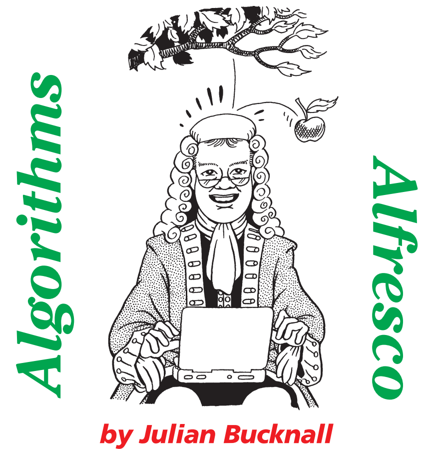
Caching In On Algorithms

The first applied Algorithms Alfresco!



by Julian Bucknall

I received my new credit card the other day. The usual magnetic stripe and hologram and entirely-too-small space on which to inscribe your signature and bright glossy numbers where the gold hasn't rubbed off yet: all this I was expecting. What I wasn't was the chip on the card: a cool new feature from this particular credit card company. Apparently certain retailers have a new point-of-sale outlet over which they merely wave the card and the system reads the chip on the card. Cool, except this card is a British one and they don't have any of these new readers in Colorado *[Us Brits are always well ahead of the game... Ed]*. Oh well, it makes a great conversation piece at parties. The first time I showed it to my wife, she wanted to know how it worked. I explained that the reader emitted radio waves that were picked up by the chip and generated enough energy for the chip to respond with its inbuilt data. Like a laser scanner in a supermarket she wanted to know. No, I said, it uses radio instead of light. But it works like a laser scanner, she continued. I should have said yes, kinda, at that point and dropped the subject. Essentially we had a problem of communication: she was describing one algorithm and I was describing another, but similar, algorithm. Both algorithms were applicable to the problem space, however the 'chip' algorithm is presumably harder to fake.

The one trap it is easy for me to fall into when writing a series of articles on algorithms and data structures is that of enthusiasm. I get so involved in writing about this, that and the other thing that I completely forget about the programmer in the street who wants to use it. I usually rattle off a simple application of the technique. He's probably wondering that all this information is very fine and advances the total knowledge of the human race no end, but he's got an application to finish by Friday and would like to know whether this snazzy algorithm is better than what he has now and, if not, which one would be better, thank you very much.

To be honest, there's no way I could possibly answer all of those types of questions in an article. Each particular application would require a different mix of algorithms and structures because each application has a different mix of features and requires a different mix of data. What to do?

Enter *Applied Algorithms Alfresco*. This is going to be an occasional series within this column where I discuss a particular real-world problem, go through the design, and show how to select a good mix of algorithms and data structures to use in the implementation. By a process of showing how to approach the problem, and how to grade and select the algorithms, I hope to teach you how to solve your own problems without having to put myself at the end of a consulting phone.

Eventually, I'd like to hear from you, my readers, on particular real life problems you have to solve. If the problem is such that I can wrap a good article around it and introduce some reusable techniques and classes, so much the better. Note that I cannot write your applications for you. Sorry and all that, but that's what you are being paid for, not me.

The Problem

My first applied algorithms article is to implement a file cache. Let's explain where I got the idea from so you can see the problem we're going to solve.

A web browser uses a protocol called HTTP to download files from websites. These files include HTML, images, XML and so on. It uses these downloaded files to build up a display on your screen, so to the average user it doesn't seem as if the browser is downloading files, it's displaying web pages, by golly. To help provide a more responsive experience, it uses a file cache to cache all this stuff it has downloaded from the internet. So, for example, if a particular page has ten little right pointing arrow images on a web page for a bulleted list, the browser only has to download the image once, and can reuse the downloaded file nine other times. If the user moves to another page on the same web site that also has the little right pointing arrows, again the browser doesn't have to download the image from the site and can reuse the file it's already received. Each image that is repeated on a web page or in an entire site is likely to be small, but adding up all the times they're used could amount to a sizeable chunk of download time.

So the browser caches the files it downloads. If it can find a particular file in the cache then it reuses it and therefore doesn't need to download it again over the (potentially slow) internet. The HTTP protocol has a complex set of operations that were designed to help caches, to know when a file *has* to be downloaded or when a cached version may be used.

Groovy, but eventually the browser would fill up your hard disk with files from sites you

visited once and never went to again. So the browser usually has some feature whereby you define how big the cache is allowed to grow on your hard disk before it starts deleting the least recently used files. On my installation of Netscape, for example, I've set my disk cache to be 7.5Mb. Once the cache has saved so many files that this limit has been reached or breached, Netscape deletes as many files as it needs to, to bring the total size of the cache down to the limit again. It deletes those files that haven't been used recently. Internet Explorer works in a similar way. I've seen some caches that also have a feature whereby all files over seven days old (or whatever) are automatically deleted on the premise that if you don't visit a site weekly, you probably wouldn't be visiting it ever again, or at least very infrequently.

The files the browser downloads have names like

```
http://www.turbopower.com/
  images/products/Systools/
  Sys3HPtitle.gif
```

You can hardly create a file with this name on your hard disk: Windows would get very confused by the colon and the slashes. So what we need to do is to implement an indexing scheme as well: index this external name to a file name that we actually create on the disk. For example, this particular web file might get associated with a file called XFGRETSA.GIF in the cache folder, and it would be up to the cache to read the index when asked for the blahblahblahblah/Sys3HPtitle.gif file and discover that it could read XFGRETSA.GIF from the disk (or, better still, find it in memory) instead of going off to TurboPower's site, connecting to the web server, asking for the file blahblahblahblah/Sys3HPtitle.gif and receiving it.

The index must be resilient enough to cope with stupid users like me who go into the cache folder and delete some of the files. ('Oooo, let's see if this breaks Netscape!')

What else? Well, you may not know this, but files downloaded from a website have an expiry date attached to them. A lot of the time the expiry date is infinite (the file never 'expires') but sometimes the web designer will set the expiry date so that browsers out there are forced to download a new version of the file from the website every now and then. We should cater for this in our cache as well. If there is no expiry date, caches will usually set it to be 24 hours after download. (Please note that I'm deliberately simplifying the HTTP protocol here. The mechanism by which a client cache decides that a file needs downloading again is very bizarre and if I were to describe it all it would drown the purpose of this article in a sea of complexity.)

The Design

Already you can see the complexities of the problem I've chosen. We can't just pick up Knuth's *The Art of Computer Programming, Volume 2* and find the algorithm on page 432. We are going to have to design the solution and work out which primitive algorithms and data structures we can profitably use. There might even be some algorithm that we have to design from scratch.

The problem as I see it boils down to this: design and implement a file cache that could be used by a web browser. We have to design an indexing scheme to associate a web file address with a file name on our local disk. This indexing scheme should cope with files being deleted by another external process, with different ways of marking a cached file as being available for deletion (download date, expiry date, whatever). We have to implement a scheme whereby the cache is limited in size and, indeed, have a facility to clear the cache. The design must also encompass the facility for storing the files temporarily in memory, again with a cache memory limit.

What the cache shouldn't do is get involved in the actual mechanism of downloading the files from the website concerned. This is functionality that goes beyond the attributes of a cache and into the realm of an HTTP protocol class.

Usually what I do now is to design a class or classes that embody the functionality I want. I don't get involved in algorithm choice at this stage: it's only by defining the interface that it becomes more obvious what is needed.

The first thing to think about is what we need to store about a downloaded file that's in the cache. First, there's the external name, or the web name. Second there's the filename for the file we have on disk. We also need to store, at a minimum, the timestamp for the time we downloaded this file, the time we last used the file, and the timestamp for its expiry date (we could also store the initial creation time, the last used time, and so on, but they won't help us in our quest to write the perfect cache). When we need to mark files for deletion (because the cache has grown too large), we

➤ *Listing 1: The internal cache item class.*

```
TCacheItem = class
  private
    FDownloadDate   : TDateTime;
    FExpiryDate     : TDateTime;
    FExternalName   : string;
    FInternalName   : string;
    FLastUsedDate   : TDateTime;
    FSize           : longint;
  protected
  public
    constructor Create(const aInternalName : string; const aExternalName :
      string; aExpiryDate : TDateTime; aDownloadDate : TDateTime;
      aSize : longint);
    constructor LoadFromStream(aStream : TStream);
    destructor Destroy; override;
    property DownloadDate : TDateTime read FDownloadDate;
    property ExpiryDate : TDateTime read FExpiryDate;
    property ExternalName : string read FExternalName;
    property InternalName : string read FInternalName;
    property LastUsedDate : TDateTime read FLastUsedDate;
    property Size : longint read FSize;
end;
```

shall be looking at the expiry date and at the last-used date. Those files that have expired will be deleted; those files we have used least recently will be deleted. We should also store the file size: this will help us calculate the total size of the cache, both on disk and currently in memory.

So that's our first class. Listing 1 has a proposed interface to this `TCacheItem` class. For now all I've done is indicate that the class has a constructor and a destructor and properties that expose the data for each item. I'm deferring any decision to add other methods and properties until later, once we've discussed the file cache class. This item class, by the way, is only going to be used by the cache class; the users of the cache class will not see this internal item class.

The cache class itself is more involved. It seems plain that the cache class is going to be a list of cached files in some form. We won't decide exactly what form until we've done a little more design. We should be able to add a file to the cache, that's obvious enough. To do that we should supply the external name of the file, the contents of the file and its expiry date. From this the `Add` method should be able to calculate the last used date (which is right now) and the size of the file. It will create a new file in the cache folder (mental note: we need a `Folder` property to define where this is) whose name it will decide on and remember. As to how we pass the file contents to this method, I think the easiest way would be to provide a stream of some kind holding the contents of the file. We could then easily copy the data to our internal file. It would be up to the caller to create the stream; I dare say most often the caller would use a buffered file stream but for smaller files a memory stream would suffice (the caller gets to see how big the file is before it's download: it's held in an HTTP header before the data).

OK, what's next? We should have a method that returns whether a given external file name is in the cache or not. If it is in the cache, the `Get` method should return a stream with the data (this again might be a buffered file stream, but at this stage the cache knows the file size so it could use a memory stream). The caller would read the data in this stream and then call another method to say that the stream was finished with (`GetComplete`), otherwise the cache would not know when the caller had finished with the stream. This latter method would update the last-used date for the cached item and maybe also free the stream.

If the external file were not found in the cache, or the file were expired, the `Get` method would return a nil stream and then presumably the caller would be going out to the internet to download the file, and then adding this data to the cache in the manner we've already described.

We should provide a `Delete` method whereby we can delete a file from the cache given its external name. This enables the browser to force itself to refresh a web page and all associated files from the web, rather than from the cache. The final method is a `Clear` method for deleting all files from the cache. As for properties, I've already mentioned the `Folder` property, but we should also have a `MaxDiskSize` property for the maximum size of the cache on the disk in bytes.

The Interface

This gives me Listing 2 as the interface of the cache class. Notice that I'm deliberately ignoring all implementation issues and what goes in the `private` or `protected` sections. All I want to do is to nail down what the cache looks like to the outside world. This is an important point:

```
TaaFileCache = class
    private
      ???
    protected
      ???
    public
      constructor Create;
      destructor Destroy; override;
      procedure Add(const aExternalName : string; const aExpiryDate : TDateTime;
        aStream : TStream);
      procedure Clear;
      procedure Delete(const aExternalName : string);
      function Get(const aExternalName : string) : TStream;
      procedure GetComplete(const aExternalName : string);
      property Folder : string read FFolder write fcSetFolder;
      property MaxDiskSize : integer read FMaxDiskSize write FMaxDiskSize;
  end;
```

➤ *Listing 2: The interface to the file cache class.*

as I've designed it so far, the user of the file cache class has no knowledge of any internal information or structures at all. He doesn't know what names the cache gives files on disk (or even that the cache uses separate files at all, for all he knows the cache could be using BLObs on an SQL server). He doesn't know the indexing scheme used by the cache. He doesn't know how the cache identifies which files are due for deletion to bring the overall size down, or even when this process occurs.

Because we've defined the interface and published it, we are free to make whatever choices we like for the implementation. We could implement a version of the file cache class, and then provide a better, more efficient, implementation in version 2. It won't affect the user of the class. (Of course, sometimes in the real world, we find that we have to smudge the interface a little to get some extra speed or functionality out of it, but we're going to endeavor not to do that here.)

The Implementation

Right. Implementation, then. Let's identify some processes that must happen with our cache, drawing up a non-exhaustive list:

1. Given an external name, we must efficiently find the cache item for that name, or, just as efficiently, discover that the name is not present. This will be used internally by the `Get` and `Delete` methods.

2. At certain times when using the cache we have to efficiently identify the items whose files have expired, so that we can delete

them. This would probably be used most of all during the `Add` method when the user adds a file to the cache, since it is at that point we have to decide if the cache is too large, and it's at that point that we have to delete excess files.

3. At the same or similar times, we have to efficiently scan through the items by last-used date so that we can delete the items we haven't used in a long time until the cache has been brought back down to size again. `Add` would call this routine.

Now we can start to think about algorithms and data structures. Hooray! This is where your knowledge of the possibilities will help you out. The first process seems to indicate using one of Delphi's standard classes, the `TStringList`. This possibility goes like this: store all the cache item objects in a sorted string list, with the external name as the sort order. The string list would be used to efficiently find a given external name. Those of you who've been reading these columns for a while would recognize this search as a binary search, an O(log(n)) algorithm. Populating the list would be an O($n$log(n)) process. Seems pretty good. This implementation possibility does not help us with processes 2 and 3, though.

Actually, we can go one better than this, by analyzing the situation a little further. If you read the description of the first process again, you'll see that it accurately defines the main benefit of a hash table: a very fast O(1) search. It seems then that a hash table implementation using the external name as key is just what the doctor ordered.

However, at the risk of beating the point to death, it still doesn't help with the second or third process. Being able to access a file in the cache quickly by external name is not going to help with determining those files past their sell-by dates, or those that haven't been used in a long time. To make these operations efficient it seems that we would need to sort the file list by expiry date or last-used date. Of course, we could ignore efficiency considerations for these operations: after all, one might think that cleaning up the cache is an infrequent process at best, but in reality, every time we visit a website we'll be getting more files to put in the cache and therefore would need to delete files from the cache.

It's at this point that most people would plump for one data structure and ignore the inefficiencies caused by using it outside its main purview. This is shortsighted to say the least. What we shall do instead is have *three* data structures holding this information: a hash table for the quick access by name, a sorted list by expiry date, and a sorted list by last-used date. Whenever we add or delete a cached item, we'll be updating all three containers. Although we will be doing all this extra work, we shall still be creating a class that is more efficient than just using a single container and then resorting to sequential searches for the other processes.

But think again. We have reasoned that a hash table would be better than a string list to hold the items for fast access by external name. Is there any other data structure that would be better than a sorted list for holding the items sorted by expiry date and by last-used date? All we need to do with the last two processes is to identify the earliest dates. We're not particularly interested in *iterating* the items by expiry or last-used date, we just need to *efficiently identify and remove* the earliest ones. This in fact is the definition of a priority queue.

Recall from November 1998's *Algorithms Alfresco* that a priority queue is a data structure with which you can add items in any order and remove them biggest (or smallest) first, however we may define 'biggest' or 'smallest'. (The name *priority queue* comes from its initial purpose: retrieving items in order of priority.) All we need to do then is to define 'priority' as being the expiry date or last-used date and use a priority queue for each purpose.

So our design calls for three containers. When we add a new item we must add it to all three containers. When we delete an item we must delete it from all three containers before freeing the item in question. When we use an item, we update its last-used timestamp and this would require moving it in the priority queue that stores the items by last used date, but leaving it alone in the other two containers. I'm sure you get the idea by now.

I have some bad news and some good news. The bad first: Delphi doesn't come with a hash table or priority queue as standard. The good news: in past issues of *The Delphi Magazine* I've presented implementations of the hash table in February and March 1998, and the priority queue in November 1998.

What else did I mention in our functional spec that I haven't yet covered? The disk index. Although we have a method of associating an external name with a disk file, the hash table, we still have to make this persistent. Our file cache wouldn't be very popular if it had to be regenerated from scratch every time we used the browser. We don't have to get very sophisticated here (there's no real point in using a database engine, for instance). The simplest method is merely to stream the set of cached items to disk; for this, we would need `StoreToStream` and `LoadFromStream` methods for our cache class. These methods would call similarly named methods of the item class.

The Reuse

At this point we can start coding. Whenever I do this kind of thing, I tend to gather up the primitive classes and routines first, and then start coding the classes that pertain to the particular problem at hand. The hash table first, then.

A brief recap about hash tables would be in order. A hash table is an array storing items that are uniquely identified by a *key*. The key can be a string (as in our case here) or an integer or anything you like. When an item is added to the hash table, the key is *hashed* to produce an index into the array

```
procedure TaaFileCache.Add(const aExternalName : string;
  const aExpiryDate : TDateTime; aStream : TStream);
var
  CacheItem    : TCacheItem;
  InternalName : string;
  QualName     : string;
  Stream       : TFileStream;
begin
  {create a unique file name}
  InternalName := fcGetUniqueFileName;
  QualName := fcGetQualifiedFileName(InternalName);
  CacheItem := nil;  {create a new cache item}
  try
    CacheItem := TCacheItem.Create(InternalName,
      aExternalName, aExpiryDate, Now, aStream.Size);
    {try and add the item}
    if not fcAddItem(CacheItem) then begin
      {if it already exists, delete unique file we created}
      DeleteFile(QualName);
    end else
      {otherwise copy the stream over}
      Stream := TFileStream.Create(QualName,
        fmOpenReadWrite);
      try
        Stream.CopyFrom(aStream, 0);
      finally
        Stream.Free;
      end;
    end;
    {check the maximum disk usage}
    if (FCurDiskSize > MaxDiskSize) then
      fcCleanUp;
  except
    {if a problem occurred, we need to delete the cache
      item and the internal file, and reraise the exception}
    CacheItem.Free;
    DeleteFile(QualName);
    raise;
  end;
end;
```

➤ *Listing 3: Adding an item to the cache.*

and the item is placed there. All very simple, but the real complexity comes when the element in the array already has an item present. This is known as a *collision*. What do we do? We can't just replace the item with our new one: we'll lose the old item, which is presumably still being used.

There are two common collision resolution methods in general use. The first is *linear probing*. In this algorithm, we try and place the new item in the next element. If that one is also occupied, try the next one, and so on, so forth. When it's time to find an item by its key, we hash the key to produce an index and look at that element. If it's the right one, we're done; if not, we start looking at subsequent elements until we either find an empty element or we found the item for which we were searching. If the hash table's *load factor* grows too large (the load factor for a linear probe hash table is the number of items in the hash table divided by the total number of entries or slots in the hash table), say about 2/3, the hash table is grown and all items reinserted.

The second common collision resolution method is known as *chaining*. Instead of each element in the hash table array being an item, it is a linked list of items. When we add an item we merely add it onto the end (or more often the beginning) of the linked list at that element. The find algorithm then reduces to hashing the key, getting an index, and then following the linked list at that element.

```
function TaaFileCache.fcAddItem(aCacheItem : pointer) : boolean;
var
  CacheItem : TCacheItem;
  Dummy     : pointer;
begin
  {typecast the cache item to something recognizable}
  CacheItem := TCacheItem(aCacheItem);
  {make sure it isn't already in the cache, if it is free the passed
   object to make sure we don't have a leak}
  if FItems.Find(CacheItem.ExternalName, Dummy) then begin
    CacheItem.Free;
    Result := false;
    Exit;
  end;
  Result := true;
  {add it to the hash table first}
  FItems.Insert(CacheItem.ExternalName, CacheItem);
  {add it to the expiry queue}
  CacheItem.ExpiryHandle := FExpiryQueue.Add(CacheItem);
  {add it to the lastused queue}
  CacheItem.LastUsedHandle := FLastUsedQueue.Add(CacheItem);
  {increment the disk size}
  inc(FCurDiskSize, CacheItem.Size);
end;
```

➤ *Listing 4: The real code to add an item.*

We'll either find the item we want or we'll simply run out of linked list. Again the load factor comes into play and if it grows too large we must grow the entire hash table (here the load factor is calculated in the same way and is equal to the average length of the linked lists, and, with string keys, we would like the average length of each linked list to be five or less).

Amazingly, looking back through my articles for *The Delphi Magazine*, I find that I've never implemented a chained hash table (I did use one for the LZ77 compression method in May 1999, but it was customized for that purpose). Since this article is all about reuse and using classes in general use, I'm going to stick with the linear probe hash table and leave the chained hash table for another time.

As for the priority queue, we shall need to use the more specialized version. Recall that the standard queue allows us to insert items in any order and remove them smallest (or largest) first. Other items in the priority queue are hidden. For our purposes, though, we have a small problem: every now and then we shall have to change the last-used date for a given item, and it is very likely that the item for which we're changing the date is stuck deep inside the priority queue. Also, we'll be deleting arbitrary items in a priority queue (for example, the last-used priority queue might indicate an item to be deleted and we'll have to delete it from the expiry priority queue). In January 1998's article on graph algorithms I presented a version of the priority queue that allowed you to delete and change the 'priority' of a given item and to rebalance the internal tree. We'll use this version, then.

The Coding

At this point, we can start writing our code in earnest. We have selected the 'low-level' containers we shall be using and must now weave our application-specific code around them, binding them to our purposes.

I won't go through each and every method here but just pick out the highlights. The Add method (Listing 3) generates a new unique file (using the Win32 GetTempFileName routine) as the internal name of the cache item, creates a new cache item object, adds it to the cache (using fcAddItem), and then copies the data from the incoming stream to the internal file. If the disk usage has now grown too much, the fcCleanUp method is called to delete some old cache items.

The fcAddItem method, then, is the interesting one (Listing 4). However, as you can see, it is very simple. It first checks to see if the item already exists, if it does the new cache item is then destroyed. Otherwise, the item is added to the FItems hash table, and is inserted into the two priority queues, FExpiryQueue and FLastUsedQueue. These two return a handle that the cache item must take care of (it's through this handle that we can alter the 'priority' of an item in the queue or delete an item).

The fcCleanUp method is pretty simple too. It deletes enough cache items to bring the disk usage down within bounds; firstly by checking the expiry dates, and then by checking the last-used dates.

The Delete method called either by fcCleanUp or the outside world merely finds the cache item in the hash table, checks it isn't in use (in other words, the user of the cache has called Get but not GetComplete), and then deletes the item from the priority queues and the hash table and finally disposes of it.

The only real interesting code that's going on here (in my mind!) is in the hash table and in the priority queue. But we're merely reusing those classes and they don't enter into this design/coding discussion. For the full gory details please check out this month's disk: I've included all the relevant units. The simple example program merely caches the .PAS files in the application's folder into C:\TEMP, with the external name being the fully qualified name of the .PAS file.

➤ *Listing 5: Cleaning up cached files.*

```
procedure TaaFileCache.fcCleanUp;
var
  CacheItem : TCacheItem;
  StaticNow : TDateTime;
begin
  StaticNow := Now;
  {first check our expiry dates}
  CacheItem := FExpiryQueue.Peek;
  while (FCurDiskSize > MaxDiskSize) and (CacheItem <> nil) and
    (CacheItem.ExpiryDate < StaticNow) do begin
    Delete(CacheItem.ExternalName);
    CacheItem := FExpiryQueue.Peek;
  end;
  {if we've reduced the disk usage enough, exit}
  if (FCurDiskSize < MaxDiskSize) then
    Exit;
  {now start getting rid of old, not recently used stuff}
  CacheItem := FLastUsedQueue.Peek;
  while (FCurDiskSize > MaxDiskSize) do begin
    Delete(CacheItem.ExternalName);
    CacheItem := FLastUsedQueue.Peek;
  end;
end;
```

➤ *Listing 6: Deleting a cached file.*

```
procedure TaaFileCache.Delete(const aExternalName : string);
var
  CacheItem : TCacheItem;
  Handle    : TaaPQHandle;
begin
  {find the cache item for this external name}
  if FItems.Find(aExternalName, pointer(CacheItem)) then begin
    {if the cache item has a data stream, it's in use so we can't
     delete it: raise an exception}
    if (CacheItem.DataStream <> nil) then
      raise Exception.Create('TaaFileCache.Delete: file is in use');
    {delete the cache item from the two queues}
    Handle := CacheItem.ExpiryHandle;
    FExpiryQueue.Delete(Handle);
    Handle := CacheItem.LastUsedHandle;
    FLastUsedQueue.Delete(Handle);
    {delete the item from the hash table}
    FItems.Delete(aExternalName);
    {reduce the total disk usage}
    dec(FCurDiskSize, CacheItem.Size);
    {free the cache item}
    CacheItem.Free;
  end;
end;
```

The Conclusion

Of course, in the real world there might be other considerations of which we would have to take account. For example, it might not make sense for the caller to gather all the data for a downloaded file in a stream, just so the cache can copy that stream to a file stream. That's two copies of the same data. An enhancement would be for the caller to say 'I'm just about to download a file called X, give me a stream into which I can copy the contents.' The downloaded file would then only exist once on the user's machine. Of course, the cache would then have to have an AddComplete method to be called once the file was fully downloaded (and an AddCancel method should the download be incomplete and the connection broken).

To aid in making the browser even more responsive, the file cache could also keep files in memory as well as on disk. To this end, the file cache usually has another limit: the in-memory cache limit. My Netscape configuration has this set at 1Mb, for example.

There are many such enhancements to be made to this cache class, however I hope you got a feel for the design decisions that must be made in order to implement a fairly simple file cache class and have seen how some simple *Algorithms Alfresco* data structures could be reused.

Julian Bucknall wrote this article for the cache. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.

© Julian M Bucknall, 2000