T-76.115 Software Project
RevRatio
Technical Specification

Group Duckpond

Table of Contents

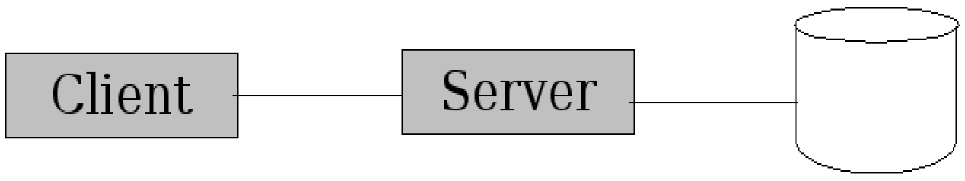
1. Introduction

The purpose of this document is to describe the architecture and design of the RevRatio software. It is mainly written for open source developers who wish to further develop the software.

It is assumed that the reader is familiar with the main domain concepts described in the requirements specification, in order to better understand this document.

2. Architecture Overview

The software architecture is based on a RMI client-server model, where the server is connected to a database. The reason for using a client-server model is based on the requirement that the architecture should support teams (see requirement F4 in the requirements documentation).



An embedded hsqldb database is used by default, but it is possible to connect other databases, that are usable with JDBC, by altering the configuration (however, at the moment only MySQL has been confirmed to work with the software).

RevRatio is physically composed of two separate programs: RRServer and Revratio.
Revratio works simultaneously as a server and a client, so that internally there is no RMI connection, but it is still possible for external clients to connect via RMI.
RRServer contains, as its name implies, only the server functionality and it is useful in cases where someone wants to run a dedicated server, that other clients can connect to.

See Appendix A for links to more information about the mentioned technologies.

2.1. Analysis

The purpose of this chapter is to analyze some major design decisions and explain why things have been done a certain way.

2.1.1. Client-Server

The reason for using a client-server model is based on requirement F4 (architecture must support teams) in the requirements documentation. The design team didn't know of any real alternatives to this. Without any remote connections, all review data would have to somehow be sent manually with files within a team (which would be cumbersome and not very user friendly) , but keeping the data synchronized would then be very difficult. If there is a centralized server we can simply lock the data and ensure that only one user at a time can modify it.

2.1.2. RMI

Two options for handling the communication between the server and the client were regarded: RMI or sockets.

The general consensus was that we would win more time by using RMI: The team had experience in using RMI (resulting in a very small learning curve) and we thought that having to parse socket-based commands would be more tedious to implement.

The drawback with using RMI is that it likely results in more communication overhead, compared to a socket-based solution.

2.1.3. Hsqldb

The two basic options regarded for data storage were to either use ordinary files or a database.

When it came to files we saw two options: Serializing Java objects to files or defining our own file format. Due to time constraints the latter option was quickly discarded.

Advantages and disadvantages of using a database:

- Atomic transactions are automatically ensured, that is simultaneous actions of several users won't accidentally corrupt the data.
- Specific pieces of data can be easily retrieved. So there no need to read a whole file.
- Using a database for this kind of an application that most of the time is used by a single user (as opposed to a team) goes against what a user might expect, that is having the ability to save in a file.
- Requires some extra effort in keeping the program user friendly, so that the underlying database is hidden from an ordinary user.
- Difficult to move data to another computer.

Advantages and disadvantages of serialized files:

- With Java it would be easy to store files by serializing them.
- Would require a lot more effort to ensure atomic transactions, in the case of several users storing
- If a class is changed, so is the file format. The issue of having to address compatibility between different versions would be very cumbersome.

When taking into consideration the requirement for team support, using a database seemed as the better and less riskier solution as a database management system automatically takes care of some difficult problems that would otherwise have to be solved.

There were some requirements that affected our choice of a database:

C2: “The program should in theory be platform independent, but we must verify that it works as intended on at least Windows and Linux”

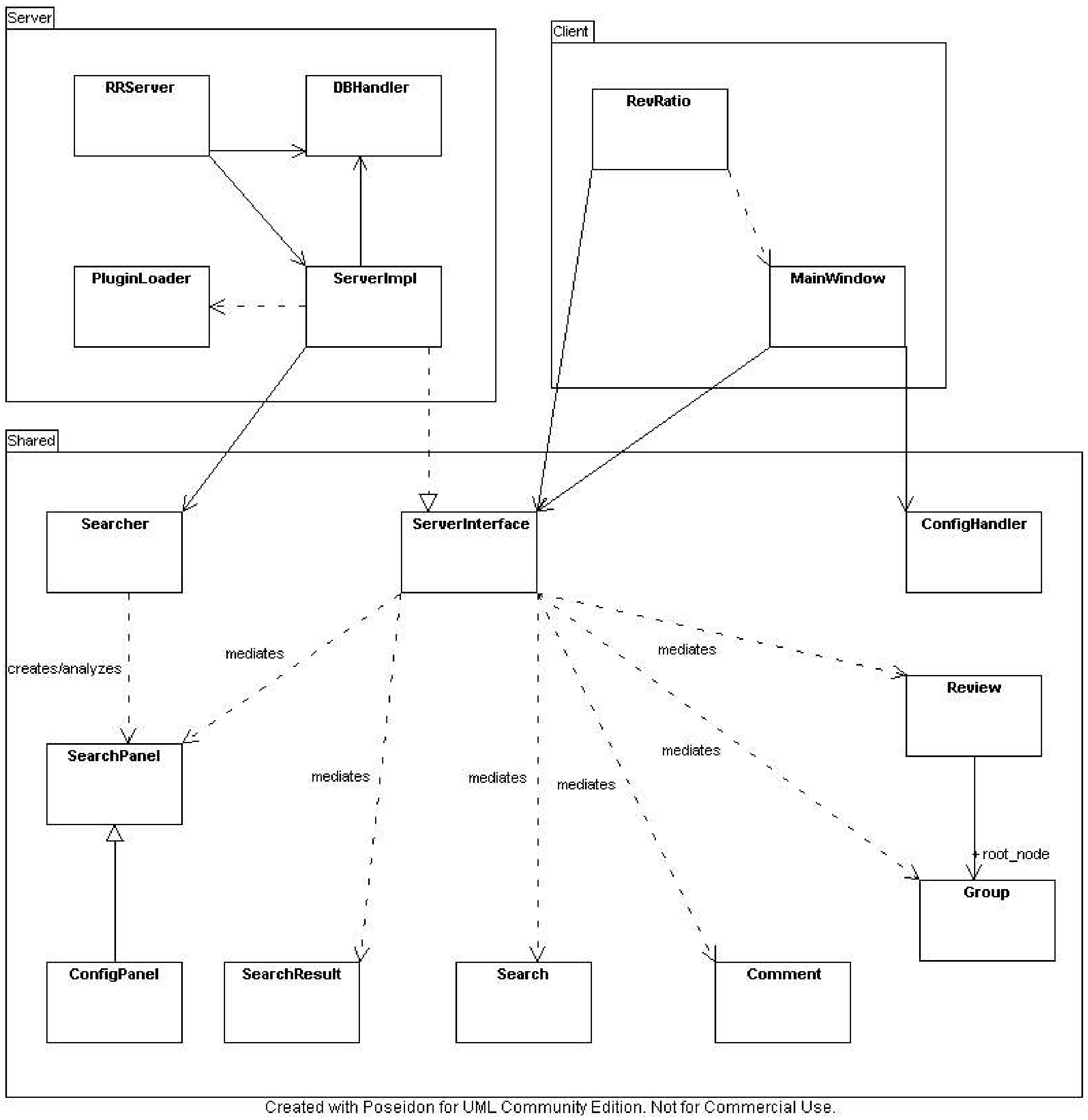
NF2: “User interface that is easy to use and learn”

NF2 implicated in our opinion that a user should not have to bother with setting up a database for the system. So there needs to be a database that “just works” automatically under the hood, so it would have to be embedded into the system. Hsqldb seemed to suit our needs well because it is open source, Java based and it can be included in the system as a jar-package. Because Hsqldb is Java based, platform independency is also ensured.

2. Class Diagrams

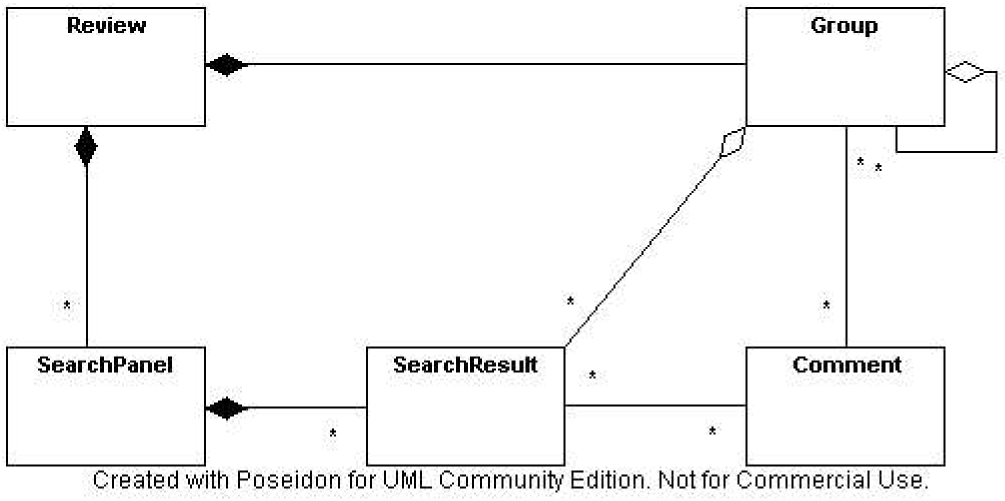
2.1 Overview

This class diagram shows an overview of the classes that belong to the system. Many classes contained in the client package have been omitted and are shown in in another diagram. All plugin classes are also not shown.



Created with Poseidon for UML Community Edition. Not for Commercial Use.

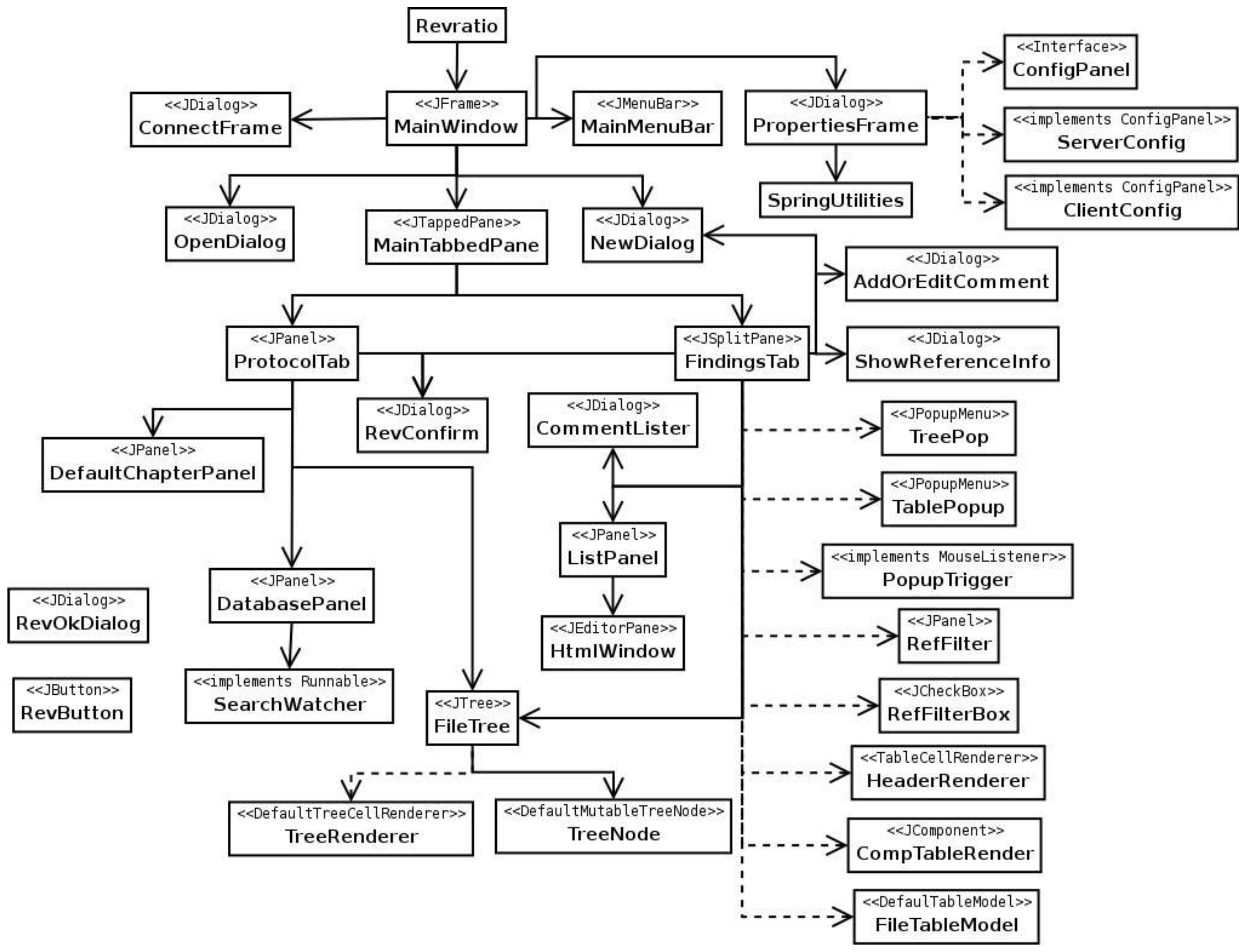
2.2 Data-oriented Classes


Review
Group
SearchPanel
SearchResult
Comment
*
*
*
*
*
*
* *
Created with Poseidon for UML Community Edition. Not for Commercial Use.

This class diagram shows how the *Review, SearchPanel, SearchResult Group* and *Comment* classes are logically associated to each other. Note that they are physically connected only in the database. In the code the associations are implemented trough id:s that are used as arguments in several *ServerInterface* and *DBHandler* methods (see Javadocs for details) .

2.3 User Interface Classes

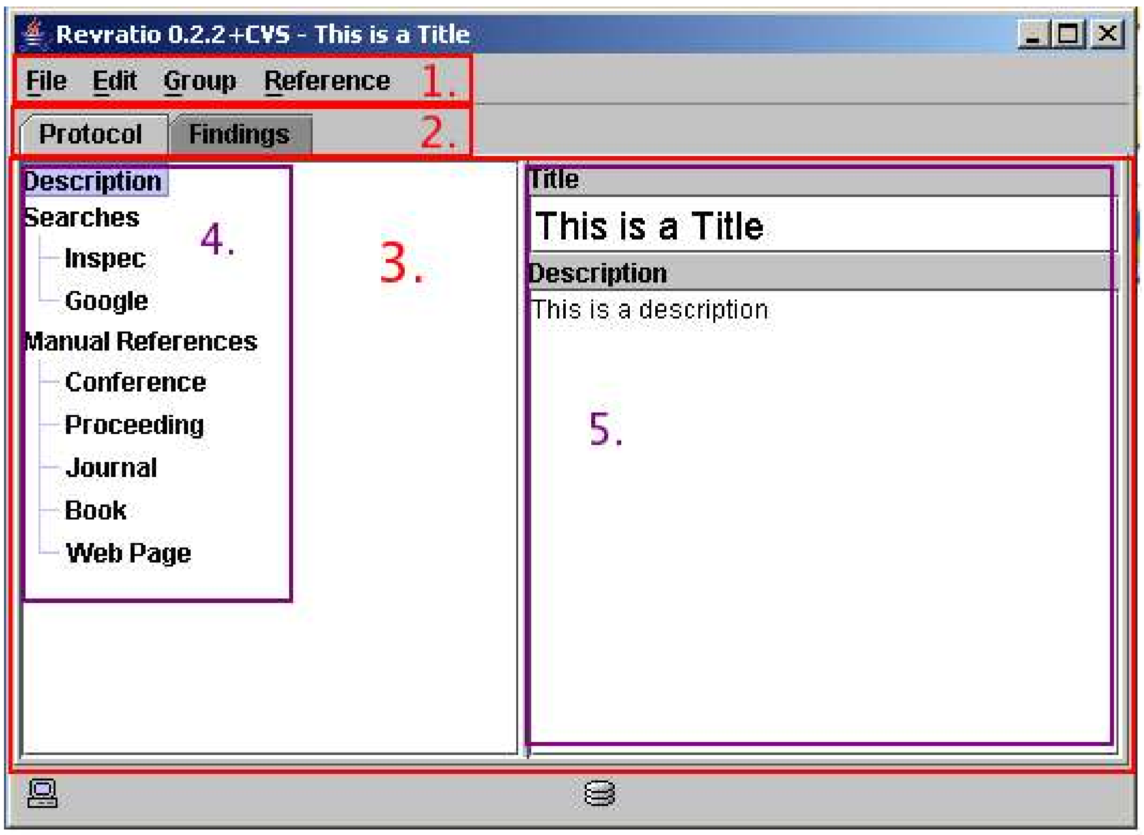
This class diagram is conceptual, and the associations don't directly map to the physical relationships in the code. This diagram doesn't fully comply with standard UML-notation. Here a one way association (arrow) between class A and B, pointing to B, means that B is a part of A. For example, in the GUI the *DatabasePanel* is a part of *ProtocolTab* that is a part of *MainTabbedPane* and so on. Inner classes are denoted with dotted lines. The *MainWindow* class is a good starting point for reading the diagram. More information about the user interface can be found in section 3.

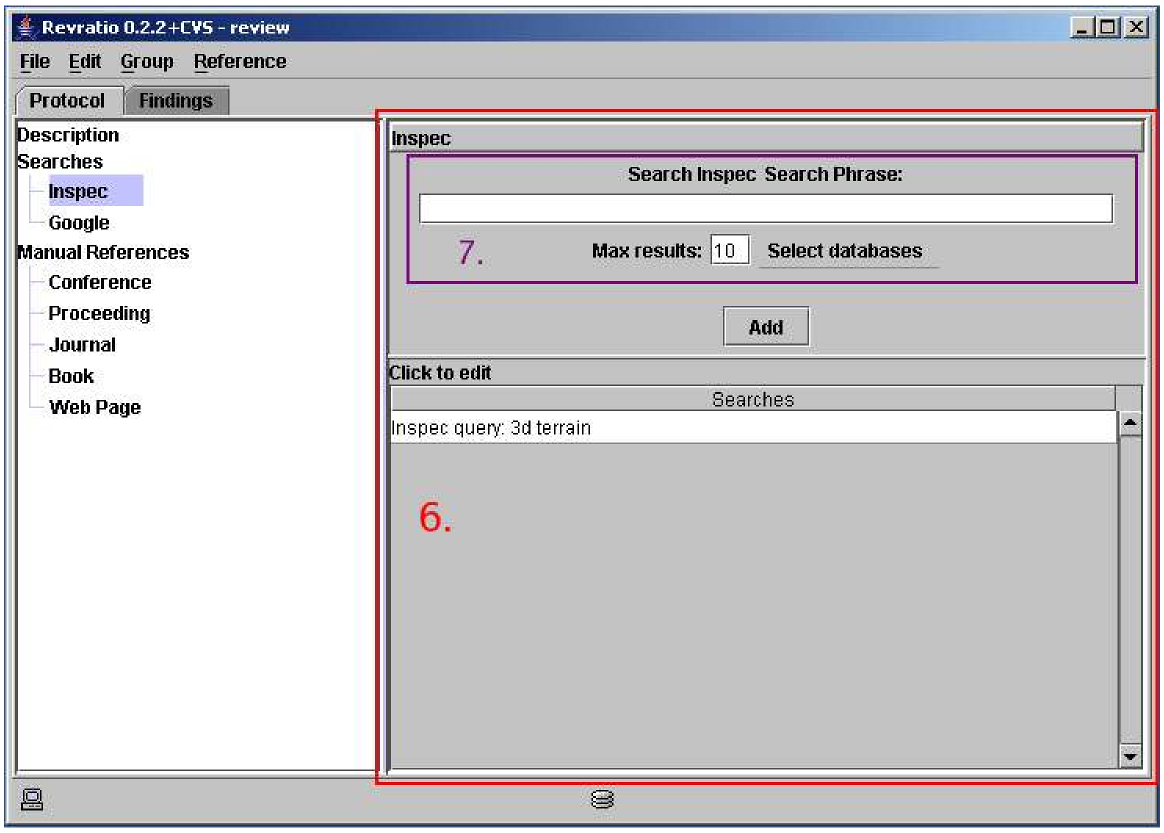

Revratio
<<JDialog>> ConnectFrame
<<JFrame>> MainWindow
<<JMenuBar>> MainMenuBar
<<JDialog>> PropertiesFrame
<<Interface>> ConfigPanel
<<implements ConfigPanel>> ServerConfig
<<implements ConfigPanel>> ClientConfig
SpringUtilities
<<JDialog>> OpenDialog
<<JTappedPane>> MainTabbedPane
<<JDialog>> NewDialog
<<JDialog>> AddOrEditComment
<<JPanel>> ProtocolTab
<<JSplitPane>> FindingsTab
<<JDialog>> ShowReferenceInfo
<<JDialog>> RevConfirm
<<JDialog>> CommentLister
<<JPopupMenu>> TreePop
<<JPanel>> DefaultChapterPanel
<<JPopupMenu>> TablePopup
<<JPanel>> ListPanel
<<implements MouseListener>> PopupTrigger
<<JDialog>> RevOkDialog
<<JPanel>> DatabasePanel
<<JEditorPane>> HtmlWindow
<<JPanel>> RefFilter
<<JButton>> RevButton
<<implements Runnable>> SearchWatcher
<<JTree>> FileTree
<<JCheckBox>> RefFilterBox
<<TableCellRenderer>> HeaderRenderer
<<DefaultTreeCellRenderer>> TreeRenderer
<<DefaultMutableTreeNode>> TreeNode
<<JComponent>> CompTableRender
<<DefaulTableModel>> FileTableModel

3. Client / User Interface

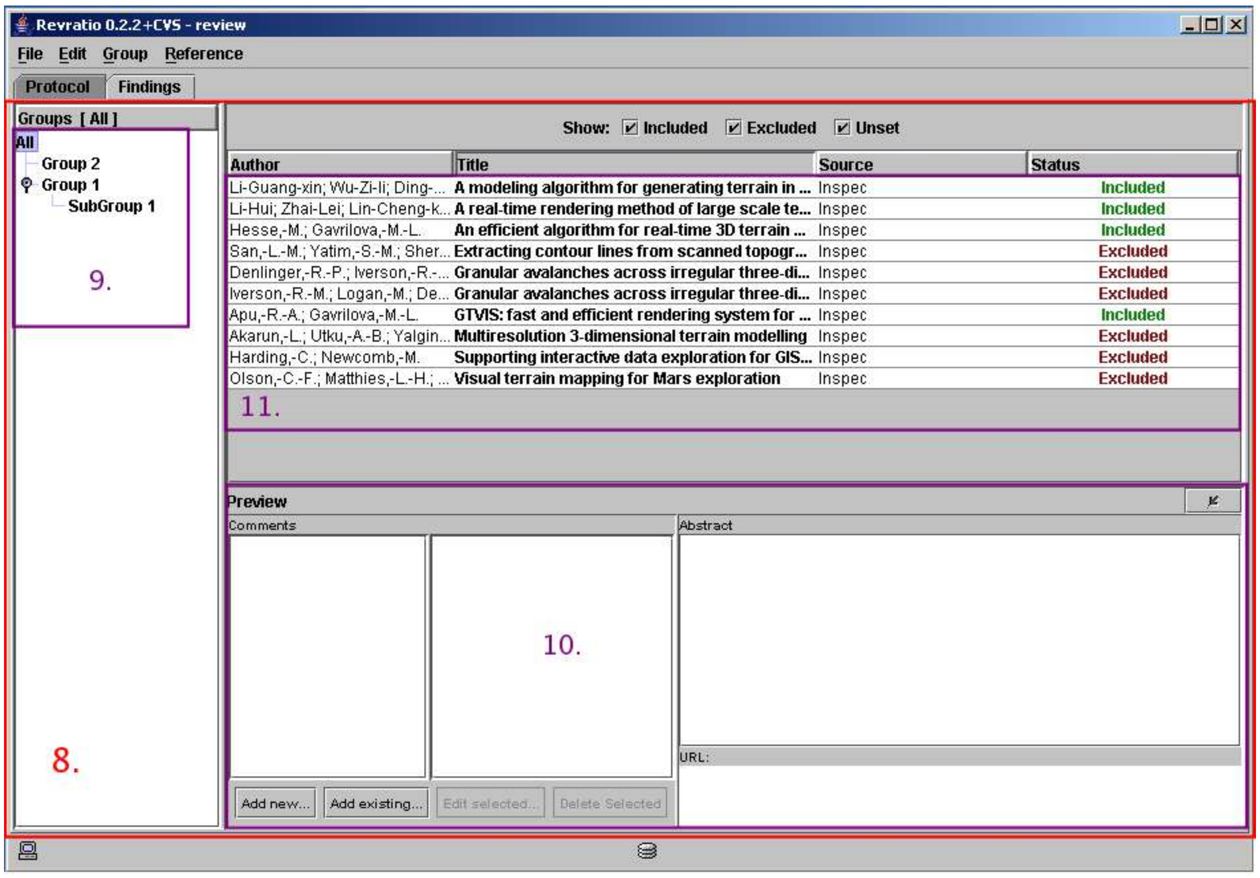
The words “client” and “User Interface” are almost interchangeable when talking about the design of RevRatio because the client is thin and almost all classes in the client are only used to handle the user interface. The amount of classes in the client-package however, is pretty big and all class names are not necessarily evident. As a result the diagram in section 2.3 may be a bit difficult to read.

A rule of thumb is that most classes map directly to a part of the user interface that can be seen by the user. Often a picture conveys more than a thousand words, so below are some edited screenshots with numbered areas, taken from RevRatio. The purpose is to give an overview of what is implemented where. Each numbered area maps to the corresponding class, that handles the indicated part of the user interface. More detailed information can be found in the javadocs.





Inspec query: 3d terrain



Show: [✓ Included] [✓ Excluded] [✓ Unset]

Author	Title	Source	Status
Li-Guang-xin; Wu-Zi-li; Ding-...	A modeling algorithm for generating terrain in ...	Inspec	Included
Li-Hui; Zhai-Lei; Lin-Cheng-k...	A real-time rendering method of large scale te...	Inspec	Included
Hesse,-M.; Gavrilova,-M.-L.	An efficient algorithm for real-time 3D terrain ...	Inspec	Included
San,-L.-M.; Yatim,-S.-M.; Sher...	Extracting contour lines from scanned topogr...	Inspec	Excluded
Denlinger,-R.-P.; Iverson,-R.-...	Granular avalanches across irregular three-di...	Inspec	Excluded
Iverson,-R.-M.; Logan,-M.; De...	Granular avalanches across irregular three-di...	Inspec	Excluded
Apu,-R.-A.; Gavrilova,-M.-L.	GTVIS: fast and efficient rendering system for ...	Inspec	Included
Akarun,-L.; Utku,-A.-B.; Yalgin...	Multiresolution 3-dimensional terrain modelling	Inspec	Excluded
Harding,-C.; Newcomb,-M.	Supporting interactive data exploration for GIS...	Inspec	Excluded
Olson,-C.-F.; Matthies,-L.-H.; ...	Visual terrain mapping for Mars exploration	Inspec	Excluded

1: *MainMenuBar*

2: *MainTabbedPane*

3: *ProtocolTab*

4: *FileTree* (contained in a *JscrollPane*)

5: *DefaultChapterPanel*

6: *DatabasePanel*

7: *SearchPanel* (Contained in a *SearchTable*, which is an inner class in *ProtocolTab*)

8: *FindingsTab*

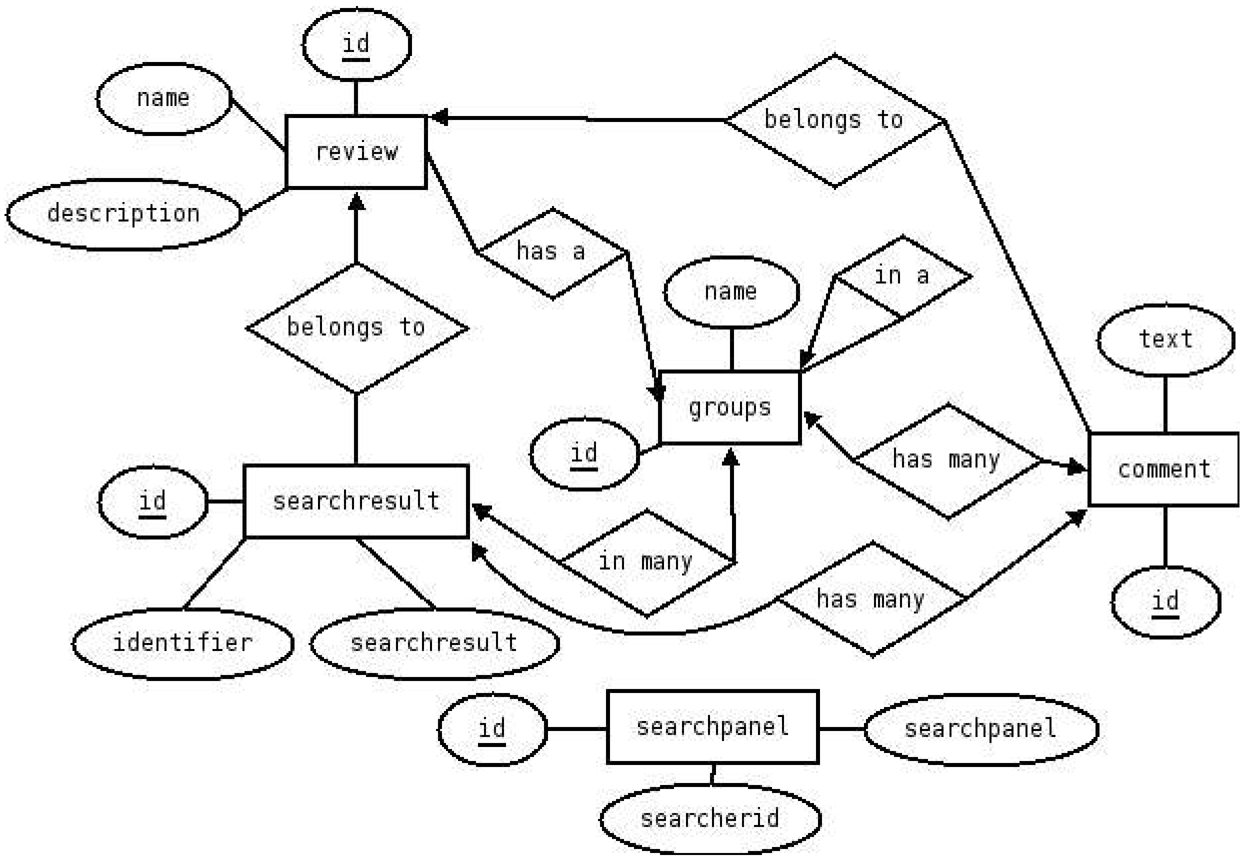
9: *FileTree* (contained in a *JScrollPane*)

10: *ListPanel*

11: *JTable*

4. Database

4.1. E/R Diagram



4.2. Table Descriptions

Here follows a short description of the SQL tables used. These are the MySQL representations. The internal (hsqldb) tables differ only in some details; not having any auto increment feature naturally removes the auto_increment options, text fields are of type “longvarchar” etc.

It is important to keep in mind that not all databases support foreign key constraints, and therefore the DBHandler class must take care of these restraints itself.

comment

This table stores the comments. A comment is always linked to a specific review, so it can be used in both group comments and reference comments.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
id	bigint(20)		No		auto_increment
review_id	bigint(20)		Yes	*NULL*	
text	text		Yes	*NULL*	

commentrelations

This table simply links comments to references and groups. If *isgroup* is **true**, the comment is linked to the group with id *target_id*, else it is linked to the reference with id *target_id*.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
commentid	bigint(20)		No	0	
targetid	bigint(20)		No	0	
isgroup	enum('true', 'false')		No	true	

group

This table builds up the group “tree”. Every group has a parent, except for the “root group” in a review, where parent is null.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
id	bigint(20)		No		auto_increment
parent	bigint(20)		Yes	*NULL*	
name	varchar(255)		Yes	*NULL*	

refrelations

This table links references with the searchpanel they resulted from.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
refid	bigint(20)		No	0	
searchid	bigint(20)		No	0	

resulttree

This table links references into groups. Every reference must be in the review group “All”, and can be in any number of other groups (in the same review).

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
res_id	bigint(20)		No	0	
group_id	bigint(20)		No	0	

review

All review are listed here. The root group (in table **group**), review name and description are stored in this table.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
id	bigint(20)		No		auto_increment
rootgroup	bigint(20)		Yes	*NULL*	
name	varchar(50)		Yes	*NULL*	
description	text		Yes	*NULL*	

searchpanel

All searchpanels (the searches and manual references) that are part of review_id are stored in this table. Searcher_id is usually the same as the plugin name.

Searchpanel stores the xml encoded panel object (because of problems with serialized swing components).

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
id	bigint(20)		No		auto_increment
review_id	bigint(20)		Yes	*NULL*	
searcherid	varchar(255)		Yes	*NULL*	
searchpanel	text		Yes	*NULL*	

searchresult

The searchresults that resulted from the searchpanel.

The searchresult itself is stored as a hashtable.

Identifier is a md5 hash of some of the fields in the searchresult, whereby this particular result can be identified as a duplicate if another result with the same hash is found.

Field	*Type*	*Attributes*	*Null*	*Default*	*Extra*
id	bigint(20)		No		auto_increment
review_id	bigint(20)		Yes	*NULL*	
searchresult	blob	BINARY	Yes	*NULL*	
identifier	text		Yes	*NULL*	

5. Class Descriptions

This chapter contains short descriptions of classes. Detailed information about which methods and attributes the classes contain, can be found in the Javadocs (see chapter 9, References).

5.1. The Server Package

Class Name: RRServer
SuperClasses:
SubClasses:
Collaborators: ServerImpl
Description/Responsibilities: Contains entry point for standalone server. Instantiates and sets up a ServerImpl object

Class Name: ServerImpl
SuperClasses: ServerInterface
SubClasses:
Collaborators: DBHandler, Revratio, MainWindow, Searcher
Description/Responsibilities: Implements the communication between client and database.

Class Name: DBHandler
SuperClasses:
SubClasses:
Collaborators: ServerImpl, Review, SearchPanel, SearchResult, Comment, Group
Description/Responsibilities: Connects to database Stores Review, SearchPanel, SearchResult, Comment and Group objects Loads Review, SearchPanel, SearchResult, Comment and Group objects Updates Review, SearchPanel, SearchResult, Comment and Group objects

Class Name: PluginLoader
SuperClasses: ClassLoader (from Java API)
SubClasses:
Collaborators: ServerImpl
Description/Responsibilities: Loads plugin files

5.2. The Shared Package

This package contains the classes that are needed both for the client and the server.

Interface: ServerInterface
SuperClasses:
SubClasses: ServerImpl
Collaborators:
Description/Responsibilities: Defines the interface for handling the communication between server and client. Stores Review, SearchPanel, SearchResult, Comment and Group objects to database Loads Review, SearchPanel, SearchResult, Comment and Group objects from to database Updates Review, SearchPanel, SearchResult, Comment and Group objects in database Performs searches Retrieves list of plugins

Class Name: Searcher
SuperClasses:
SubClasses: GoogleSearcher, InspecSearcher, ManualBook, ManualJournal
Collaborators: ServerImpl, SearchPanel, SearchResults, ConfigPanel
Description/Responsibilities: Parent class for plugin classes Creates SearchPanel objects Handles searches Analyzes SearchPanel objects (before a search) Creates SearchPanel objecs (after a search)

Class Name: SearchPanel
SuperClasses:
SubClasses: ConfigPanel
Collaborators: ServerImpl, MainWindow, Searcher
Description/Responsibilities: Implicitly defines a search Containes necessary UI-elements for typing in a search or for adding a manual reference

Class Name: Review
SuperClasses:
SubClasses:
Collaborators: DBHandler, MainWindow, ServerImpl
Description/Responsibilities:
Holds Title of a review
Holds Desrciption of a review
Holds root node to tree of Group objects

Class Name: Search
SuperClasses:
SubClasses:
Collaborators: ServerImpl, MainWindow
Description/Responsibilities:
Holds textual description of a search without graphical elements as in SearchPanel

Class Name: SearchResult
SuperClasses:
SubClasses:
Collaborators: DBHandler, Searcher, ServerImpl, MainWindow
Description/Responsibilities:
Holds information about one search hit.

Class Name: Group
SuperClasses:
SubClasses:
Collaborators: DBHandler, ServerImpl, MainWindow
Description/Responsibilities:
Holds information of a group

Class Name: Comment
SuperClasses:
SubClasses:
Collaborators:
Description/Responsibilities:
Holds a comment added by user

Class Name: ConfigHandler
SuperClasses:
SubClasses:
Collaborators: MainWindow
Description/Responsibilities: Read configuration file Parse configuration file Saving configuration information Ensuring correctness of configuration information

5.3 The Client Package

Class Name: Revratio
SuperClasses:
SubClasses:
Collaborators: MainWindow
Description/Responsibilities: Contains the entry point for Revratio program Instantiates a MainWindow object

Class Name: MainWindow
SuperClasses: JFrame
SubClasses:
Collaborators: ServerImpl, ConnectFrame, MainMenuBar, PropertiesFrame, MainTabbedPane
Description/Responsibilities: Holds all UI-related objects Establishes connection to server

Class Name: ConnectFrame
SuperClasses: JDialog
SubClasses:
Collaborators:
Description/Responsibilities: Handles RMI connection

Class Name: MainMenuBar
SuperClasses: JMenuBar
SubClasses:
Collaborators:
Description/Responsibilities: Functions as the menu bar of the main window

Class Name: PropertiesFrame
SuperClasses: JDialog, ActionListener
SubClasses:
Collaborators: ConfigPanel
Description/Responsibilities: Holds ConfigPanels

Class Name: MainTabbedPane
SuperClasses: JTabbedPane
SubClasses:
Collaborators:
Description/Responsibilities: Defines the main layout of the tabbed panel

Class Name: ProtocolTab
SuperClasses: JSplitPane
SubClasses:
Collaborators:
Description/Responsibilities: Holds the main layout for the protocol of a review

Class Name: FindingsTab
SuperClasses: JSplitPane
SubClasses:
Collaborators:
Description/Responsibilities: Defines main layout for displaying search results

6. Configuration

The configuration of the system is stored in a text file named revratio.conf, under the users home directory. The exact location of this file depends on the operative system:

Windows: Document and Settings/<username>/Application Data/RevRatio/revratio.conf

Mac OS X: /Users/<username>/Library/Application Support/RevRatio/revratio.conf

Unix variants: ~/.revratio/revratio.conf

The configuration can be altered either by editing the file manually or through the user interface. The configuration file is handled by *ConfigHandler* . Configuration items are accessed internally through a hashtable that contains strings. *ConfigHandler* checks automatically that data put into it is in the correct format.

7. Plugins

As described in requirements F4 and F5 in the requirements documentation, the user should be able to search databases from within the program and the program should be extendable so that support for new types of searches can be added. Plugins are also used for the implementation of manually inserted references.

This extendibility is done with a plugin solution. The plugins are loaded in the server, so that if a team is working on the same review, everyone is assured to have the same set of plugins.

7.1. How Plugins Work

Plugins consist of one or several compiled Java classes, that is .class files.

The class (or one and only one of the classes if there are several) should extend the Searcher class.

The plugins should be placed in folder named "plugins", under the folder that contains the RevRatio program.

The plugins are loaded dynamically in the server, and a list of supported databases is created and eventually displayed to the user. This way plugins can be compiled and distributed separately without having to make a new version of the program each time support for a new database is added.

7.2. Instructions for Making Plugins

The first step for making a new plugin is to create a Java class that extends the class *Searcher*. The abstract methods (from the *Searcher* class) that should be implemented are *search*, *createPanel, getDescription, getName* and *isConfigurable.*

So a skeleton implementation should look something like this:

Example 6.2

```
class MySearcher extends Searcher {
        MySearcher() {

        }
        public int search(int reviewId, SearchPanel searchingPanel) {
        …
        }
        public SearchPanel createPanel() {
        …
        }
        public String getDescription(SearchPanel panel) {
        …
        }
        public String getName() {
        …
        }
        public boolean isConfigurable() {
        ...
        }
}
```

7.2.1. The *createPanel* method

This method should create a *SearchPanel* object and return it. This object defines how the user interface displays the fields required to type in a search. So a simple SearchPanel could contain just a textbox. The code could look something like this:

```
Example 7.2.1

public SearchPanel createPanel() {
        SearchPanel myPanel = new SearchPanel(this.id,"MySearcher");
        JTextField textfield = new JTextField(40);
        myPanel.add(textfield, “element1”));
        return myPanel;
}
```

In the last line the second argument is optional. But in this case it will be possible to easily get the textfield from the SearchPanel object by calling: myPanel.getComponent(“element1”).

In the first line one of the arguments used is *this.id.* You don’t have to worry about this, because the id is set by the *ServerImpl* class when the plugins are loaded.

7.2.2 The *search* method

When a user performs a search, the search method of the corresponding plugin is eventually called. The search method is then responsible for performing the actual search and return the results to the server. So if we continue from the example given in example 7.2.1 a skeleton for the *search* method would look like this:

```
Example 7.2.2 a

public int search(int reviewId, SearchPanel searchingPanel) {
        JTextField textfield = searhingPanel.getComponent(“element1”);
        SearchResult result[];
        //Here you need to analyze textfield and implement the search. Then initialize and store the results of the
        //search in the result array
        return server.storeResults(reviewId, result);
}
```

Here you don’t have to worry about the *server* variable. The example above didn’t show how to create *SearchResult* objects. There are several ways to to this, but the preferred way is to call the constructor like this:

SearchResult searchResult = new SearchResult(this.id)

The next step is to define what information is shown to a user when a search result is displayed. This is done with the help of setTag –method. The method takes two strings as its arguments. The first string is a descriptive field and the second string contains the actual information. An identifier for the search result should also be created. So if a search result should contain the title of an article, and an abstract, the code could look like this:

```
Example 7.2.2 b
SearchResult searchResult = new SearchResult(this.id)
searchResult.setTag(“Title”, “This is a Title”);
searchResullt.setTag(“Abstract”, “This is an Abstract”);
```

You can add whatever tags you like, but it’s recommended that a convention is followed, where the search result contains at least a title and an abstract or description. The tags are shown to the user in the same order they were added.

The last line creates an identifier for the search result that is used for comparisons for duplicate removals (see 7.2.7 for more information).

7.2.3 The *getName* method

This method should just return a short descriptive string for a user. For example, a plugin that uses the Google search engine simply returns "Google".

7.2.4 The *getDescription* method

This method should return a string that describes a search. So the method should basically read the contents of a *SearchPanel* object and create a summary of the used search phrases. For example if the user has searched for the word "dogs" on Google, the returned description could be "Google: dogs".

7.2.5 *isConfigurable*

This method returns a boolean that tells if the Plugin is configurable. If the plugin doesn't need to be configured simply return false, otherwise true.

7.2.6 Optional: Configuration

Plugins can be (but they don't have to be) configurable by a user. Configurability is handled by three methods: *isConfigured(), getConfigPanel()* and *writeConfig(ConfigPanel cpanel).*

IsConfigured should return a boolean that tells if the plugin is configured.

GetConfigPanel() works just like *createPanel()* except that it used for a different purpose. Instead of a SearchPanel it should create a *ConfigPanel*. *ConfigPanel*s works exactly like *SearchPanel*s, they just have different names and are used for a different purpose, that is displaying the necessary fields for the user to configure a plugin.

WriteConfig needs to analyze the *ConfigPanel* object like *search* did, and then store the configuration data.

7.2.7 Optional: Removal of Duplicate Hits

The system has a feature for automatically removing duplicate hits from search results. By default all fields of a search results are used when comparing search results for equality. This default behavior is changeable by using the *setKeyFields* method in the constructor like this:

```
Example 7.2.7

MySearcher() {
        private String[] keyCompareFields = { "Title" };
        setKeyFields(keyCompareFields);
}
```

In the example above the class is told that only the title should be used for comparisons.

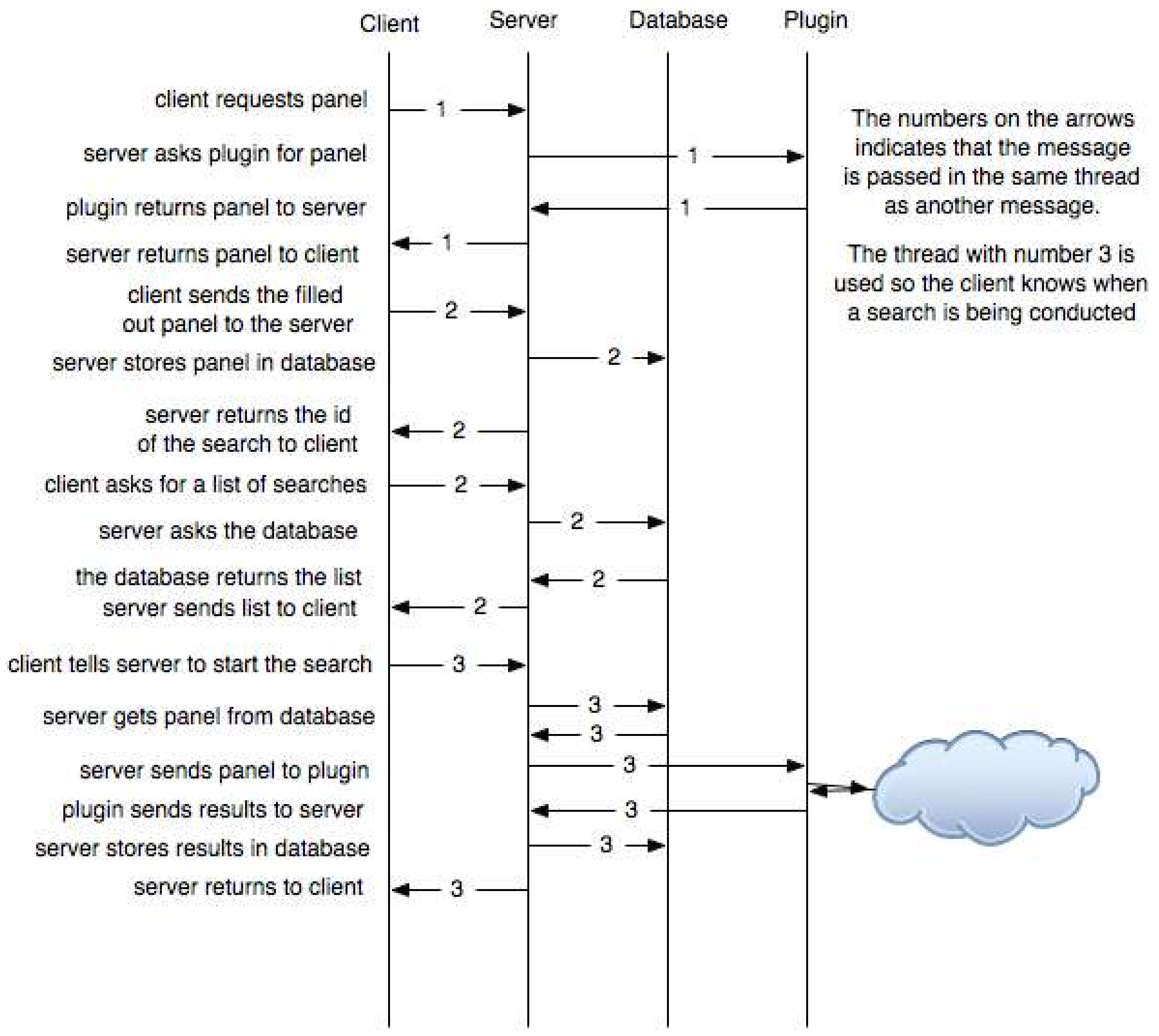
8. Major Features

8.1 Searches

Searches are implemented in plugin classes that inherit the *Searcher* class. When a user wants to do a new search, the request goes to the plugin which then creates a *SearchPanel.*The S*earchPanel* is then displayed and "filled" by the user and then sent back to the plugin object. When the plugin receives the *SearchPanel* it analyzes it, performs a search based on the contents and creates the search results.

There was a problem regarding the transportation of the *SearchPanel* object. Initially the class implemented *JPanel,* but we found out that the Java API didn't fully support serialization for gui-elements (objects are automatically serialized when they are tranported by using RMI). Another problem was that serializing this kind of objects into a database could result in incompatibilities between different Java versions. The solution was to change the class so that it contains a *JPanel* object that can automatically be xml-encoded by using a metod in SearchPanel.

The diagram below shows in more detail the steps involved from defining a new search to getting the searchresults to the client.


Client
Server
Database
Plugin
client requests panel
server asks plugin for panel
plugin returns panel to server
server returns panel to client
client sends the filled out panel to the server
server stores panel in database
server returns the id of the search to client
client asks for a list of searches
server asks the database
the database returns the list
server sends list to client
client tells server to start the search
server gets panel from database
server sends panel to plugin
plugin sends results to server
server stores results in database
server returns to client
The numbers on the arrows indicates that the message is passed in the same thread as another message.
The thread with number 3 is used so the client knows when a search is being conducted

8.2 Duplicate Removal

One problem with removing duplicates is to ensure that the following scenario doesn't happen:

Searches A and B yields the search results Sa and Sb. Sa and Sb are equal, so Sb is removed. Then A is removed and Sa is removed with A, so neither Sa or Sb exist anymore although B exist.

This was resolved by having an extra table that maps reference id:s to search id:s , so that one reference can belong to several several searches. The reference is thus completely removed first when there are no more mappings from the reference id to any search id:s.

SearchResult objects contain a hash value for doing comparisons. The hash is created based on the information fields it contains. As default all fields are used but this behavior can be overridden in a plugin (see 6.2.7 for practical information on how to do this).

8.3 Inclusion / Exclusion of Search Results

The implementation of this feature is pretty straightforward. *SearchResult* has a *status* field that is set by the client. The status field is used to tell the client if the search result is included or excluded.

8.4 Teams

Team features are still unimplemented. In a way it is technically possible for several users to connect to the a RevRatio server and modify a review, but this is not supported yet and highly insecure (currently it's not safe to start a server without a firewall), if the default “work locally” option is not enabled. To implement this feature at least the following things needs to be done:

1. Add new tables to database to hold user information.
2. Add authentication features to the server so that only users with access rights can use the server.
3. Ensure that the software doesn't crash or data doesn't get corrupted (just using a dbms may suffice, but I'm not completely sure) if several users do something simultaneously.

9. Appendix A (links)

hsqldb: http://hsqldb.sourceforge.net/

RMI: http://java.sun.com/products/jdk/rmi/

JDBC: http://java.sun.com/products/jdbc/

Java http://java.sun.com/

10. References

Requirements Documentation:
T-76.115 User Requirements Document, requirements.doc

Generated Javadocs:
http://susi.gforge.milkboy.yi.org/final/I2/revratio_javadoc.tgz