run on any Java version starting with 1.3.
2. Java Native Interface (JNI) is used to call the C++ code of both phases: C++ functions in the mapping phase are invoked directly from the Java byte code. To do that, the application code needs to be compiled with the native machine instructions. The same method, however, does not apply to the pre-processing phase, because it integrates external code for the global and local alignments. To overcome this problem, executables (or binaries) of the pre-processing phase were pre-complied for both Windows and Linux platforms. A C++ interface has been provided to invoke these executables as a new process (in other words, the pre-processing executable is considered as a "black box" that is simply invoked by the Java byte code through the C++ interface).
3. A link to MAGIC's website was added to the GUI. This way the user has an easy access to MAGIC's website where he can, e.g., learn about MAGIC or request help through the mailing list.
4. MAGIC's web site was built using HTML integrating DHTML to enable menus. It includes news, screenshots, download page, documentations, FAQ, mailing list, and contact information.
5. MAGIC's pre-compiled binaries are provided for two platforms: Windows and Linux. (This is due to the fact that native methods have to be compiled differently on each platform.)
6. Java threads were used to enhance and facilitate the usage of MAGIC: The user can navigate between tabs processing different runs, compare between them, and review the history at the same time. One should note, though, that MAGIC's processes are "heavy" (both phases require a lot of computational

resources, i.e., a high CPU usage and lots of memory - ~500MB for closely related bacteria). Thus, it is recommended to use this option on high-performance workstations.

Software Design

Design strategy:

MAGIC's GUI is built using the **Model-view-controller (MVC)** design pattern. The main idea behind this pattern is to extract the high level interactions between objects and to reuse this abstraction in other applications.
If constructed correctly, models can be fairly stable (thanks to the stability of the model's domain). The user interface code, on the other hand, usually undergoes frequent and - sometimes - dramatic changes (typically because of usability problems, the need to support growing classes of users, or simply the need to keep the application looking "fresh"). Separating the view from the model makes the model more robust, because the developer is less likely to alter the model while reworking the view.
The information passed from the view package to the model package (e.g., in order to store it) goes through the controller. This structure provides the following benefits:

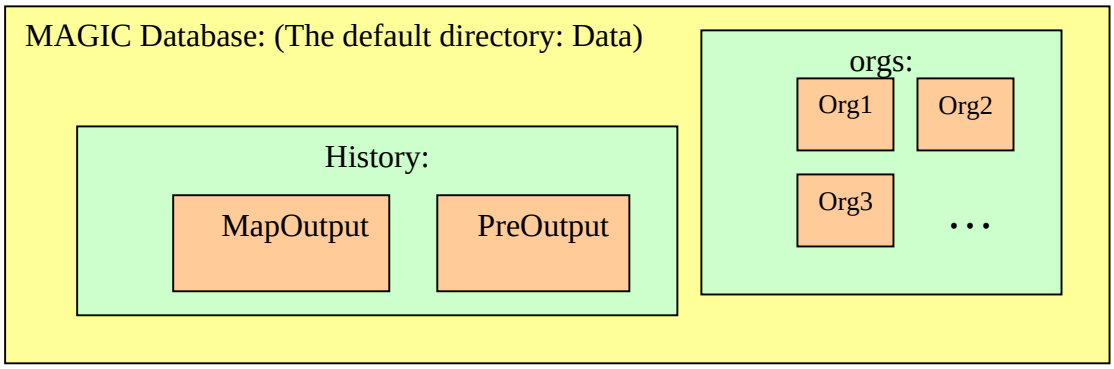
1. The controller contains a set of methods used as a "gateway" to the model and vice versa. These methods, and by verifying the correctness of the data, insure its reliability.
2. Updating a controller method, reflects immediately on each call used in the view package.
3. Verifying software correctness and performing regression tests become less tedious: When a set of stub methods are defined and stored in a workplace, one needs only to replace controller methods (gateway methods) in order to discover any lacking compatibility with excepted results which are derived from the view package.

MAGIC's Database:

MAGIC required information is stored in the "Data" subdirectory at the root of the application's directory. This subdirectory includes the file "Magic.ini". This file contains MAGIC's default parameters: the path to the input (organisms) directory, the path to the output (history) directory, and the GUI's language.
The data directory contains also two subdirectories:

1. orgs: include the organism's hierarchy (in case the default input path is chosen).
2. History: holds the pre-processing and mapping phase history in separated directories.

Each one of "orgs" and "History" subdirectories resides under the Data directory by default. The user can change their path, though, through the GUI.



The Controller package:

This package includes the main class that creates the logical view and the model. Through it, the logical view can modify or import information from the database.

Interface IDataHandler:
This interface provides a set of methods that are used by most of the view package classes. These methods are called to update and to store information in the model, e.g., genome input files, languages, parameters etc…

Class DataHandler:
The controller of MAGIC: It implements the methods declared in the interface IDataHandler. It has also provides access to the functionalities of the FileCopy class.

Class FileCopy:
It provides a single method for copying files. One needs simply to supply the destination and the source paths of the file.

The Model package:

Includes three main classes for storing application related information (MAGIC's default parameter values, the input directory hierarchical information, and imported history data from pre-processing or mapping output files).

Interface IDataManager:
This interface is used by the controller and enables it to access the model's database, e.g., to import output files into the appropriate data structures.

Class DataManager:
This class implements the methods declared in the IDataManager interface.

Class ImportedTableFile:
This class imports output file lines in order for the view to present them as a chart. In order for this class to handle a file, the file should have a specific format. The results of both phases (pre-processing and mapping) share this format. The exact description of a single line is given below.
In addition, this class handles “circular” lines (i.e. lines with a start coordinate smaller than the end coordinate) that might occur in circular genomes (details below).

Assumed line format: a tab delimited line containing these 8 fields.

First Genome.			Second Genome.				
Segment start	Segment end	Segment length	Segment start	Segment end	Segment length	Orientation	Identifier

While positive oriented lines are drawn with positive slope, negative oriented lines are drawn with negative slope. Circular lines (as defined above) are cut into either 2 or 3 sub lines in order to draw them. These lines may arise in circular genomes (e.g., bacteria and archaea). To draw a circular line with a positive orientation one needs to cut it into either 2 or 3 sub lines: One line begins from the start coordinates and ascends until it reaches one of the end chart boundaries, i.e., either the upper or the right end boundary. The second line forms a continuation of the first line by beginning at the parallel start boundary of the end boundary that the first line reached, i.e., the left boundary if the first line terminated at the right end boundary or the bottom boundary if the first line terminated at the upper end boundary. The second line continues with the same slop, until it reaches either the end coordinates or the other end boundary. In that case a third line forms a continuation to the second line as discussed above.
A similar procedure is applied to lines with a negative orientation.

Class MAGIC:
This class stores the application’s default parameter values, the workspace directory paths (input and output directory paths), and the language set by the user. The DataView class uses this class in order to present the logical view accordingly (importing genome names from the organisms directory, displaying the text according to the chosen language, and setting the default parameter values).

Class Line:
This class stores the coordinates and the sign of a single line. The coordinates are defined by the line’s start point (x1,y1) and its end point (x2,y2). The sign can be either “+” or “-“.

View Package:

This package includes the classes that handle the logical view and the interaction with the user. These classes retrieve information from the database and draw the relevant view.

Class DataView:
The main logical view: It creates the initial view, the main window including the main tab, the menu, and it presents genome names according to the information stored in the database. In addition, it creates instances of the classes that handle most of the logical view information and passes the control to them.

Class AddGenome:
This class opens a tab with an empty "add genome" form. In addition, it passes the added genome information to the controller to store it in the database.

Class RemoveGenome:
This class removes the selected organisms (and their associated files). It also removes their relevant information from the organism's database (through the controller).

Class History:
This class opens a new tab including information presented in the form of tables. This information includes the history (output results) of previously performed runs (pre-processing and mapping). It also shows the used parameter values in these runs, the time and date of the run, and it can create additional tabs showing the results of previous runs.

Class GRunTab:
This class is used for three purposes:

1. Presenting the results of previous runs. The presentation includes a chart, statistics presented in a table, and the parameter values.
2. Running a new pre-processing phase on two selected genomes followed (optionally) by the mapping phase and showing the results of the run as mentioned above.
3. Running the mapping phase on a pre-calculated pre-processing result (this form can be invoked only from the history tab by clicking on the "Run mapping" button").

Class ParametersFrame:
This class is used by the DataView to create a single parameter tab (trying to open the parameter tab when it is already opened would simply bring it into focus). The parameter values are retrieved from the database (as last set by the user). In case the user chooses to change those parameters, the new values are saved as the default parameters in the database. Still, the "factory" values of all parameters can be restored by pressing the "Restore defaults" button.

Class SWTTable:
A class that handles creating and manipulating a single SWT table. It includes a set of methods that enables setting up a modifiable table, and performing insertions, deletions, as well as other commonly used operations on it.
This class is used by the History class to render the history tables, and by GRunTab to render the parameter values and statistics.

Class PaintChart:
A class that creates a single canvas chart: It inserts lines into the canvas and draws chart scale. It is used by GRunTab to view pre-processing and mapping results in the form of a chart.

Class OptionDialog:
The OptionDialog class enables the user to change MAGIC's settings.
It creates a new frame with two sub-tabs: one for setting MAGIC's input and output directories, and one for setting MAGIC's default language.

Class Help:
The Help class opens a new frame with a link to MAGIC's site (e.g., to enable the user to post help requests or to view general information).
It also displays the "About" information and links to the project developers page at MAGIC's site.

Cmapping package:

This package includes a single class for invoking the C++ mapping code.

Class Native:
This class serves as an interface between the Java code and the C++ code; it invokes the C++ "main" method, and passes to it the needed parameters.

Cpreprocessing package:

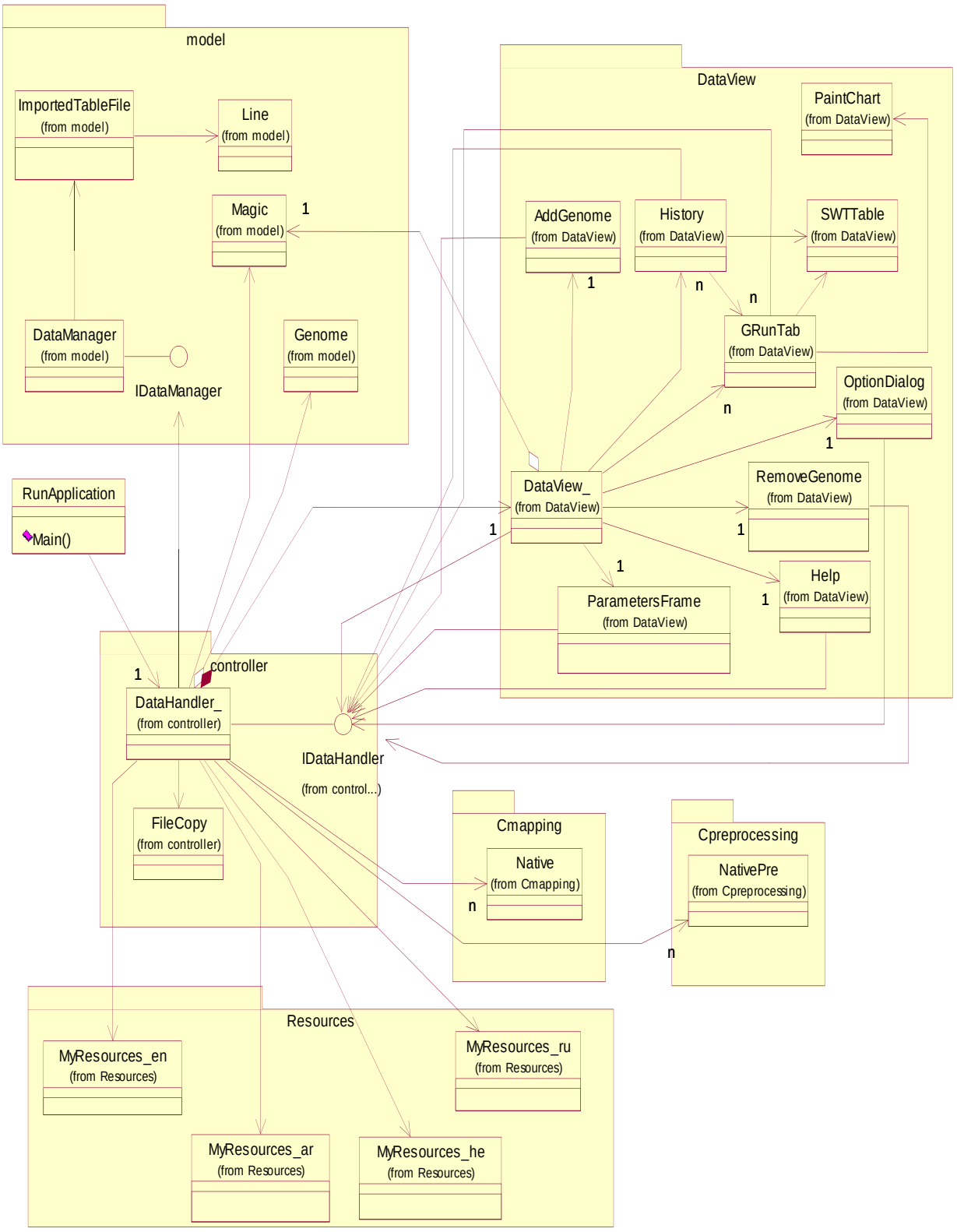
This package includes a single class for invoking the C++ pre-processing code.

Class PreNative:
This class serves as an interface between the Java code and the C++ interface; it invokes the C++ method, and passes to it the needed arguments. The C++ interface executes the pre-processing executable and passes to it the needed arguments.

Resources package:

Contains classes in the form of Key-Value tables for internationalization purposes. In the meanwhile, four languages are supported: Arabic, English, Hebrew, and Russian.

Class diagram:



Guideline to the Developer:

1. In java, one cannot control threads that are running outside the current virtual machine. However, in order for the user to terminate a running process, e.g., a pre-processing phase run or a mapping phase run, (e.g., through a "cancel" button) one needs to integrate a C++ code that implements terminating a thread or a process when given the process' ID (PID). For that reason, the PID for the pre-processing process is identified and stored in the GRunTab class. On the other hand, in order to terminate a mapping phase thread (which resides inside the Java virtual machine) a set of methods provided only in the new versions of Java (1.5.0 and above) is required.
2. Deleting files in Java requires having a writing access. Thus, in order to delete a file, one has to acquire first the writing access (e.g., by opening the file for writing and immediately closing it). Only after that one can delete the file. The submitted code includes controller method "deleteFileOrDirectory" for deleting directories recursively.
3. Native methods can be stored in one package. However, in order to do that without mixing up C/C++ files of both phases (pre-processing phase and mapping phase), the compilation should be done in separate directories, and binary output files (.obj/.dll for Windows and .o/.so for Linux) should be stored in the same library path (under the application's root directory). Java allows only a single library path and expects to find there the required binary files for executing native methods. For the JVM (Java Virtual Machine) to locate those binary files, one should use the following argument when calling the JVM (through the "Java" command-line):

   "-Djava.library.path=<The directory path>"

   This directory should also include the swt.jar file (for both Linux and Windows).

   Note that in Linux, native system library files (.o and .so) must be located in the library path; if one does not set a path, the JVM chooses by default the application's root directory.

4. When, updating the pre-processing code, one should only replace the existing "pre-pro.exe" file with the new one (no re-compilation is needed). In case of changing the way arguments are passed to the executable "pre-pro.exe", one needs to update and compile the interface "preprocessingPhase.cpp" only. In addition, the binary files should be in the library path directory.