ID Engine Mid Level Architecture

Author: Kevin Guerra.

**Scope**

This document describes the ID engine architecture in a high to mid-level view.

This explains how the ID Engine is organized, the different modules, how they inter-relate, and how they operate. For a detailed overview refer to the Doxygen documentation which is generated from the code.

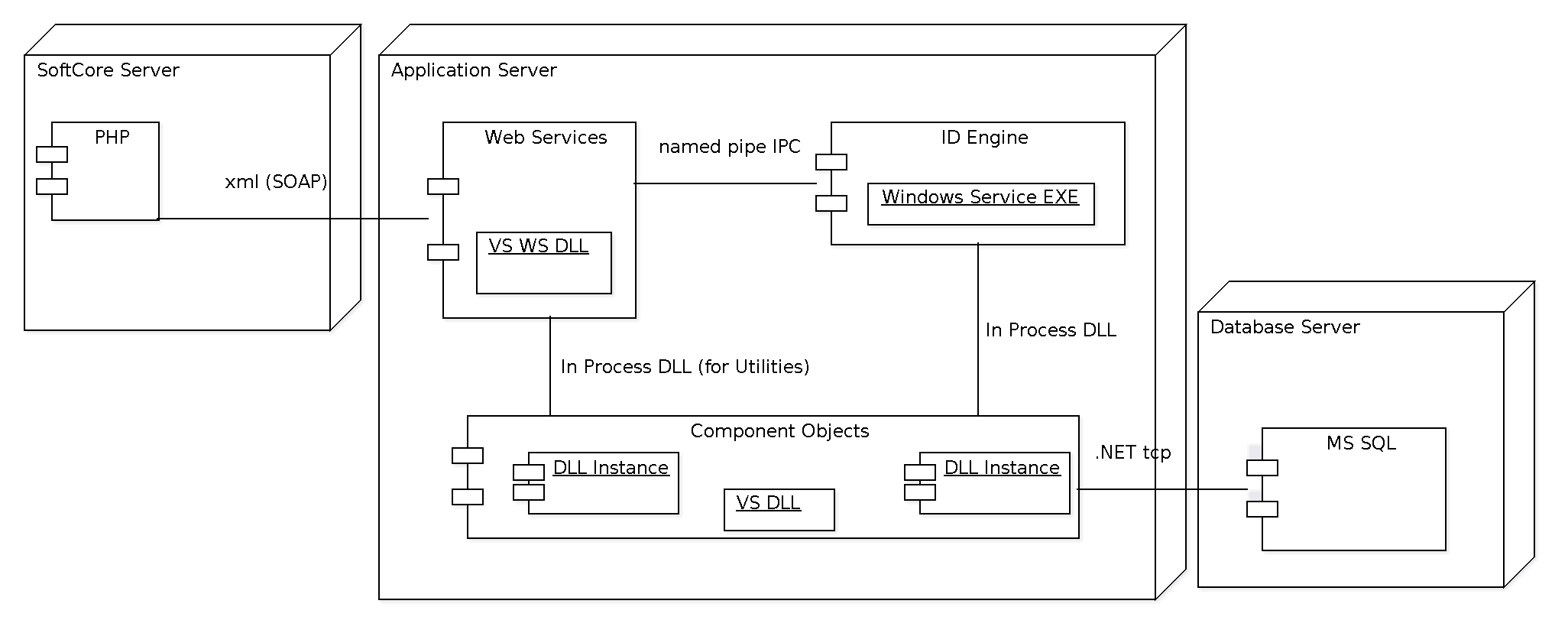
In this document we will present a fairly static view of the architecture, so that we can illustrate the different parts of the system and how they relate. We will have another document that will illustrate the system in action, with activity diagrams in UML style and uses some cases.

It was necessary to re-engineer the ID engine in order to provide the service level required and for other reasons as well, such as ease of maintenance.

First let's look at how the differences pieces fit together. The following is a deployment diagram. We can see that there are 3 nodes, where each node is a computer. The first node is a linux machine which hosts php server pages using the apache2 server. This node uses XML SOAP messages to communicate with the Application Server. The same node uses either JSON or XML to communicate with the phone app.

The Application Server communicates with the Database Server using a standard .NET connection using WCF (Windows Communications Foundation). This second node consists of two parts. The Web Services DLL which is a Visual Studio project, most of this code is generated by the code generator, while the rest is custom utility code to provide diverse functions. This first component uses the component DLL which is shared with the ID engine but only in that it provides some utility functions and some enumeration types which are auto-generated.

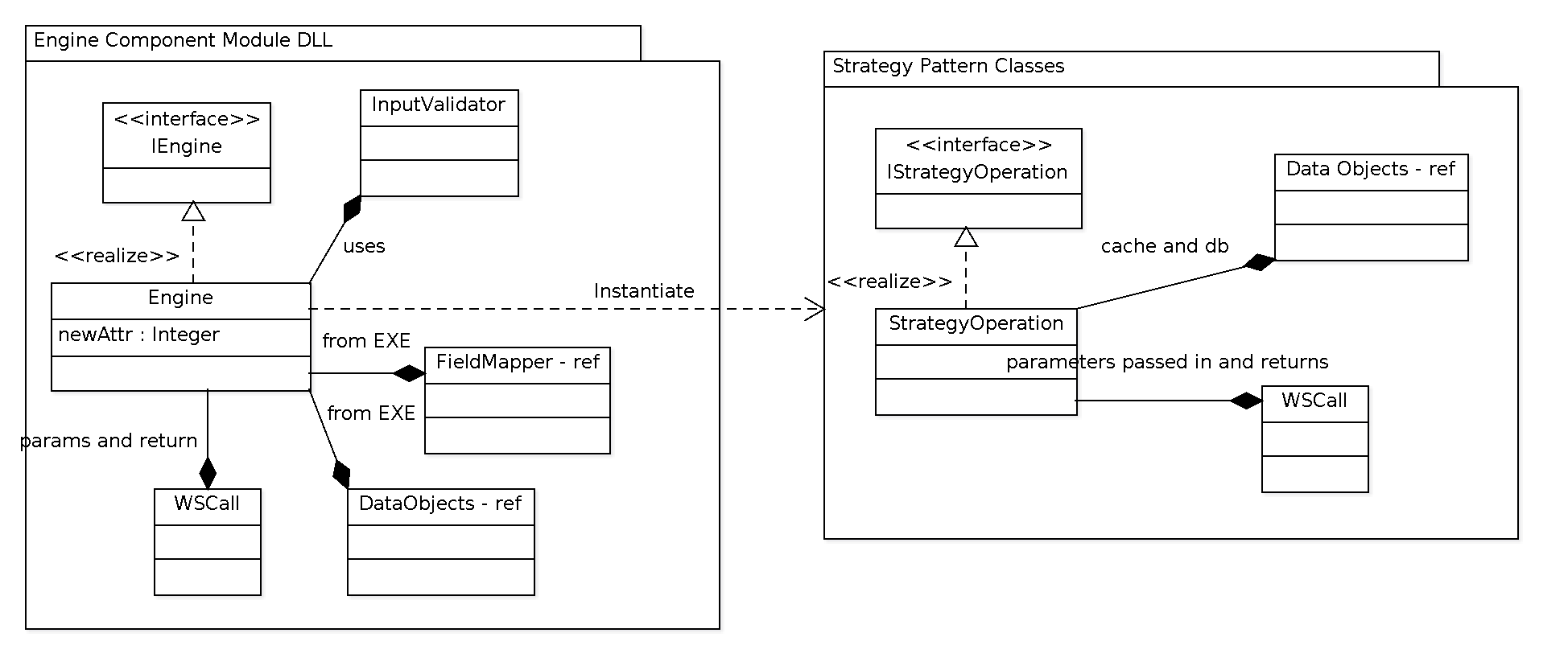
The WS DLL receives calls from SoftCore or from a web browser and wraps those parameters using the PolyModel pattern, it instantiates a named pipe client connection to the named pipe host which is the Windows Services engine. Upon completion of this call, it unpacks the string which is most cases is xml content and converts it appropriately depending on the WS call return type.

The second component is the ID engine itself which consists of two parts. The first module is a Visual Studio Project which produces an executable binary file that is a Windows Service, actually there is yet another such project which produces a regular executable with the exact same functionality which is used during testing, as it is easier to debug.

This executable simply initializes the named pipe host so that it can accept calls from the Web Services DLL. It initializes the cache and loads it from database using the component DLL which will be explained in more detail later on. It's purpose is to remain a running application while the server is running. The following graph shows the classes used.

# A description...

The second part of this component is another Visual Studio project that produces a DLL, and this is the most important part of the application and the main focus of this document.



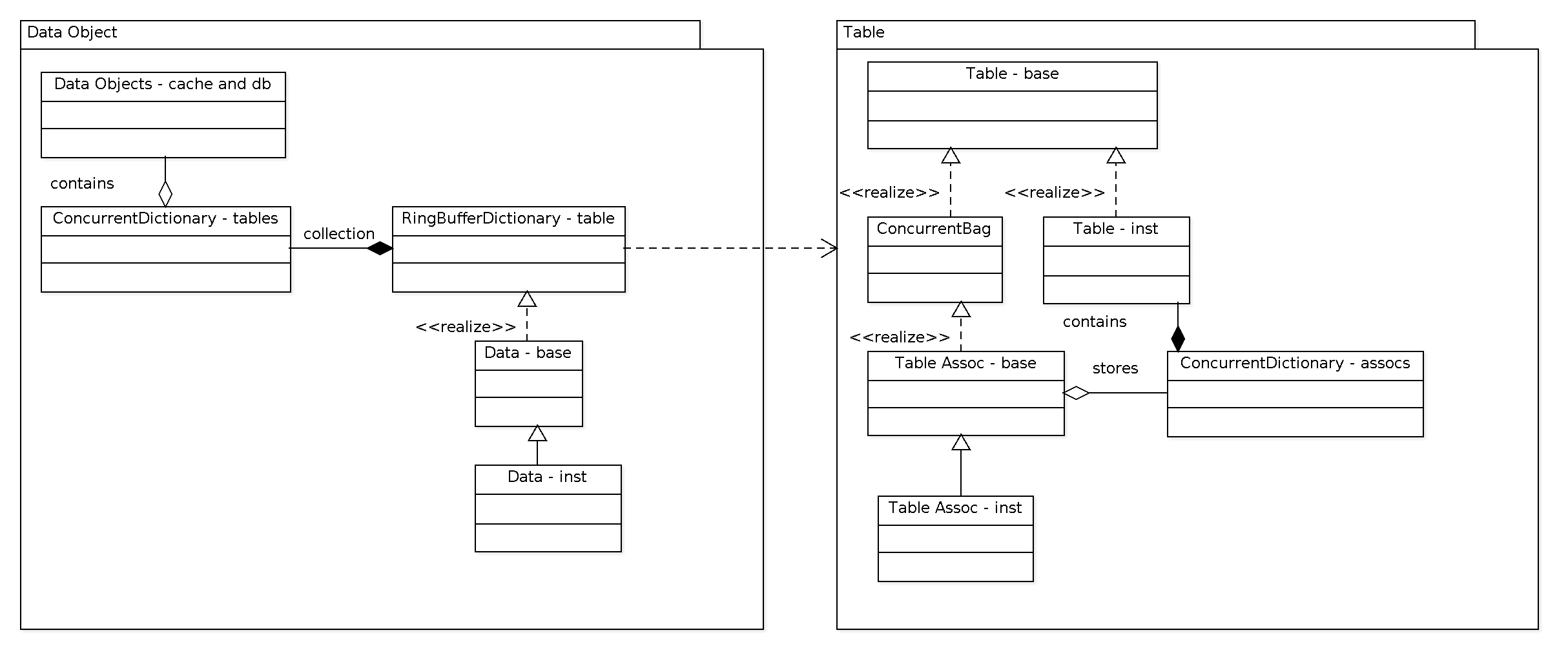
In the above graphic we see two logically grouped modules, (there is only logical grouping since they are part of the same DLL code, and as you can see they also share some common classes.) In the first module we can see the reactor part of it which is the part that contains the listening named pipe, it handles concurrency by creating a task for each call that comes in. It is responsible for performing parameter validation using the field mapper to know what gets validated and how, it uses a validation class that in turn uses regular expressions that are stored in the app.config file. When a call completes the result is sent back via WSCall to the caller via the IPC mechanism which in this case is a named pipe. This module also contains references to the data objects which is what contains the cache and data base access objects, and a reference to the field mapper, as explained above these two references are created by the executable that calls the DLL, but most of the interesting work happens in this DLL.

The next module illustrates how the data is organized internally, how it is cached and how it is created, saved, retrieved and deleted from the database.

The DataObject is a class that contains all data, this data is stored in a dictionary who's key is a table type. The table types are tables that are not associations, because association tables 'live' inside the 'real' tables. This, dictionary then contains all data, retrieved by table type, and they value retrieved via the table type key is yet, another dictionary object. The relationship is quite complex once you drill down to the database table level, but by studying the illustration below, you can understand this more easily.

Once you obtain a value from this dictionary, you now have a group of data objects. This group, is a dictionary who's key is a record ID, and in this system 'all' records are identifiable via a database record ID, because we use a relational database model and table normalization. It follows that this object is of a particular type, for example a profile record.

At this level, the data record is not simply an entry in the profile table, but 'all' its associations that are, 'directly', related. A direct relationship is one for which there is a record entry in an association table, in which the most important ID in the record directly correspond to the table type in this instance of the data object. This will become more understandable once I provide a use case scenario.



The Data Object again is a dictionary with a record ID as the key, and table object, as the value, it is here where the association tables are found using object containment, and not object derivation, because a table object can have so many association types. Let's keep in mind that at this level we are talking about types of associations, meaning, what tables 'can' be associated. However, an actual association is an actual record entry in the association table, of course, I'll provide an example later on in this document.

The association objects are not dictionaries but bags. Why? Because, it's record ID for this association is implicit and is the same as the record ID of the object where it is contained.

Let's walk through a concrete example so we can better understand how this relationships are represented in the code and the above illustrations.

Let's take for example the most complex object in the system, incidentally this most complex object is the one that best illustrates. A profile object, we are using the work object in an abstract sense and in the app it is the code and structures necessary to make this object possible to exist. This profile object has so many associations but I'll only talk about the minimum. A profile contains a name, address, etc... but these objects (or db records) are found in separate tables. In this case, we need a many to many relationship. This means that we have many profiles and a profile can be associated with many addresses, such as home address, work address, etc... For this reason, we cannot simply store an address record ID directly in the profile table, therefore, we use association tables, for this exact purpose. So, in this concrete example, let's say we have a profile ID of 1489, and we want to store a home address, and this record ends up having an ID of 38547, so, we create an association by adding a new record to assocProfile\_Address, in which the new record will contain (IDProfile=1489 and IDAddress=38547).

A Data object then is a concrete record who's key is its record ID. It contains a dictionary with table types as key, and the complete group of associations that are available. Then, it also contains a Bag, of each and everyone of its associations. These bags are contained in the dictionary mentioned here in this paragraph. This makes possible for the code to load all associations in a uniform manner across all object types.

All data objects derive from the table object and are treated the same when it comes to data access, because they all are tables in the database. All data is processed through stored procedures, but these are not all the same, because the association tables do not have auto increment record Ids.

In the new architecture these details are completely irrelevant at the mid level, these details are handled by the 'plumbing' classes and modules uniformly, so that the programmer can focus on productivity using higher level constructs. I will give an example of this during a concrete web service call, and how the data flows in the system.

**Class Organization**

The following is a list of most files in the Component DLL project and what classes they contain, not comprehensively, I'm leaving out some details in order to keep complexity at a minimum. For these details please refer to Doxygen documents.

1. WSCall.cs: parameter and return value wrapper.
2. Utility.cs: diverse helper code.
3. ValidateInput.cs, InputValidator.cs: validate parameters using field mapper and regular expressions stored in app.config file.
4. JakeKnowsEngine.cs, IjakeKnowsEngine.cs: Initialize app, reactor pattern (named pipe reactor), concurrency management.
5. RingBufferDictionary.cs: A concurrent dictionary that provides a configurable caching mechanism, per table and will recycle records that haven't been used for a while. For example, let's say we want to cache maximum of 100 profiles, when we are processing a call, if the profile is not in cache, we retrieve from the database in store here, but if the count exceeds 100, we delete one or more records that have not been used for a while so we only keep the most recently used data.
6. ContextStrategy.cs, (AUTO-GENERATED) StrategyXXX(strategy\_name).cs: The former contains a list of all 'strategy pattern' classes that are serviceable. All the rest, each corresponds to exactly one strategy. A strategy 'is' a web service call. I will explain this in the data flow. The strategy classes receive the unwrapped parameters, do the actual business logic. NOTE: The business logic is the part that currently needs more development. This architecture is a viable solution because it's been proven to work and the Sony Job Application Phone app. In this application the business logic is handled via higher level objects that encapsulate functionality at a higher level. For example the profile object has functions to deal with it's associations by function name and you can associate an applied for job as this example illustrates: profile.AddJob(jobID), or something like that. There should be another document that explains this in more detail but because this is the part that has not been developed in the ID engine, I'll not go in detail here, just suffice it to know that it will work in a similar way as the Sony app.
7. Objects, a directory with the following files: Contact.cs, Device.cs, DeviceContact.cs and Profile.cs: As mentioned above, this are higher level objects that are under development and are used in a higher level kind of way to compose the business logic required to accomplish a web service call.
8. Database, a directory with the following files:

a) (AUTO-GENERATED) FieldMapper.cs : classes InputFieldInfo, FieldsDictionary, plus support enumerations and other code. This module deals with how the system or engine is to deal with input parameters. It is used in the strategy classes to do type conversion and recordID to Data and Data to recordID mapping. An example of this is when for instance, you have a web service call with a parameter called FirstName, which is a string, this data is found in a table called tblPerson, and this table contains columns, IDPerson, Person, TYPerson (type of person, firstName, lastName, etc...) As you might suspect we identify data via its recordID, since we are using a relational database, therefore we must find it's recordID. It is this FieldMapper object that helps us to do so, this object contains auto-generated code and it is initialized with data from the database so that it 'knows' based on a table type and column name, where exactly to get this recordID, it knows where the cache is and it instructs the cache to get us the record weather it is found in the cache or else retrieve it from the database, this happens transparently to the strategy class, we don't care where to get it as long as we get it. This object is used in several places too, as explained before, it is used by the reactor to know what to validate and how, and what to convert and how.

b) (AUTO-GENERATED) ObjectsContainer.cs: class DataObjects, enumeration of all table types, Dictionary with all main table types, the key is a table type and the value is a CData object, each entry is initialized with a maximum number of records that varies depending on type, and the delete threshold or how many records to delete at a time, when the cache is full, this prevents these operations from happening to many times, hindering performance.

c) (AUTO-GENERATED) ObjectsBase.cs: class DataObjects, manages cache and database access in an abstract way. But also, the contained objects know how to do the same, actually this is because the 'real' operations happen in the lower level object and this class simply calls their functions, so they are dependent classes that are tightly coupled for efficiency. It is also, incidentally easy to maintain since the dependencies are self contained here and shouldn't leak out to higher level constructs. --- class CData inherits from RingBufferDictionary<CTable>, this class actually invokes the data access operations contained in the table objects, it loads associations, does data to record ID mapping, or rather translates transparently. Adds associations, creates, deletes, updates and retrieves objects. Note that this object is NOT an instance of a record, but rather a container of those instances and provides access via a dictionary and maps the key/value pair via recordID to the actual CTable object.

d) (AUTO-GENERATED) Objects.cs: This file is completely auto-generated and does absolutely nothing other than specify what CData objects will be available. Please understand, that again a CData object is a container of key/value (recordID, CTable) objects. This class also allows us to specify how much data we will cache the delete threshold. Note that these objects can only be, main tables, 'not' association tables, those are handled internally by their respective owners.

e) TablesBase.cs: classes CTable and CTableAssociation, the later deriving from ConcurrentBag<CTable>. These classes provide initialization and CRUD database operations. They undertand internally and only they know what stored procedures to call and how, as to the when, that is handled by the cache manager at the higher level and other objects according to need. When a CRUD operation needs to take place the caller passes in a command object to be filled with the stored procedure name and the parameter types and values needed. Then the caller which holds a reference to a connection object will invoke the database operation. Once the database record is retrieved a record object will be provided to the CTable object so that it can be used to initialize the object, this is an example of a record retrieval. For a record creation a similar pattern is followed and the stored procedure will return the record ID to be used during initialization. Delete follows the same pattern, but the caller will be responsible for the actual object deletion, this is also a lower level implementation which does not hinder the higher level code with these kinds of nuisances.

f) (AUTO-GENERATED) TablesAssociations.cs: Nothing interesting to see here, it is completely generated and holds instantiations of association table objects that are derived from the above CTableAssociation.

g) AUTO-GENERATED) Tables.cs: Same as above, completely auto-generated, but this one is more interesting. As you may gather an association table, also 'is', a table. So we will find their actual data definitions here. This file contains data definitions that map directly as they exist in the database schema. The columns or fields are accessed through class attributes, but in reality they are functions that cause a state change in the object if an attribute is updated. For example, during record creation or retrieval, the object state is clean, meaning that if you invoke a save operation, it will be ignored, since there is nothing to save. However, if you assign a value to an attribute AND the value is actually different from it, it will then change the object state to dirty. So the save invocation in this case will cause a database save operation to happen. By this mechanism, we are in effect using accessor methods for the data. This object also contains all association objects for the table, of course only in the case of it being a 'main' table. And again these assoc objects are ConcurrentBags, or rather CTableAssociation objects which they derive from. These association objects are also stored in a dictionary, so that save operations for all related objects can happen in two loops, one within the other, in an object type agnostic kind of way.

h) TablesComplement.cs and TablesFunctions.cs: classes defined here, can be any class that provides additional code for the classes of generated code, so that the future code regeneration do not affect our custom code. For example, if I need to have a function that only applies to a particular type of table, for instance, tblDevice, I would put here, so that these functions don't get clobbered during code regeneration. The functions or attributes defined here complement and provide additional custom functionality to our auto-generated Ctable and CTableAssociation classes.

**Data Flow**

1. A web service call is invoked from a phone app, or a web application, or browser, it reaches our SoftCore server. This server knows how and where to route the call, it knows what server address and port and if necessary can do protocol conversion between JSON and XML SOAP messages.
2. The web service call resolves to our Web Service DLL IIS Web Application hosted in our Application Server. The auto-generated wrapper code will wrap the parameters, create a new named pipe client object per call to transmit the wrapped parameters and receive the result, which can simply be any common data type, a string or an xml document. Upon completion it will relay the returned result object according to the WS call return type to the caller, in our case SoftCore.
3. The web service call parameters arrive via a DataSet object as the wrapper, to our ID engine, which is in the Component Object DLL. The reactor will extract the web service call type from the wrapped object and will then figure out which Strategy class object to forward it to, or rather it will instantiate a new object of the appropriate strategy type and will send the parameters via a WSCall object, along with a DataObjects reference that contains all cache. This module will then instantiate the business objects as necessary according to the type of web service. These objects will do the actual work necessary, since at this point we have all we need for this to happen, namely parameters, and access to the data. The business objects is what needs to be developed on a per web service call basis. We'll go into detail later. After the work is done, the result is returned via the same WSCall object, which the reactor will break apart and only send back that which is needed, mostly only the result and the parameters that were passed in initially will be removed since they are no longer necessary to the caller.

This explains how the 'plumbing' in the new architecture works, how the data is accessed, some of the design patterns used, how the data flows, the different modules and their locations.

The actual business objects to be developed vary greatly depending on the call, depending how much data is needed, what data sets, or is it all data needed, this is an important issue since some of the cached tables only contain a set of the data, since hosting all data in memory might not be possible or even desirable. In the current (old) architecture, a lot of the business logic is found in the database and stored procedures, for this reason, there is an incredible number of stored procedures there. In the new architecture, this is no longer necessary, stored procedures are provided for CRUD operations uniformly and a few extra sets of these for some variations of the same. The reason for this is that we have a good portion of the data in memory that can be accessed very fast via the dictionaries, so for a lot of the web service calls it will only be necessary to locate the data to be operated on, make the necessary changes and update the underlying data in the database when and if necessary. As you may have gathered by now, the table associations are contained in the 'main' tables, so that when you have located a main type of table, you already have all its associations at your disposal, for this reason a lot of stored procedures to retrieve associated data are no longer necessary.

However, there is a significant number of processes that, due to the nature of this application, provide a challenge during data access. The nature of this application, since it has potentially an incredible number of connections per node, here I'm using the term node in the context of a network, and in this case it happens to be a social network. This network, just like neurons in our brains, have these nodes that connect to other nodes in seemingly unpredictable ways. This makes data access through our current architecture quite challenging for certain operations, since as explained above, we do not cache all data. So, what's the solution? For this situation, it becomes necessary to have extra stored procedures only for those operations that warrant it.

An example of the above situation is a very simple case to understand in the context of how the current ID engine works. Let's say that we hold 1000 profiles in cache, and my own profile, let's say is connected, or associated in some way, or by virtue of having those people as contacts in my phone. Of those 1000 profiles, only 40 of them are users in the system that have recently used the app, so they exist in cache, but my complete contact list is composed of 100 profiles. This means that, 60 of those profiles are not in the cache and exist only in the database. Of course we don't want to load those 60 profiles, since this would defeat the purpose of having a cache in the first place and besides we would run the risk of running out of memory very quickly.

The solution, would then depend on what we're trying to do. For example, a web service call is getPhonesImIn() which returns a list of those user profiles whose phones contain my information as a contact. So, in this case what we want to do would depend on weather we want to obtains a simple list of those profile ID's or if we need the actual data, such as names, email addresses, phone numbers, etc.. and depending on which we would have to write a stored procedure that retrieves this information efficiency and according to need.

So, it stands to reason that we need to have a set of stored procedures that we would plug in to our architecture for these kind of situations. At the same time, once we see more of these situations, we might be able to abstract some of the functionality and perhaps even find yet another use for our code generator to be tasked with producing such code and stored procedures.

The Sony application provides a way for us to understand how this system works currently. But it is a much simpler application in that it contains much fewer tables, but uses the exact same mechanism explained in this document. In that same app, you can see from the code how several classes provide business object functionality at a higher level, the same as we expect to provide here. For example, we have a Profile class that handles data retrieval for that data type, it also contains other classes to handle the other tables. So that when you are writing code for a particular strategy (web service call) you deal with operations at a much higher level, and you don't care at that level if the data is in cache or has to be retrieved, or ID to data mapping, or validation, or other operations what make an architecture a pleasure or the lack thereof a nightmare to work with.

In the following section we will explore some of the features of this architecture that is already provided, noting that some are still under development and need completion and/or testing.

1. Automatic input validation, conversion and mapping of ID to data and vice versa.
2. Always on engine, avoid costly initialization and de-alocation of objects.
3. Web Services DLL is hosted in IIS. This is a counter feature really, once we figure out how, the engine should provide its own hosting, avoiding having to wrap the parameters and named pipe IPC. Work was done, but proved unreliable.
4. Data access patterns, including cache are abstracted from higher level coding.
5. Multi-threading and multi-tasking approach to concurrency. This is handled transparently in the reactor and the .NET framework decides the best optimization of the number of threads that it allows the system to create, preventing very costly overhead of going over the optimal number of threads, depending on how many cores the server has.
6. Asynchronous operations, so that the caller can be released early on. For example, if you are done with the call and all you need to wait to happen is the saving of data to database, there is no need to keep the client waiting for this, it's just rude. So we free the client early and fire up a new task to complete the save operation.
7. As complex as this system may seem, there really aren't a lot of moving parts, that is the beauty of it. It's simple, and it's even simpler to code at a higher level. If you compare what we have here to the former system, you will find that the same functionality is accomplished in, my guess so far is, about 10% of the code or less. Look, I know it's not finished, specially the business objects part. But remember that we do have the Sony app, that at least can give us a clue as to what to expect. No, I don't mean the Taleo part of it. Just the architecture. For example, we found, or actually I found, since I was the only one updating code, that during our debugging sessions, when this bugs occurred it was due to higher level code not complying with the specification, or in most cases changes to the specification, or additions, and then broke other things, etc.. However, we were coding at a higher level, nothing to do here about database, cache, or other lower level constructs, that worked for the most part as expected.
8. Great care is taken to avoid hard coding data and values that belong in configuration files, or database tables.
9. Extensive logging capability, using log4net which is a highly configurable component, and it's the best performant one available. Tracing ability is also available should the need arise. The debug version of the product instruct the compiler to avoid including certain type of logging so as to make the application more efficient.
10. Adaptability, and ability to evolve. Oftentimes new features or changes require changes in database schema, since this application is heavily dependent on data. No worries, all you have to do is change your data, or if you add new tables or columns update the tables that are used by the code generator, re run the code generator, re import the code and recompile.
11. Data Reflection. The reason why the code was kept to a small size, or at least one of them, is that we rely on data reflection. This is a technique by which we infer structure from data type, that is, if we can know the type of data we need to operate on, we can know, or should know (in the code) what operations are supported. An example of this is the technique I invented for mapping data to ID. Let's say, we get a parameter called firstName, from that I already know that this is part of the tblPerson, do I need to know this, actually no, our FieldMapper object has this information, so the DataObject already has a reference to this, all I know is the parameter name, so I pass in 'firstName', the field mapper does a look up and finds that it is part of the table tblPerson, and it knows that the name of the data column is 'not', firstName, but Person, and it also knows that the record ID is not named IDFirstName but IDPerson. The field mapper returns to the DataObject object the table type, it is then used internally to retrieve the dictionary for tblPerson, and then we simply do a retrieval and we will either have a cache hit, or we'll wait until it is retrieved from the database, then we will obtain a IDPerson record ID integer number, this in turn is used in the strategy file to instantiate for example a Person object that provides higher level functionality for this data type. Similar techniques are also used so that a good number of the generated files are simply type information for the instantiation of those objects, while the actual code for them resides in a common base class shared among similar types.