4. CONTAINER DATABASE OBJECT ORIENTED DESIGN

We propose an Object-oriented design for the Container Database. The abstraction itself is not necessarily dependent on OO technologies nor do we claim it to be the best choice for realizing it; this choice relies solely on the fact that Object Orientation a widely used technology with several IDEs and compilers available in the market.

Based on this approach we define the three proposed interfaces as a single Object-Oriented API, where the Client Interface roughly maps the API public methods, and the Local Storage Interface and Cloud Storage Interface are both designed as internal methods, to be used by the public methods.

We'll start by defining the key classes the API uses to represent some fundamental abstractions on the proposed model. Then we enumerate the public members defined on each of these classes. Finally, we will show how these ideas fit together by giving an example of integrating this API in the Client Application code.

The next two chapters present two different implementations based on this design, being one for Windows desktop applications and another for Windows Mobile Phone applications.

4.1. Container Database Key Abstractions

The proposed API is centered upon 3 basic classes, representing 3 fundamental aspects of the CDB system. These classes are:

- Class CDB, representing the database itself;
- Class Blob, representing files of any type users would like to store, referred to as ***blobs*** in this work, as well as their metadata;
- Class DataFileToken, a means of communication between both CDB and Blob instances.

4.1.1.Class Blob

We work with the concept of Entity classes - classes that model the problem domain. They represent the type of files to be stored and their fields represent which information is necessary to describe those files. For instance, an entity class "Photo" could represent a class for storing photographs. A set of fields for this class could be "resolution", "camera type", and "author", among others.

All entity classes should derive from class Blob if they are going to have their files managed by the CDB. In this case, we would define the class as follows:

```
public class Photo : Blob {
   int resolution;
   string cameratype;
   string author; }
```

A Blob instance works as a representation of a file and its metadata. Once entity classes are created, properly derived from class Blob, Client Applications can use its instances to set their metadata and persist it. The **CDB** instance uses Blob instances to add files to the database, remove files from the database or change files metadata.

4.1.1.1.Blob serialization to allow persistence

Entity classes derived from Blob inherit two interesting properties: they know how to serialize their instance fields to a metadata stream, and they know how to use their metadata stream to bring a blob instance back to life. Putting it simply, if you ask a Blob instance to serialize itself, it will write its field contents to a data stream in a structured way. This stream can be an ordinary text file or a more sophisticated one, depending on the chosen implementation. So if you ask an Entity Class to search for the metadata of a specific file, it will be able to read its metadata file and de-serialize it into an object of its type. This inherent serialization/deserialization property is used in conjunction with class CDB for manipulating file metadata in what we call the "To Let" pattern, described in section 4.1.3.

4.1.1.2.List of Migration Policies

A Blob instance contains a ***MigrationPolicies*** field, which is a List<Policy> field used to decide at runtime which blobs are going to be stored either locally or in the Cloud.

A Policy is a delegate that takes no parameter and returns true or false. At design time, migration conditions can be specified as policies, and added to the Policy list. If one condition returns true, a file will be forcibly stored in the Cloud Container. If it returns false, an attempt to store it locally is made, but if there's no space available in the Local Container it can still be stored in the Cloud Container.

In the pseudocode example below, we create a blob instance named newSong, which contains an integer field named year. We'll use it to add to the CDB a file already provided by another service, **filename**. We've added one policy to the Policy List that returns true if the value of filed year is even. As a consequence, when the CDB instance (**db**) tries to add this blob to the storage containers, it will upload the blob straight to the Cloud Container.

```
var newSong = new Song
{
  author = "Unknown",
  year = 2000,
  size = 1000,
  title = "Millenium bug”
};

newSong.MigrationPolicies.Add(
    () => {return (newSong.year % 2 == 0); });

db.AddBlob(newSong, fileName);
```

4.1.2.Class CDB (Container Database)

Class CDB represents the database manager itself.

When a CDB instance is created, it sets aside local storage for data, metadata and indexing data for retrieval. It's the class that coordinates the Create, Read, Update and Delete actions requested by the Client Application, controlling whether a blob can be stored locally or should it overflow to the Cloud. This class can also reclaim blobs from the Cloud when there is local space available.

4.1.2.1.Mapping CDB Methods to Conceptual Interfaces

In chapter 3 we've proposed three Interfaces to compose the CDB overall abstraction: The Client Interface, the Local Storage Interface and the Cloud Storage Interface. CDB **public** methods embody the Client Interface. These public methods define internal methods that call both the Local Storage component and the Cloud Storage Component. The implementation of these internal methods will vary depending on which Local Storage Component and Cloud Storage Component will be chosen. In the Dependency graph below, we see how the CDB set of methods encompasses the three conceptual interfaces.

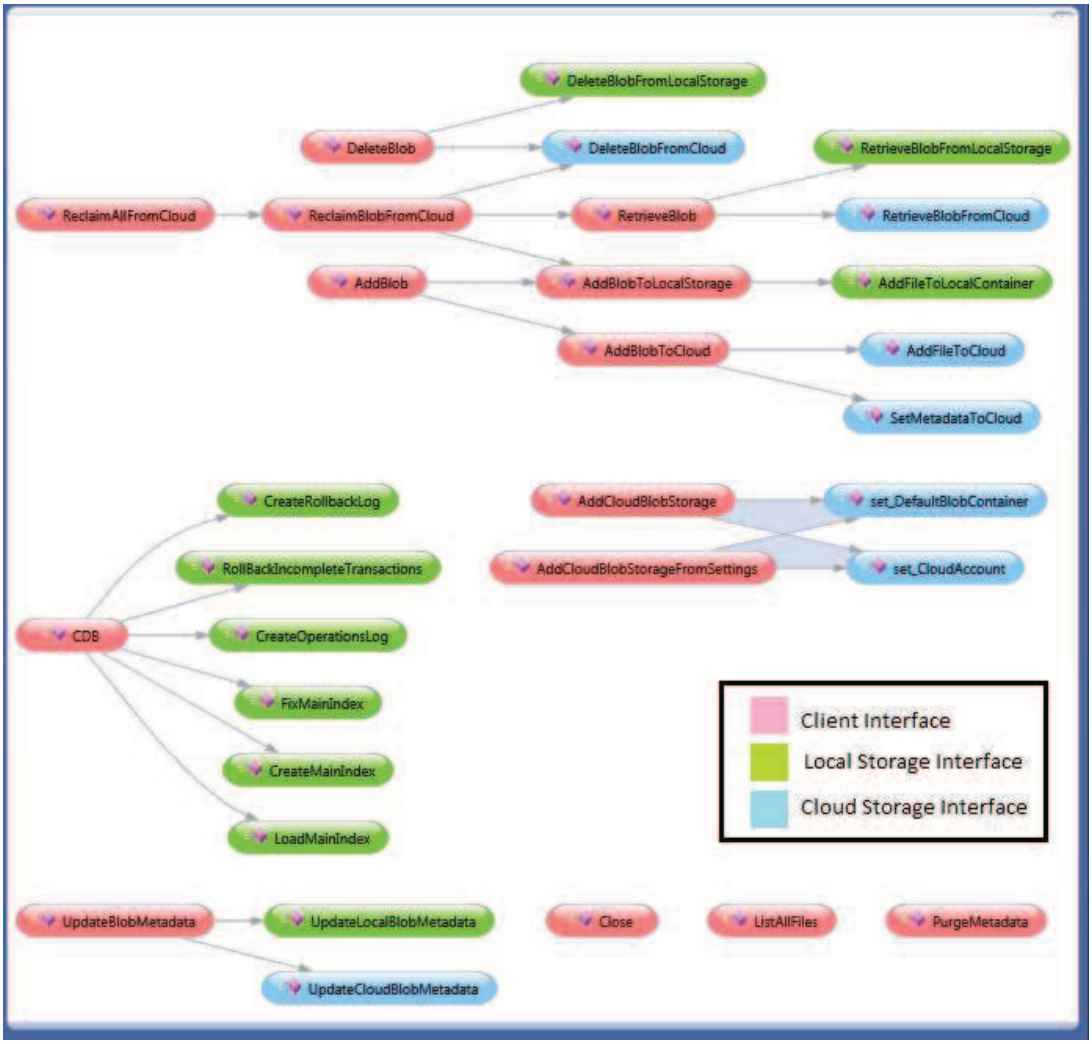

Figure 5: Dependency Graph for the CDB public interface

4.1.2.2.Main Index

A CDB instance will keep a Main Index structure where it tracks all files stored in the system, and where they are located. Each entry in the Main Index should contain:

- A unique filename as the search key;
- The file type, which corresponds to an Entity Class name;
- A reference to where the file is stored, either locally or in the Cloud (as an URL);
- A reference to where the file metadata is stored (if any).

The Main Index should be a data structure suitable for Insertion, Deletion and Search operations on O(1) time; for instance, a Map.

The Main Index should also be serializable – this is a *sine qua non* requirement for the system to fulfill its claim to be a light weight database – providing accessible, durable storage.

At CDB instance creation time, it searches for a serialized Main Index file. If it is found, the CDB instance will load its content into main memory. Before the Client Application finishes, it should ***Close*** the CDB instance, so the Main Index can be serialized to a file.

4.1.2.3.Rollback log

The CDB instance tracks the execution of its public CRUD operations to make sure they completed without leaving the database state inconsistent.

When a Client Application asks the CDB instance for any CRUD operation, it creates an entry for it in the Rollback Log – this entry is called a ***transaction***. During the whole operation execution, different instructions can alter the database state. Before executing each instruction that may alter the database state, the CDB instance adds an entry to the Rollback Log, containing the reverse operation of the instruction to be executed**,** as the first entry in the log. If the whole operation executes without errors, the transaction is removed from the log immediately after the operation is finished. If the system halts before the end of the operation, it will leave the log with a **stack** of operations needed to bring the system back to a

consistent state. When the system is re-initialized, the CDB instance checks for a rollback log and, if present, it will execute its instructions. See the pseudo-code below for an example on how the log is used in a generic operation.

```
public bool DoSomething(Blob blob)
{
  int transactionNumber = GetNextTransaction();

  AddTransaction(transactionNumber);

  PushDBAction(transactionNumber,
    "Rollback_MyFirstAction", paramlist);

  MyFirstAction();

  PushDBAction(transactionNumber,
    "Rollback_MySecondAction", paramlist);

  MySecondAction();

  MyThirdAction();

  RemoveTransaction(transactionNumber);
}
```

In the example above, let's say the application crashes before it executes method MythirdAction(). It will leave a Rollback log with entries

Rollback_MySecondAction *applicable_params*

Rollback_MyFirstAction *applicable_params*

When the Client Application restarts the CDB, it will read this file line by line and, for each line, invoke the method specified on it. These methods "Rollback_MyFirstAction" and "Rollback_MySecondAction" should be defined among the CDB methods at design time, and they will bring the database back to a consistent database state, before the method DoSomething(Blob) was attempted.

4.1.3. DataFileToken and the "To Let" Pattern

Metadata for blobs who were stored in the Cloud are too stored in the Cloud. Metadata for blobs that are stored locally are stored locally as well. To retrieve metadata for a single blob given its filename, the CDB instance will look first at its Main Index to find an entry for that file. The Main Index will tell if the

file is locally stored or if it is in the Cloud. If it is locally stored, a metadata file for that blob type is used to retrieve its metadata. If it is stored in the Cloud, an URL is inferred from the CDB settings and it is used to fetch the metadata in the Cloud.

Therefore, in both cases, the retrieve operation will be a 2-step operation.

It is important to note that the best way of storing blobs metadata depends on the following considerations:

Fast access time is a desirable characteristic for most database systems, and this one is no exception. Therefore as a design decision a CDB instance keeps a file index in memory at runtime, persists it at execution end, and loads it to memory when the CDB instance is created again. However, when we consider Locally stored files, a question that arises is what should the simplest yet functional way of persisting files metadata be, considering the overall systems CRUD operations.

The simplest idea would be to store the metadata for each file in the Main index, too. Each Main Index entry would have all metadata for its file:

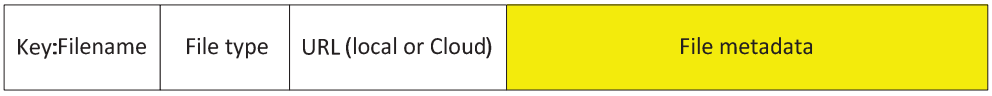

Figure 6: Not a good choice - metadata can occupy too much memory.

This idea would not work well if we plan to keep the Main Index all the time in memory, as the Main Index could grow too big. There is another disadvantage, too: we oblige the Metadata serialization/de-serialization mechanism to be coupled to the Main Index structure.

To keep the Main Index as small as possible, so it is manageable in memory all the time, we are storing metadata in separate files. Using this approach, each Main Index record, instead of keeping the metadata content itself, it keeps a "DataFileToken" instead, which indicates where to find the metadata content for the respective file:

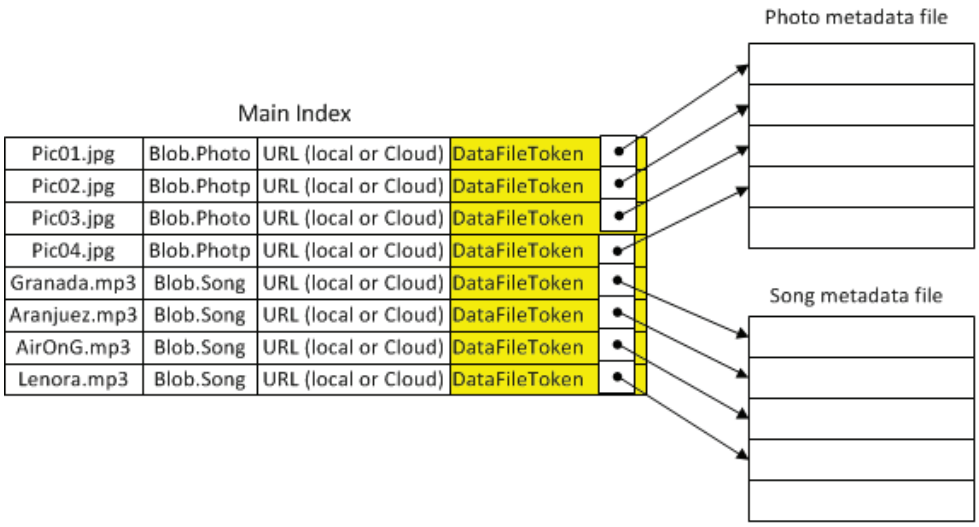

Figure 7: DataFile Tokens point to stored file metadata

The "To Let" pattern

To orchestrate whole database manipulation using the Main Index and Metadata files structures, both CDB and Blob classes work in conjunction as in the "To let" metaphor: Blobs know how to serialize and de-serialize themselves, but the CDB instance is the "owner" of both Local and Cloud Containers. So the latter assigns each Blob-derived Entity class a full path to a file their instances can use to store their metadata, as if it were renting the location to the blob instances.

Each time a blob is asked to write its metadata to its respective metadata file, it gives back to the CDB instance one DataFileToken containing the position of the written metadata in the metadata file. The CDB then adds the DataFileToken to its Main Index.

Each time the CDB needs to retrieve the metadata for a specific file, it already has a DataFileToken it can use to seek the information and uses the Blob Class to return a blob instance with that specific metadata.

4.2.Object-Oriented Public Interface

Figure 8 lists the public classes and its public methods that are part of the CDB Object-Oriented model.

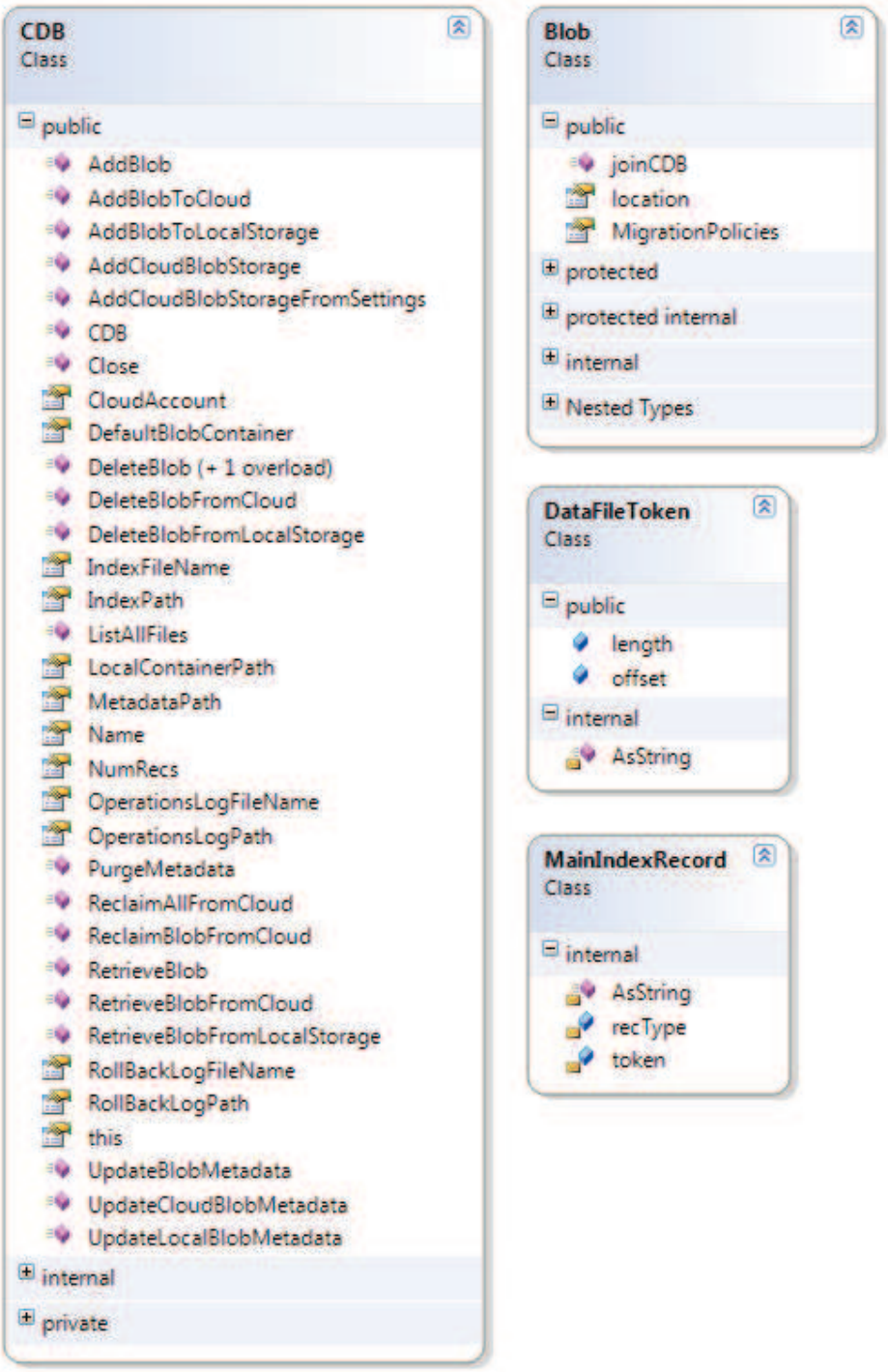

Figure 8: CDB Public interface, with its classes and methods

4.3. Integration into Client Code

In this section we illustrate the usage of the public API by providing a short example program:

1. The CDB is initialized for use. The local container, the Cloud container and the Blob types are specified.
2. A Blob instance is added to the system.
3. We query whether a file is part of the database; if it is, it is then deleted.
4. We close the CDB instance so the Main Index is serialized.

```
public class Song : Blob {
  public int size;
  public string author;
  public string title;
  public int year;
}

class MyApplication {
  static string toStore= "DieMainacht.mp3";
  static string toSearch= "FürElise.mp3";

  void Main()
  {
        1.
    CDB db = new CDB(
        "SongContainer", @"Data", @"MetaData", @"Index");
    db.AddCloudStorageFromSettings();
    Blob.joinCDB(db, typeof(Song));

        2.
    Song newSong = new Song
    {
      size = 1024,
      author = "Johannes Brahms",
      title = "Die Mainacht",
      year = 1850
    };
    db.Add(newSong, toStore);

        3.
    Song currentSong = db[toSearch];
    if (currentSong != null){ db.Delete(currentSong); }

        4.
    db.Close();
    }
}
```