The CSI Journal on
Computer Science and Engineering
Vol. 8, No. 2 & 4 (b), 2010
Pages 14-20
Short Paper

A Java Pattern for File Uploading

Keivan Borna[1] Zahra Nilforoushan[2]

[1]Faculty of Mathematics and Computer Science, Kharazmi University, Tehran, Iran
[2]Faculty of Engineering, Kharazmi University, Tehran, Iran

Abstract

The main objective of this paper is to provide a dependable, secure and strong pattern for the file uploading using Java servlets and JSP. Our pattern consists of a Java servlet, upload handler, which is responsible for managing the uploading process and a Java class, Multipart Request, for holding the uploaded files, parameters and the parsed results. Multipart request consists of a private inner class named Source, a public inner class named file info, the constructor, and some other methods in order to describe the files encapsulated in the request. Our pattern and its performance is based on the high functionality of maps.

Keywords: File Uploading, Hash Map Data Structure, Java Programming Language, Java Servlet, Performance.

1. Introduction

File upload is too rarely discussed by even respectable Java literature and with the growth of the internet, file upload has now also played significant roles beyond email applications. Other Internet/Intranet applications such as Web-based document management systems and the likes of "Secure File Transfer via HTTP" require uploading files to the server extensively. For browsing some good survey on file/bean uploading with Java servlets see [1], [2], [3], [4] also [5] where a framework for building scalable wide-area upload applications is provided. In [6] uploads correspond to an important class of applications, whose examples include a large number of digital government applications.

This paper discusses file uploading using Java hash map data structure. We need to understand the underlying theory, i.e., the Map functionality and the HTTP request. This is done in Sections 2 and 3. In Section 4 we present the client and server side Java codes for developing our methods. In fact we provide a Java pattern for the server side part. Our pattern consists of a Java servlet for managing the uploading process, Upload Handler, and a Java class, Multipart Request, for holding the uploaded files, parameters and the parsed results. Multipart Request itself consists of a private inner class named *Source*, a public inner class named File Info, the constructor, and some methods in order to describe the files encapsulated in the request. Our method and its performance is based on the high functionality of maps. Finally Section 5 is devoted to conclusions.

2. Map Functionality

An array list allows one to select from a sequence of objects using a number, so in a sense it associates numbers to objects. But what if we like to select from a sequence of objects using some other criterion. A stack is an example: its selection criterion is "the last thing pushed on the stack". A powerful twist on this idea of "selecting from a sequence" is alternately termed a map, a dictionary, or an associative array. Conceptually, it seems like an array list, but instead of looking up objects using a number, we look them up using another object. This is a key technique in programming. The concept shows up in Java as the map interface.

Hash table based implementation of the map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The hash map class is roughly equivalent to hash table, except that it is unsynchronized and permits nulls.) This class makes

no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. This implementation provides constant-time performance for the basic operations "get" and "put"), assuming the hash function disperses the elements properly among the buckets. As an example of the use of a hash map, consider a program to check the randomness of Java's Random class. Ideally, it would produce a perfect distribution of random numbers, but to test this we need to generate a bunch of random numbers and count the ones that fall in the various ranges. A hash map is perfect for this, since it associates objects with objects (in this case, the value object contains the number produced by "Math. Random ()" along with the number of times that number appears).

An instance of hash map has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method. Iteration over collection views requires time proportional to the capacity of the hash map instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it is very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important. Performance can be adjusted via constructors that allow us to set the capacity and load factor of the hash table. As a general rule, the default load factor (0.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the hash map class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. If many mappings are to be stored in a hash map instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

A big issue with maps is performance. If one look at what must be done for a "get ()", it seems pretty slow to search through (for example) an array list for the key. This is where hash map speeds things up. Instead of a slow search for the key, it uses a special value called a *hash code*. The hash code is a way to take some information in the object in question and turn it into a "relatively unique" in for that object. All Java objects can produce a hash code, and hash Code () is a method in the root class Object. A hash map takes the hash code of the object and uses it to quickly hunt for the key. This results in a dramatic performance improvement. Hashing is the most commonly-used way to store elements in a map.

The "put (Object key, Object value)" method adds a value (the thing we want), and associates it with a key (the thing we look it up with). "get (Object key)" produces the value given the corresponding key. One can also test a map to see if it contains a key or a value with "contains Key ()" and "contains Value ()".

The standard Java library contains different types of maps:

Hash Map, Tree Map, Linked Hash Map, Weak Hash Map and Identity Hash Map. They all have the same basic map interface, but they differ in behaviors including efficiency, order in which the pairs are held and presented, how long the objects are held by the map, and how key equality is determined. For the ease of reader we bring the detailed descriptions of each map type.

Map: This interface maintains key-value associations (pairs), so we can look up a value using a key.

Hash Map: Implementation based on a hash table. (Hash map is used normally instead of hash table) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow us to set the capacity and load factor of the hash table.

Linked Hash Map: (JDK 1.4) Like a hash map, but when we iterate through it we get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than a hash map, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.

Tree Map: Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order. The point of a Tree Map is that we get the results in sorted order. Tree Map is the only map with the sub Map() method, which allows us to return a portion of the tree.

Weak Hash Map: A map of weak keys that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.

Identity Hash Map: (JDK 1.4) A hash map that uses == instead of "equals ()" to compare keys. Only for solving special types of problems; not for general use.

3. The HTTP Request

Knowledge of the HTTP request is critical because when we process an uploaded file, we work with raw data not obtainable from an Http Servlet Request object's methods such as get Parameter, get Parameter Names, or get Parameter Values.

Each HTTP request from the Web browser or other Web client applications consists of three parts:

1. A line containing the HTTP request method, the Uniform Resource Identifier (URI), and the protocol versions
2. HTTP request headers
3. The entity body

These three parts are explained in the following.

3.1. The Request Method, URI and Protocol Versions

The first subpart of the first part, the HTTP request method, indicates the method used in the HTTP request. In HTTP 1.0, it could be one of the following three: get, head, or post. In

HTTP 1.1, in addition to the three methods, there are four more methods: delete, put, trace, and options. Among the seven, the two methods that are most frequently used are get and post. Get is the default method. We use it, for example, when you type a URL such as "http://www.onjava.com" in the Location or Address box of your browser to request a page. The post method is common too. We normally use this as the value of the form tag's method attribute. When uploading a file, we must use the post method. The second part of the first part, the URI, specifies an Internet resource. A URI is normally interpreted as being relative to the Web server's root directory. Thus, it starts with a forward slash (/) that is of the form /virtual Root/page Name for example, in a typical JSP application the URI could be the following /eshop/login.jsp. More information about URI can be found in 2. The third component of the first part is the protocol and the protocol version understood by the requester (the browser). The protocol must be HTTP and the version could be 1.0 or 1.1. Most Web servers understand both versions 1.0 and 1.1 of HTTP. Therefore, this kind of Web server can serve HTTP requests in both versions as well. Combining the three sub-parts of the first component of an HTTP request, the first component would look like the following.

POST/virtual Root/page Name HTTP/version
For instance, POST/eshop/login.jsp HTTP/1.1

3.2. The HTTP Request Headers

The second component of an HTTP request consists of a number of HTTP headers. There are four types of HTTP headers: Pragma general, entity, request, and response. These headers are summarized in the following. The response headers are HTTP response specific, thus not relevant to be discussed here.

- Pragma header: The Pragma general header is used to include implementation specific directives that may apply to any recipient along the request/response chain. This is to say that pragmas notify the servers that are used to send this request to behave in a certain way. The Pragma header may contain multiple values. For example, the following line of code inform all proxy servers that relay this request not to use a cached version of the object but to download the object from the specified location: Pragma: no-cache.
- Allow header: This header lists the set of method supported by the resource identified by the requested URL. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. The Allow header is not permitted in a request using the post method, and thus should be ignored if it is received as part of a post entity. For instance, Allow: get, head.
- Content-Encoding header: This header is used to describe the type of encoding used on the entity. When present, its value indicates the decoding mechanism that must be applied to obtain the media type referenced by the Content-Type header. For example, Content-Encoding: x-gzip.
- Content-Length header: This header indicates the size of the entity-body, in decimal number of octets, sent to the recipient or, in the case of the head method, the size of the entity-body that would have been sent had the request been a get. Applications should use this field to indicate the size of the entity-body to be transferred, regardless of the media type of the entity. A valid Content-Length field value is required on all HTTP/1.0 request messages containing an entity-body. Any Content- Length header greater than or equal to zero is a valid value. For example, Content-Length: 32345.
- Content-Type header: The Content-Type header indicates the media type of the entity-body sent to the recipient or, in the case of the head method, the media type that would have been sent had the request been a get. For example, Content-Type: text/html.
- Expires header: The Expires header gives the date and time after which the entity should be considered invalid. This allows information providers to suggest the volatility of the resource or a date after which the information may no longer be accurate. Applications must not cache this entity beyond the date given. The presence of an Expires header does not imply that the original resource will change or cease to exist at, before, or after that time. However, information providers should include an Expires header with that date. For example, Expires: Thu, 29 Mar 2011 23:30:00 GMT.
- Last-Modified header: The Last-Modified header indicates the date and time at which the sender believes the resource was last modified. The exact semantics of this field are defined in terms of how the recipient should interpret it. If the recipient has a copy of this resource that is older than the date given by the Last-Modified field, that copy should be considered stale For example, Last-Modified: Thu, 10 Aug 2000 12:12:12 GMT.

3.3. The Entity Body

The entity body is the content of the HTTP request itself. It is best to illustrate this with an example. An example of an HTTP header is given below.

Accept: application/vnd.ms-excel, application/msword,
Accept-Language: en-au
Connection: Keep-Alive
Host: local host
Referrer: http://localhost/examples/jsp/num/demo.jsp
User-Agent: Mozilla/4.0 (compatible; Windows 98)
Content-Length: 31
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Last Name=borna, First Name=keivan

A lot is revealed in the HTTP header above. The first line tells us that the browser that sends this request can accept a number of file formats, including Microsoft Excel and Microsoft Word. It is followed by the language used (in this case, Australian English), the type of connection (keep-alive), and the name of the host (local host). It also tells the server that the request is sent from the demo.jsp which is located in http://localhost/examples/jsp/num/ directory. Then in the User-Agent entry, the Request tells that the user is using a Microsoft Internet Explorer version 4.01. The user operation system is also recorded to be Windows 98.

Following the header is two pairs of carriage-return line-feed characters. The length of this separator is 4 bytes since each carriage-return line-feed character pair consists of the ASCII characters number 13 and 10. From the HTTP header

above we can see that the body consists of the following code which has length 31:

Last Name=borna, First Name=keivan

This is clearly from a form with two input boxes: one called Last Name with the value “borna” and the other named First Name with the value “keivan”.

In the following we summarize the entity headers:

From: The from header specifies who is taking responsibility for the request. This field contains the email address of the user submitting the request.

Accept: This header contains a semicolon-separated list of MIME representation schemes that are accepted by the client. The server uses this information to determine which data types are safe to send to the client in the HTTP response. Although the Accept field can contain multiple values, the Accept line itself can also be used more than once to specify additional accept types (this has the same effect as specifying multiple accept types on a singe line). If the Accept filed is not used in the request header, the default accepts types of text/plain and text/html are assumed. For example, accept: text/plain; text/html Accept; image/gif; image/jpeg.

Accept-Encoding: This header is very similar to the accept header in syntax. However, it specifies the content-encoding schemes that are acceptable in the response. For instance, Accept-Encoding: x-compress; x-zip.

Accept-Language: This header is also similar to the Accept header. It specifies the preferred response language. The following example specifies English as the accepted language: Accept-Language: en.

User-Agent: The User-Agent, if present, specifies the name of the client browser. The first word should be the name of the software followed by a slash and an optional version number. Any other product names that are part of the complete software package may also be included. Each name/version pair should be separated by white space. This field is used mostly for statistical purposes. It allows servers to track software usage and protocol violation. For example, User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98).

Referrer: This header specifies the URI that contained the URI in the request header. In HTML, it would be the address of the page that contained the link to the requested object. Like the User-Agent header, this header is not required but is mostly for the server’s statistical and tracking purpose. For example, Referer: http://localhost/Details.htm.

Authorization: The Authorization header contains authorization information. The first word contained in this header specifies the type of authorization system to use. Then, separated by white space, it should be followed by the authorization information such as a user name, password, and so forth. For example, Authorization: user abc: 1 2 3.

If-Modified-Since: This header is used with the GET method to make it conditional. Basically, if the object hasn’t changed since the date and time specified by this header, the object is not sent. A local cached copy of the object is used instead.
For example, If-Modified-Since: Thu, 13 May 2013 22:15:30 GMT.

4. Implementations

We just finished dissecting an HTTP request and now we are ready to take this information to the coding stage. To program a complete file upload application, we need to know both the server side and the client side. First we present how to program the HTML on the client side, and then we will study the Java code for the server side.

4.1. The Client Side HTML

Prior to the RFC 1867 standard there were eight possible values for the type attribute of an input element: checkbox, hidden, image, password, radio, reset, submit, and text; see 5.

While these form elements have proven useful in a wide variety of applications in which input from the user needs to be transferred to the server, none of these is useful for sending a file, either text or binary. Next, it was proposed that the type attribute of an input element has another possible value: file. In addition, it defines a new MIME media type, multipart/form-data, and specifies the behavior of HTML user agents when interpreting a form with enctype=”multipart/form-data”. More precisely, the client side looks as follows:

```
<form action="servlet/Upload Handler"
enctype="multipart/form-data"    method="post"\>
  File to Upload: <input name="filename" type="file"/>
  <input type="submit" value="Browse" />
</form>
```

When an input tag of type file is encountered, the browser might show a display of previously selected file names and a “Browse” button or selection method. Selecting the “Browse”' button would cause the browser to enter into a file selection mode appropriate for the platform. Window-based browsers might pop up a file selection window, for example. In such a file selection dialog, the user would have the option of replacing a current selection, adding a new file selection, etc. For more details about this API see Sun's web site. What is of importance here is the HTTP request and how to process it. When working with the HTTP request, we will work with either the javax. servlet. servlet *Request* interface or the javax. servlet .http .http servlet *Request* interface.

4.2. The Server Side Code

Internet programming in Java always involves the use of servlets or JSP pages, depending on the architecture one choose to implement. This means that we will use one of the classes or interfaces in the javax. servlet package or the javax. servlet. http package. The most important interface is the servlet interface in the javax. servlet package that must be implemented by all servlets. The servlet request interface is a generic interface that is extended by Http servlet Request to provide request information for HTTP servlets. We leave to

the reader studying Http servlet *Request* interface important methods. In order to describe our method server-side's part we use a Java servlet named Upload Handler which is responsible for the output. We mention that our implementation also holds for a wide range of uploading several files simultaneously.

```
Public class upload handler extends Http servlet {
Protected void do post (Http servlet request req,
Http servlet response res) throws
servlet exception {
  Http session session = req. get session ();
  Res. set content type ("text/html");
  Print writer out = null;
  Try {
    Out = Res. Get writer();
    Multipart request mr = New multipart request (req);
    String c path = "E:\\temp\\";//for instance
    Out. Print ("Uploaded Files <ul>");
    For (Iterator k = mr. get file info names (); k. has Next ();){
      String name = (String) k. next ();
      Out. Print ("<li>"+name);
      Multipart request. File info [] values =
                                  mr. get File Info Values(name);
      Out. Print ("<ol>");
      For (int i = 0; i < values. length; ++i) {
        If (values[i]. get Content ().length <= 6144000) {
          String c File name = values[i].get Source File name ();
          Byte [] a content = values[i].get Content ();
          If (c file name. last index of ("\\") > -1)
            C file name = c file name. Sub string (
            C file name. Last Index of ("\\") +1);
            Out. Print ("<li>"+c file name);
            Out. Print ("<br>"+values[i].get Mime Type ());
            Out. Print ("<br>"+values[i].get Content ().length+"
                                                    Bytes");
            File g = New file (c path);
            File f = New file (c path+c file name);
            If (g. exists ()) g. Delete ();
            Else g. mkdirs ();
            File output stream fos = new
                        File output stream (f);
            Fos. Write (a Content) ;
            Fos. Flush ();
            Fos. Close ();
          }
          Else {
            Out. Println ("<br>"+"File size is larger than
                                                  6MB.");
          }
        }
      }
  } Catch (Exception e) {
     Out. Println (e. to string ());
  }
  Out. Flush ();
  Out. Close ();
}
}
```

For holding the uploaded files, parameters and the parsed results we use Multi part Request class. This class consists of a private inner class named Source, a public inner class named File Info, the constructor, and some methods in order to describe the files encapsulated in the request. A general view of this class is shown in the following:

```
Public class multipart request {
    // to hold parsed results
    Private mime multipart mime parts;
    // to hold uploaded parameters
    private hash map params = new hash map ();
     // to hold uploaded files
    Private hash Map files = new hash Map ();
    // Scratch buffer space
    Private byte [] buf = new byte [8096];
    Private class source {}
    Public class file info {}
    // the constructor
    Public multipart request (Http servlet request req) throws
      Messaging exception, IO exception {}
       // Get parameter, get parameter values, get file info names,
       //Get file info and get file info values methods in order to
       //describe the files encapsulated in the request.
}
```

The private inner class source which implements javax. activation. *Data source* is a simple implementation of the Data source interface. It provides the bridge between the Http servlet request and the java mail classes.

```
Private class Source implements
                    Javax. Activation. data source {
  Private input stream stream;
  Private string mime type;
  Source (Http servlet request req) throws IO exception {
    Stream = req. get input stream ();
    Mime type = req. get header ("CONTENT-TYPE");
  }
  Public input stream get input stream () {
    Return stream;
  }
  Public string get content type (){
    Return mime type;
  }
  Public output stream get output stream () {
    Throw new runtime exception ();
  }
  Public string get name () {
    Throw new runtime exception ();
  }
};
```

```
Public class File info {
  Private byte [] content;
  // the byte-copy of the file's contents
  Private String source name;
  // the name of the file on the browser's system
  Private String mime type;
  //the mime type supplied by the browser
  Public file in fo(byte [] content, string source name, string
                                                  Mime type) {
    This. Content = content;
    This. Source name = source name;
    This. Mime type = mime type;
  }
  Public byte [] get content () {
    Return content;
  }
  Public string get source filename () {
    Return source name;
  }
  Public string get mime type () {
    Return mime type;
  }
}
```

```
Public multipart request (http servlet request req) throws
Messaging exception,
IO exception {
//Here's the line which does all of the parsing.
//The request size and content type could be checked before
//calling, if desired.
  Mime parts = new mime multipart (new Source (req));
//Now iterate over the parsed results
  In part Count = mime parts. Get count ();
  For (in i=0; i<part count; ++i) {
    Mime body part bp = (Mime body part)
                          Mime parts. Get body part (i);
    String disposition = bp. get header ("Content-
                                    Disposition","");
//Here we use the filename to indicate if this is a file or a
//parameter.
//could instead use bp. Get content ().get class () to indicate if //we
have a String, an Input stream, or a (nested) multi part.
     String filename = bp. Get file name ();
//this   file name appears to lack "\" chars.
     If (file name == null) do parameter (bp, disposition);
     Else do file (bp, disposition);
  }
}
```

```
Public iterator get parameter names (){
  Return params. Key set  (). Iterator ();
}
Public string get parameter (String name){
  List value list = (List) params. Get (name);
  If (value list == null) return null;
  Return (String) value list. Get (0);
  // Return first value, as per servlet 2.2 API
}
Public string [] get parameter values (String name){
  List value list = (List) params. Get (name);
  If (valuelist == null) return null;
  Return (String []) value list. To array (new
                                    String [value list. Size ()]);
}
//Return an iterator for the file info items describing the files
//Encapsulated in the request.
Public iterator get file Info names (){
  Return files. Key set  (). Iterator ();
}
Public File info get File info (String name){
  List file list = (List) files. Get (name);
  If (file list == null) return null;
  Return (File info) file list. Get (0);
}
Public file info [] get file info values (String name){
  List file list = (List) files. Get (name);
  If (file list == null) return null;
  Return (File info []) file list. To array (new
                                    File info [file list. Size ()]);
}
Private string find value (String parm, String header) {
  String to kenizer st = new string to kenizer (header, "; =");
  While (st. has more tokens ()) {
  String token = st. next token ();
  If (token. Equals ignore case (parm) ){
    Try {
      Return st. next to ken ("\"=");
    }
    Catch (No such Element Exception e) {return "";} //e.g.
      File name="";
    }
  }
  Return null;
}
```

The public inner class *file info* is used to store information about uploaded files. Users of the multipart request class should generally refer to this class as multipart request. File info (as for usual Java rules).

The constructor of multipart request accepts an Http servlet request (which it assumes to be from a post of a ENCTYPE=”multipart/form-data” form) and parses all the information into a mime multipart object. It then iterates through that parsed object, extracting the parameters and files from it for the user.

The get parameter (), get parameter values (), get file info names (), get file info () and get file info Values () methods describe the files encapsulated in the request. Finally we note that buffered input stream *and* byte array output stream are used to handle the required processing for a file and to extract a parameter value from a header line.

Finally the do parameter, do file methods are presented in the following:

```
Private void do parameter (mime body part bp, String
Disposition) throws
Messaging exception, IO exception {
  String name = find value ("name", disposition);
  String value = (String) bp. Get content ();
  List value list = (List) params. Get (name);
  If (value list==null) {
    Value list = new Lin ked List ();
    Params. Put (name, value list);
  }
  Value  list. Add (value);
}

Private void do File (Mime body part bp, String disposition)
Throws messaging exception, IO exception {
   String name = find value ("name", disposition);
  String filename = find Value ("filename", disposition);
  Buffered input stream in = new
  Buffered input stream (bp. Get input stream ());
  Byte array output stream out = new
  Byte array output stream (in. available ());
  int k;
  While ((k=in. read (buf)) != -1 )
   Out. Write (buf, 0, k);
  Out. Close ();
  File info f = New file info (out. to Byte array (), file name, bp. Get
content type ());
  List file list = (List) files. Get (name);
  If (file list==null) {
    File list = new Lin ked List ();
    Files. Put (name, file list);
  }
  File list. Add (f);
}
```

5. Conclusions

File upload is not an easy topic, and the code to process it is Not readily available. This article has presented the theory and the code for a file upload servlet using Java hash maps.

Acknowledgments

The first author is thankful to the Elite National Foundation of Iran for partial financial support.

References

[1] T. Anderson, "XForms tip: Use XForms to upload a file to Java," IBM series, 2004.

[2] B. Kurniawan, "Uploading files with Beans," ONJava.com, October 2001.

[3] E. Nebel, "Form-based file upload in HTML," Xerox Corporation, 1995.

[4] Apache Software Foundation, http://jakarta.apache.org/commons/fileupload/, January, 2008.

[5] S. Bhattacharjee, W. C. Cheng, C. F. Chou, L. Golubchik, and S. Khuller, "Bistro: a framework for building scalable wide-area Upload applications," *ACM Trans. SIGMETRICS Performance Evaluation*, vol. 28, no. 2, pp. 29-35, 2000.

[6] Y. Yang, L. Cheung, and L. Golubchik, "Technology to support data gathering via the web: Data assignment in fault tolerant uploads for digital government applications: a genetic algorithms approach," *Proc, Int'l conf. Digital Government Research*, pp. 29-38, 2005.

Keivan Borna is an Assistant Professor in the Department of Computer Science at Faculty of Mathematics and Computer Science of Kharazmi University of Tehran since 2008. He completed his Ph.D. in September 2008 from the Department of Mathematics, Statistics and Computer Science of University of Tehran in Computational Commutative Algebra. He was a visiting scholar in ``Dipartemento di Matematicha, Universita' di Genova-Italia" and ``Department of Mathematik and Informatik at Essen University, Germany'', from Sep. 2007 to Apr. 2008 during his graduate work. He is very interested in studying and researching in interdisciplinary topics like "Cryptography", "Approximate Algorithms", "Musical Data Analysis", "Motion Planning" and "Computational Geometry". He published some papers in such areas. He is the author of the ``Advanced Programming in JAVA'' (in Persian) and is the member of ``Elite National Foundation of Iran''.
E-mail: borna@khu.ac.ir

Zahra Nilforoushan she is an Assistant professor in the department of computer engineering at faculty of Engineering of kharazmi university of Tehran. She completed her Ph.D. in March 2009 from the department of mathematics and computer science at amirkabir university of Technology and her thesis was about the generalization of the voronoi diagram in some non-Euclidian spaces. She was a visiting scholar in ``Institute fur informatics at universitat Bonn (Germany)'' from Oct. 2007 to Apr. 2008 during her graduate work. Previously, she received her M.Sc. in algebraic geometry from the Department of Mathematics and computer science at Amirkabir University of technology in 2002. She did some research about the applications of algebraic geometry in the study of robotics as an undergraduate.
E-mail: nilforoushan@khu.ac.ir

Paper Handling Data:

Submitted: 22.06.2011
Received in revised form: 25.05.2013
Accepted: 15.07.2013
Corresponding author: Dr. Keivan Borna,
Faculty of Mathematics and Computer Science, Kharazmi University, Tehran, Iran.