

# YAPL - Yet Another Persistence Layer

(V1.0)

Taras Shymbra

## Abstract

The design ideas and concepts of a generic persistence framework YAPL are described in this document. The design YAPL is largely based on the Scott Ambler's persistence layer design [AMPL] which was revisited from the C++ point of view to leverage the power of generic programming as well as modern design techniques described by Andrei Alexandrescu [AAMCD].

The YAPL focuses mainly on introducing the *persistency* to existing applications which may not be easily re-architected to support object persistence. The YAPL enables smooth systems extension by generating *persistent types* for domain classes and using them for implementation of the persistency related business logic.

Applying the described design ideas and concepts, the YAPL developers can implement its core functionality as platform independent software using STL and the Loki library. The only part that might require requires some porting efforts is the support of concrete data management systems.

## Introduction

Object persistence is the ability of an object to survive termination of the process it was created in and to retain the values of its properties between runs of a program. For this, the object's data must be stored in some external data storage (*persistent storage*) managed by some data management implementation (*persistence mechanism*) and retrieved from it later. Due to variety of *persistence mechanisms* (databases, file systems etc.), data representation formats and data access APIs there should be an intermediary layer that will provide the generic *persistent storage* interface and uniform data query abstractions. This layer often referred to as a persistence layer will be represented by the persistence framework and will serve for transparent programmatic access and to any kind of *persistent storage*. It will fully encapsulate access to *persisted data* from your business logic.

The persistence framework YAPL is a software component that implements the persistence layer functionality. It is also class package that provides infrastructure types for transparent transformation of a business type into the dependent *persistent type* that can be saved/retrieved/deleted from different *persistent storages*. The metadata, which models the underlying data schema, is represented by the runtime object structure that consists of mapping objects. This runtime mapping infrastructure also defines relations between *persistent types* and enables automatic relationships management and referential integrity enforcement.

The design of the YAPL is geared toward the concept of *transparent object persistence*. The transparency is about making domain objects persistent without extending or modifying their

classes. It is especially crucial for large-scale software systems that consist of numerous components especially when persistency should be added to an existing application and changes to the legacy object schemas should be minimal.

In this document we will concentrate on the YAPL’s design issues and will dismiss the object-to-relational mapping topics.

### Working Example

As the working example, let’s consider a simple conceptual data model (can be easily mapped to the relational physical schema) of a classical personnel management information system. For simplicity, we model only three entities and relationships among them. Figure 1 represents the ER-diagram of our data model.

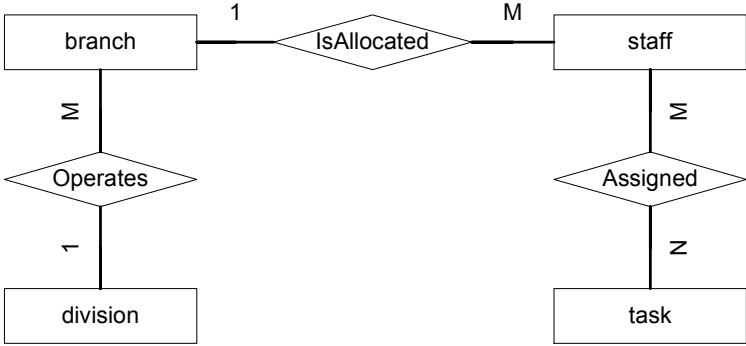


Figure 1. The ER-diagram of the example application.

The logical data model in figure 1 is then transformed into physical data model for relational database management systems. Figure 2 depicts this physical schema according to the UML data modeling profile [AMDMP].

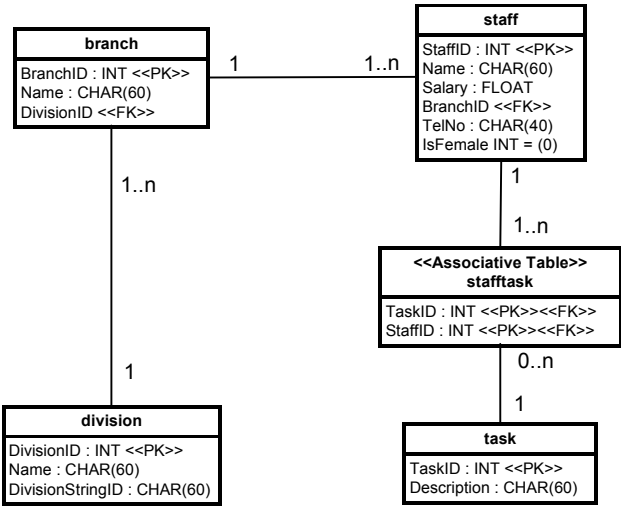


Figure 2. The UML model of the data schema of the example application.

Following the classical approach to object-relational mapping, each entity of the data model and therefore the corresponding database table is mapped to a single class. Similarly, each

column in the database is mapped exactly to one business attribute. As a result, the object schema of our application will contain the classes depicted in figure 3. We will refer to these classes as simply domain classes.

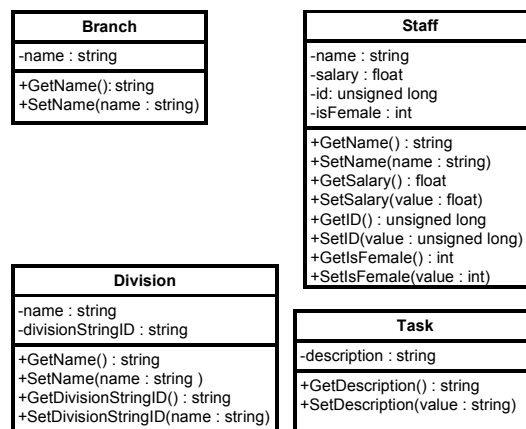


Figure 3. The class diagram of the object schema of the example application.

Note however, that there are no relationships shown in figure 3. The objects schema also omits the infrastructural attributes like primary and foreign keys. Relationships in object schemas are implemented normally by a combination of references to objects and operations. Our example application is supposed to use the YAPL, and the YAPL is responsible for relationships management we do not have to define the relationships between the domain classes.

Of course, our example application is an idealized case which is tailored to the YAPL concepts. In the real world, the YAPL has to enable persistency to existing object schemas and most likely these schemas already implement relationships. In general, the relationship management is one of the most critical YAPL issues and will be discussed in details later.

## Persistency Aspect

There are different approaches to adding persistency aspect to an application. The YAPL as a C++ framework takes the *transparent object persistence* approach.

One of the most fundamental heuristics of good C++ software is the quality of its physical design. Physical design addresses the issues surrounding the physical system entities – components as well as organizational issues such as compile-time or link-time dependencies [LCSA]. Avoiding excessive, forced and cyclic dependencies between components is the primary artefact of a sound physical design. This is where the transparent object persistence comes in. That is, it does not imply producing additional coupling throughout the whole system once some component depends on the YAPL.

To present further arguments for transparent persistence consider inheriting a domain class from some base persistency class [AMPL]. This introduces coupling to the YAPL. Other components that use this class will be likely forced to depend on the YAPL as well. Moreover, the base class impacts the interface of the domain class. It can possibly violate the semantics of already defined in the domain class methods that have the same signature as basic persistency create, retrieve, update, delete (CRUD) operations from the base class.

Components that call, for instance, “save” on the domain class to perform some domain specific work must be changed with regard to new persistency related meaning of this operation. This can finally result in tiresome maintenance burden.

The implementation of the transparency strongly depends on the programming language and the specifics of the execution runtime environment. In Java, it is relatively easy to implement. For example, the data management interface Java Data Objects (JDO) achieves transparency by applying so-called “enhancement“ of the class files [COMPJDO]. In C++ we can’t add support for transparency in this way – it is hardly possible to inject some kind of interceptors or control code directly into binaries. So, we have to elaborate another way to transparently introduce persistency into a C++ application.

Let’s consider how the YAPL’s achieves the transparent object persistence. The YAPL escapes traditional layered architecture where the persistency stuff normally resides under the domain class layer [AMPL]. It inverses the dependency between system packages as shown in figure 4.

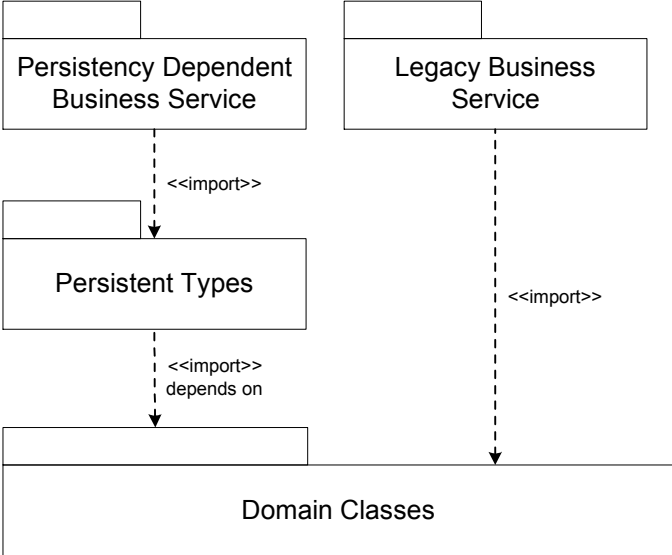


Figure 4. UML package diagram dependencies in a YAPL-based application.

In this case the *persistent types* are just another users of the domain classes. The functionality of the domain classes is *demoted* to a shared package. Demotion is the technique of moving common functionality to lower levels of the physical hierarchy to remove unnecessary dependencies and enable independent reuse [LCSA]. So, with this new architecture any number of independent business service packages can be added to the application and none of them will depend on the *persistent types*. Similarly, the *persistent types* can be modified and enhanced without affecting the domain classes and other subsystems. It is not hard to imagine that these business service packages could be reused in various combinations in other standalone applications where the persistency is not a requirement. Moreover engineers who work on persistency dependent business services can develop and test its software in complete isolation. This architecture facilitates high cohesion and low coupling by effectively detaching the persistency aspect from other system aspects. Components working directly with domain classes layer will not be able to call *persistency* related CRUD operations when they are not authorized to make objects persistent.

## Persistent Objects

Now, when we defined how the *persistent types* integrate into overall system physical architecture, let's consider how they are defined using the YAPL's classes. The class `PersistentObject` is the core class of the YAPL. This class encapsulates the behavior needed to make its instances persistent i.e. saved into one *persistent storage entry*. The relationship "depends on" between *persistent types* layer and the domain classes layer as shown in figure 4 is represented by the ISA-relationship between the `PersistentObject` and a domain class. Since the behavior of all `PersistentObject` instances is the same regardless the domain class type the `PersistentObject` should be implemented as a class template (figure 5).

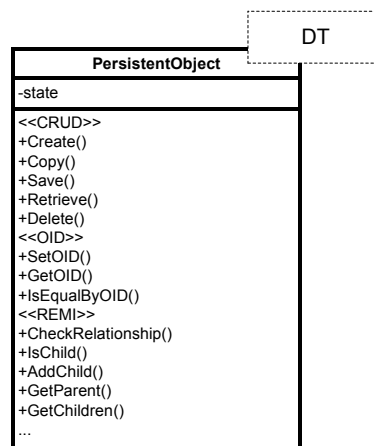


Figure 5. Class diagram of the core YAPL class `PersistentObject`.

Concrete *persistent class* is generated from `PersistentObject` class template by substituting the `PersistentObject`'s template parameter `DT` with some domain type (*base domain class*). Moreover, the *persistent class* derives from the base domain class via public inheritance. Considering our working example, the figure 6 depicts the generated *persistent class* for the domain class `Division`. We will further use the capitalized name of domain classes (e.g. `DIVISION`) to denote their *persistent types*. And, to denote a collection of *persistent objects* we will append “\_RECORDS” to their *persistent type*'s name (e.g. `DIVISION_RECORDS` is the collection of persistent objects of type `DIVISION`).

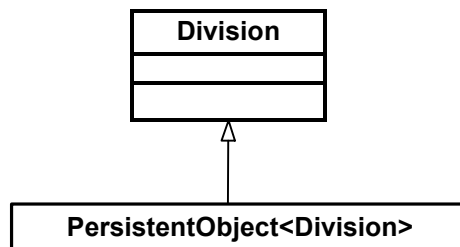


Figure 6. Class diagram of the `PersistentObject`'s instantiation.

Public inheritance in C++ represents an ISA-relationship. This is the way of making the *persistent class*' instances (further simply *persistent objects*) to be totally polymorphic to objects of the domain class (domain objects). Thus, *persistent objects* can replace related domain objects in any of its contexts [OOCAPP].

Every time we apply public inheritance we have to guarantee that the subclasses being defined will be true subtypes according to the Liskov Substitution principle (LSP). It can be defined as follows: we can always pass a pointer or reference to a derived class to a function that expects a pointer or reference to a base class [OOCAPP]. In other words, we can handle *persistent objects* in components that were initially implemented to work with domain objects. To exclude any possible overriding of inherited virtual functions that with the same signature as well as avoid name hiding (when some function is declared in the derived class, it automatically hides all functions with the same name in all direct and indirect base classes [EXCC47]) and hereby obey the LSP, `PersistentObject` has no member functions. Therefore, all operations that represent the interface of the `PersistentObject` (*persistence interface*) are declared as static member functions. These functions accept one or more *persistent objects* as parameters (*subject persistent objects*) to manipulate on.

The class `PersistentObject` enhances its base domain class by adding additional data member for *persistent object's* state management. This data member should track whether a *persistent object* is already associated with the corresponding *persistent storage entry*. If yes, the *persistent object* has the state `BOUND_TO_PSE`. Otherwise it is in the state `NOT_BOUND_TO_PSE`.

## Persistent Objects Management

The efficient management of dynamically created and shared *persistent object* instances should be implemented via intellectual wrappers that own and delete *persistent object* automatically. Moreover, to facilitate the implementation of very critical YAPL features like *persistent objects cache*, such intellectual wrappers must also support reference counting. The idea of reference counting is easily stated: track the number of smart pointers that point to the same object and when that number drops to zero, destroy the object.

In other words, YAPL clients should not work directly with pointers to *persistent object* but should instead manipulate on managed *persistent object* wrapped by smart pointers that imitate a regular pointer to a *persistent object* in syntax and some semantics, but do more. The YAPL developers should choose most transparent implementation of a smart pointer. It should be possible to change the internals of the used smart pointer or use another smart pointer implementation without incurring major modification in clients' code.

A full-fledged smart pointer that satisfies all mentioned requirements of an intellectual wrapper is the class `SmartPtr` from the Loki library. The `SmartPtr` is elaborated according to the policy-based class design and this allows us to specify elementary behavioral elements such as ownership, conversion, checking and storage policies as separate template parameters [AAMCD].

For the sake of simplicity, we will use further the term *persistent object* instead of *managed persistent object* keeping in mind that all *persistent objects* are actually managed by some kind of smart pointer.

## Persistent Objects Manipulation

The basic CRUD functionality (create, retrieve, update, delete) that any persistence framework must support CRUD functionality is accessible via the `PersistentObject`'s CRUD methods which are the part of the persistency interface (see figure 5).

Concerning *persistent object* creation, all `PersistentObject`'s constructors are private. Instead, the `PersistentObject` provides external *persistent object* construction methods. There should be at least a simple creation method `Create` that would create a new *persistent object* and a copying method `Copy` that would create a new *persistent object* and would initialize with the specified domain object of the corresponding base domain class. The YAPL should leverage the copy initialization. Copy initialization means the persistent object being created is initialized using the copy constructor and the assignment operator of the base domain class. Here we encounter one of the stipulations on domain classes: if YAPL clients want to utilize the copy initialization they must ensure that the corresponding domain classes possess accessible (i.e. public) and meaningful implementations of the assignment operator and the copy constructor. Of course, all construction methods must return *managed persistent objects*.

The CRUD methods are used to move the data between the memory and the *persistent storage*. The method `Save` checks the `PersistentObject`'s state to figure out whether a new entry must be created in the *persistent storage* or an existing one must be updated. It performs the operation of *persistent object* marshalling (simply *marshalling*). Marshalling is the conversion of a *persistent object* into a *persistent storage entry*.

The `Retrieve` initializes the specified *persistent object* with the data from the corresponding *persistent storage entry* accomplishing the *unmarshalling* operation. Unmarshalling is about initialization of a *persistent object* using the dataset fetched from a *persistent storage* according to the retrieve criteria (*raw persisted data*). It is the conversion of a *persistent storage entry* to a *persistent object*.

The method `Delete` physically deletes the *persisted* data represented by the *subject persistent object*. This method is especially interesting because it must ensure that the *persistent storage* remains in the consistent state according to the referential integrity rules (see section “Referential Integrity Enforcement”).

Among issues that affect both performance and data consistency, the update of *persistent object* properties is one of the most important one. When a `Save` is called on already persisted `PersistentObject`, the YAPL should not simply update all attributes of the corresponding *persistent storage entry*. The right way would be to consider only those native properties (i.e. properties defined in the `PersistentObject`'s base domain class) which have been changed since the last *persistent object*'s update. In other words, a YAPL application should not overwrite *persistent attributes* unless the YAPL clients' application has changed them. One of the ways how to implement such consistent *persistent attributes* update would be to apply the concept of so-called *persistent object snapshot*. A *persistent object snapshot* would be an additional object of type `DT` stored in a host `PersistentObject` instance. This object would represent the backup of the host *persistent object* since its last unmarshalling. Each time the `Save` operation is called on the host *persistent object*, the native properties of the *persistent object snapshot* can be compared with the host *persistent object*'s native properties to figure out which ones have been modified.

For proper functioning of the CRUD methods, a *persistent object* must provide the value of its persistent object identifier (POID). POID often simply called an object identifier (OID) is an

attribute (normally a large integer number) assigned to a *persistent object* to distinguish it among other *persistent objects* of the same type. OIDs are the object-oriented equivalent of keys from relational theory, columns that uniquely identify a row within a table [AMPL]. In the YAPL, OID value of a *persistent object* represents the primary key value of some *persistent storage entry* and serves as the binding medium to connect objects in memory to *persistent storage entries*. To specify the OID value, the persistency interface includes the OID-setter (`SetOID`) and the OID-getter method (`GetOID`). The OID-setter must also support the *automatic persistent object retrieval* i.e. the *subject persistent object* must be automatically unmarshaled if the *persistent storage entry* with the given OID exists in the *persistent storage*. Setting the OID value to a *persistent object* is the best way to retrieve this *persistent object* in one operation.

Since the type of OID is not known on the `PersistentObject` level, the `SetOID` should provide a functional behavior that can be called for any type of OID. Therefore, the OID-setter should be a member function template which obtains the type of identifier via its own template parameter. On the other hand, the OID-setter should provide some runtime type check and work properly only with one predefined for this *persistent class* type of the object identifier.

The definition of the OID as the unique identifier implies that there could not be more than one *persistent object* with the same identifier value. Ideally, when the OID-setter leverages the *automatic persistent object retrieval*, the YAPL should try searching in the memory for an instance with the specified identifier value and, if found, initialize the *subject persistent object* to reference this instance. The YAPL developers should implement such a *persistent object* sharing feature (it can be implemented for instance through the use of the *persistent objects cache*) to disable manipulations on multiple *persistent objects* that have identical object identifiers but different *persistent object* states. Such objects are connected to one *persistent storage entry*, because OIDs represent unique key references of *persistent storage entries*.

Without *persistent object* sharing feature, there could be two different objects with the same OID value that have different states: `BOUND_TO_PSE` and `NOT_BOUND_TO_PSE` respectively. Attempt to save the second object will fail because the `Save` function will try to create a new entry in the *persistent storage*. Since the entry with this OID already exists (the state of the first object is `BOUND_TO_PSE`) the *persistent storage* most likely will disallow to create a duplicate. YAPL clients should not track the lifetime of *persistent objects* manually, their state transitions. They should not care about different usage scenarios that lead to existence of multiple instances with the same identifier. It is responsibility of the YAPL to provide the *persistent objects* consistency. In addition, YAPL should provide a comparison method `IsEqualByOID` to test two *persistent objects* of the same type for equality in terms of the OID values.

In general, the issues like *persistent object* sharing and *persistent objects cache* are very subtle ones and must be scrutinized by YAPL developers with great attention.

## Mapping Object Schema to Data Schema

The responsibility of the `PersistentObject` class is to surrogate some domain class for persistency purposes. As was outlined above, *persistent objects* have interface identical to those of domain objects they replace. In addition, they provide also persistency interface.

The transportation of persistent objects between memory and permanent data storage requires some *persistent storage* description including the names of *persistent entities* and attributes along with attributes' types as well as information about the associations among those entities. In other words, we need description of the underlying data schema which should be available at runtime.

In the YAPL, the data schema is naturally mirrored in memory as an object structure - a *runtime data model* (RDM). On the lowest level, the RDM is the set of `AttributeMap` instances (*attribute maps*). The primary role of the `AttributeMap` class template is to represent one *persistent attribute* as it is defined in the data schema and to describe its characteristics such as name, value type and the role. The attribute value is introduced to the `AttributeMap` by one of its template parameters (e.g. `AT` as shown in figure 7).

To gather all attribute maps that belong to one *persistent entity* they are aggregated into higher-level object `ClassMap` (*class map*). The class `ClassMap` is the descriptor of one *persistent entity* and is directly associated with one `PersistentObject` (see class diagram in figure 7). Each `ClassMap` instance knows the name of the *persistent entity* it is mapped to as well as its master *persistence mechanism* (*persistence mechanisms* are described in section "Persistent Mechanisms"). In this way the runtime data model provides metadata for unambiguous mapping between a *persistent object* and a *persistent entry*, object's properties and entry attributes.

The `ClassMap` maintains the inhomogeneous collection of `AttributeMap` objects (for each possible attribute value type there will be separate `AttributeMap` instantiation). Collections in C++ are normally implemented by using STL standard containers. But one standard container instantiation can store elements only of one type. This is actually the reason why the `AttributeMap` is factored to inherit from the `AttributeMapBase` class. So, the attribute maps collection in `ClassMap` can store pointers to objects of the `AttributeMap`'s base type. The attribute maps are referenced within its host class map via *attribute handles*.

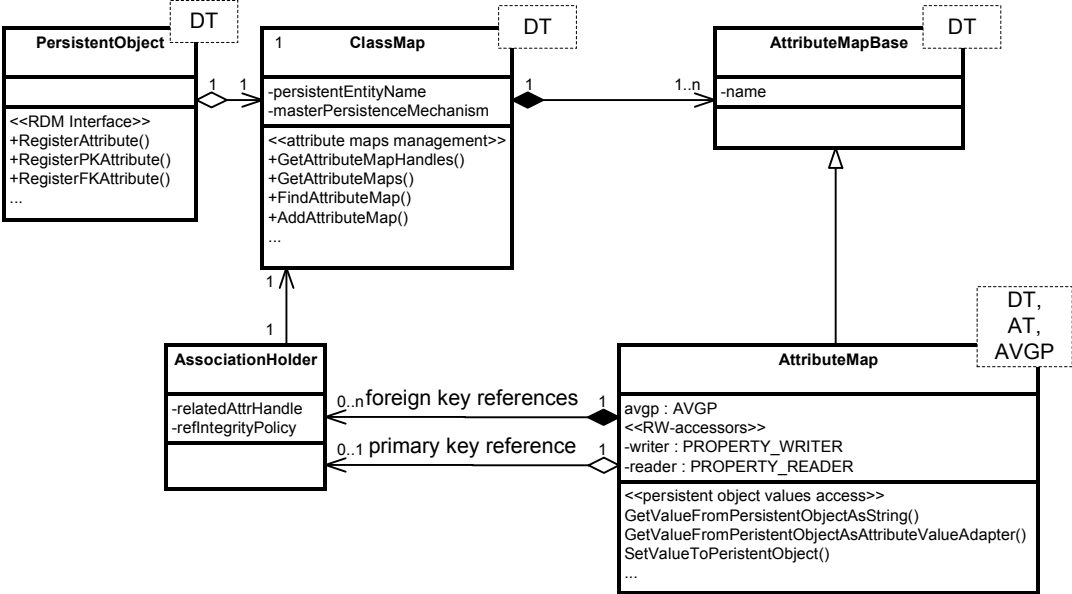


Figure 7. Class diagram of the RDM classes.

The runtime data model models the associations among *persistent storage entities* as parent/child relationships and applies the concept of primary/foreign key from the relational theory. We define a foreign key as a single data attribute that appears in a child entity and references the primary key of the parent entity. Since primary and foreign keys are elementary connectors between *persistent storage entities*, the class `AttributeMap` is the natural choice to store the associations-related information. Therefore, the `AttributeMap` has the collection of `AssociationHolder` objects to maintain foreign key references as well as a member variable of type `AssociationHolder` to reference a primary key. One `AssociationHolder` (*association holder*) instance encapsulates the description of one relationship its host attribute map is participating in. It references its `ClassMap` and holds the related attribute handle, along with referential integrity policy (specifies how to handle children when a parent is modified. See section “Referential Integrity Enforcement”). Therefore, when an `AttributeMap` represents a primary key attribute its collection of foreign key references will contain association holders of all related foreign keys. Similarly, the `AttributeMap` of a foreign key attribute will store an association holder of the parent’s primary key in its member variable for primary key reference.

Although runtime data model is a standalone object structure, the declaration of the `ClassMap` types, their creation and initialization is driven by the *persistent classes*. That is, each instantiation of the `PersistentObject` automatically generates a new class map declaration from the `ClassMap` template class and a base domain class which is used as the template argument and replaces `ClassMap`’s template parameter `DT`. There is only one instance of the declared class map type (*host class map*) and therefore it is a singleton which can be implemented using the template class `SingletonHolder` from the Loki library.

Initialization and construction of the runtime data model is performed via parts of the `PersistentObject` interface that is represented in figure 7. These higher-level methods hide the details of the runtime data model classes `ClassMap`, `AttributeMap` and `AssociationHolder` and actually add attribute maps to the host class map. So, the method `RegisterAttribute()` adds attribute maps of native properties, the method `RegisterPKAttribute()` adds an attribute map for the OID attribute. Finally, the function `RegisterFKAttribute()` is used to add an attribute map that will represent a foreign key connecting it with the attribute map that represents a primary key. This method is actually utilized to define a parent/child relationship among *persistent classes*.

The runtime data model is connected to the object model via the one-to-one association between a `PersistentObject` and a `ClassMap`. This is how *persistent classes* are mapped to the persistence storage entities. We also know, that `ClassMap` contains attribute maps that describe attributes from the data schema. Next important issue to discuss is how actually *persistent storage entry*’s attributes are mapped to the `PersistentObject`’s properties. First of all, the YAPL assumes that native properties have corresponding getters and setters. In this case, during marshalling of a *persistent object* the YAPL will be able to access its native property by requesting its getter and setter operations. We need an indirect representation of the property access member functions. Such representation would also serve as unique property descriptor and would be stored in the corresponding `AttributeMap` for attribute-to-property mapping purposes. C++ offers pointers to member functions - a facility for indirectly referring to a member of a class. Just like ordinary pointers to functions, pointers to member functions are used when a function should be called without specifying its name[STR]. This is exactly what we need for implementation of the generic marshalling code. During the creation of runtime data model we simply initialize `AttributeMap` objects with two pointers to member functions. These pointers to member functions will represent a

typical property setter and getter will be declared in `AttributeMap`. The type of the pointer to property setters will be defined as:

```
typedef void (PersistentObject<DT>::*PROPERTY_WRITER)(AT);
```

Similarly, the type of the pointer to property getters will have the following definition:

```
typedef AT (PersistentObject<DT>::*PROPERTY_READER)();
```

Since these pointers to member functions are used to read and write some native property we will call them Read/Write-accessors (*RW-accessors*). When a *persistent object* is marshalled, YAPL will combine registered RW-accessors with a pointer to the object and will be able to set and get a particular property value.

The runtime data model is the basis object structure that the YAPL execution engine uses to perform its core functionality. In other it is a director that drives operations on *persistent objects*.

## Attribute Values Handling

We outlined how native properties are mapped to the corresponding attributes in the data schema. The values of native properties are stored in domain objects (*native storage*) and the attribute maps with RW-accessors offer just the right magic to make them persistent. Moreover, the runtime data model contains attribute maps of infrastructural attributes (primary and foreign keys) for data schema relationships representation. These attributes require storage for their values. Although it is possible to map infrastructural attributes to native properties, the domain classes must be forced to introduce additional ones only for persistency purposes and this will violate the transparent object persistence concept. From the application perspective the infrastructure related information is just a kind of shadow data that must be transparently handled by the YAPL. Therefore, there should be a mechanism that allows the values of infrastructure related attributes to be maintained beyond the normal domain data. This mechanism denoted as *automatic attribute values manager* (AAVM) can be integrated directly in the `PersistentObject` or implemented as a separate class.

The essential task of the automatic attributes manager is to provide the *automatic storage* for shadow data, to manage this storage (allocation and revocation of memory) and to provide the attribute values access interface. It should associate attribute handles with attribute values. One such association entry in the manager will be denoted as a *value cell*. The number of value cells and their creation should be completely driven by the runtime data model. That is, in the initialization phase the automatic attributes manager will iterate over all attribute maps registered for the host `PersistentObject` searching for attributes that are subject to automatic management (we will call such attributes simply *automatic attributes*) and for each of them allocates a value cell.

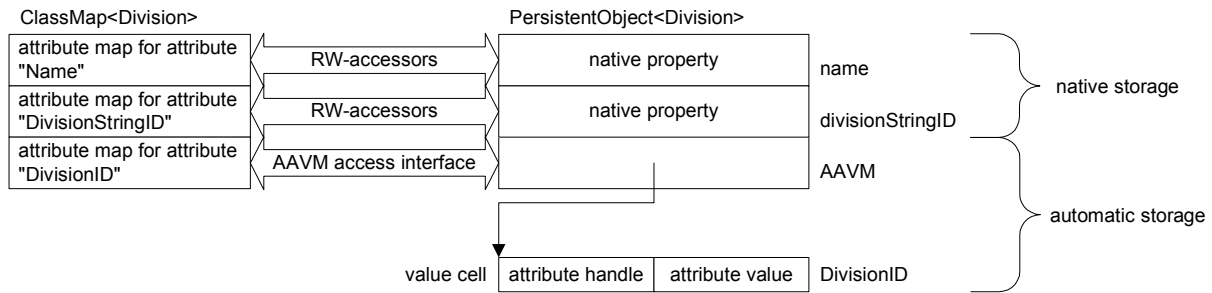


Figure 8. Schematic representation of different attribute value storages.

In general, attribute maps without RW-accessors are considered to be automatic attributes. Therefore, if infrastructural attributes are not mapped to some native properties they will be managed automatically and will become an automatic storage (see figure 8).

A value cell is a generic artifact that should store attribute values of any type in unified manner. Therefore, the YAPL introduces a class hierarchy for uniform attribute value representation. The base class of this hierarchy is `AttributeValueBase` (see figure 9). Let's first consider the `AttributeValueBase`'s subclass `AttributeValueAutoCell` which is actually used by AAVM for keeping attribute values in value cells.

In contrast to native properties that are initialized in the domain classes, the initialization of value cells should be performed by the YAPL in autonomous fashion. In simple cases it would be enough to reset the values of arithmetic types to 0 and call the default constructor of user-defined types (to-zero initialization). But in practice, the clients most likely will need to provide application specific default value generation algorithms (*automatic value generation policies* or simple AVGPs). For this purpose, YAPL allows encapsulation of such algorithms in form of policy classes that can be passed as template parameter to the `AttributeMap` class template (e.g. AVGP as shown in figure 7). YAPL clients can implement policies in various ways as long as they are statically polymorphic and respect the policy interface. The `AttributeMap` holds an instance of the specified policy type. The AAVM can pass this instance to value cells it creates, so that they can automatically initialize themselves. To be more exact, automatic value generation policy objects will be passed to the `AttributeValueAutoCell` objects. Different types of YAPL clients specific are introduced to the `AttributeValueAutoCell` class template via its template parameter AVGP. The `AttributeValueAutoCell` keeps a policy object and uses to initialize itself.

In general, value generation policies are most frequently used for OID values generation as discussed in section "Object Identifier Attributes".

The information about which storage (native or automatic storage) must be used for setting and getting values of some *persistent object* is encapsulated in the `AttributeMap` because this class manages RW-accessors. When RW-accessors are specified, they must be used to access the corresponding native properties otherwise an automatic storage must be requested. Therefore, `AttributeMap` is also responsible for `PersistentObject` values access. It is a façade that implements an interface for unified *persistent object* values access (see figure 7). This interface is widely used by the YAPL to extract and set attribute values of an `PersistentObject`. The `GetValueFromPersistentObjectAsString()` operation extracts an attribute value and converts it to textual representation (e.g. as `string` from C++ standard library). This is the final value representation and is used for building of

*persistent storage* access commands (e.g. SQL-Queries). When more sophisticated values handling is required (like type related values comparison or transportation of values between attribute maps) `GetValueFromPeristentObjectAsAttributeDataAdapter()` operation is used. It returns attribute value as an object of the `AttributeDataAdapter` class (*av adapter*). The operation `SetValueToPeristentObject()` is the single way to set a value to a *persistent object*. Since this operation accepts one `AttributeDataAdapter` object, *persistent object* values setting is performed only via attribute value adapters.

When an attribute map is the central gateway for attribute values traffic then the `AttributeDataAdapter` is the transportation mean. It is not a template class, it does not generate domain class specific and the attribute type specific attribute value adapter types. This allows objects of `AttributeDataAdapter` to travel between different persistent type domains (a persistent type domain denotes set of generated types for a base domain class including its *persistent class* along with class maps and all attribute maps).

Access to an `AttributeDataAdapter` object's attribute value is provided via multiple entry points defined as conversion operators for each supported type. There is a set of member functions `AttributeDataAdapter::operator T()`, that define a conversion from an attribute value adapter to arithmetic types [STR] as well as to standard library's type string. Conversion operators allow the generic code to access an attribute value of the required type by applying explicit conversion `static_cast<ATTRIB_TYPE>(va)` where `ATTRIB_TYPE` is an attribute type name.

How `AttributeDataAdapter` represents attribute values internally? As you might already suggest, it simply references an `AttributeValueBase` object to leverage the uniform attribute value representation. YAPL developers should consider managing of `AttributeValueBase` objects by smart pointers with reference counting. The reference counting is required for tracking multiple attribute value adapters that point to the same `AttributeValueBase` instance. The main rationale behind reference counting on the `AttributeDataAdapter` level is the value state consistency, value sharing efficiency (excessive deep copying of `AttributeValueBase` objects should be avoided).

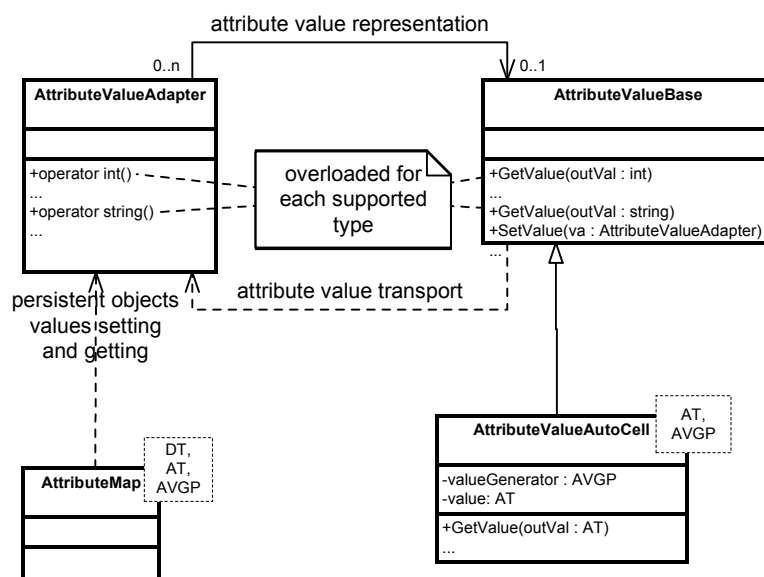


Figure 9. Class diagram of the attribute value handling classes.

Let's now revisit the `AttributeValueBase` class hierarchy. Besides value cells, concrete `AttributeValueBase` specializations are used for other attribute value representation needs. In general, their responsibility is to hold an attribute value, encapsulate the value representation specifics and provide reasonable implementation of value access operations declared in the root class `AttributeValueBase`. Value access interface is represented by the operation

`SetValue()` that has one parameter of type `AttributeValueAdapter` and by the set of virtual functions with the same name `GetValue()` overloaded for all supported YAPL types. The value writing operation `SetValue` is defined as a no-op method. The default implementation of each `GetValue` should signalize this `GetValue()` might be the wrong entry point for value access (e.g. by throwing an exception). This is actually a good way to mimic type safety at runtime. Derived attribute value specializations must provide redefinitions of `GetValue()` functions according to the value type they represent.

Combining virtual functions with names overloading is a powerful technique because it facilitates easy implementation of generic `AttributeValueBase` specializations. A specialization can be implemented as a class template with a template parameter that represents the type of attribute value it holds. The class `AttributeValueAutoCell` is an example of a generic attribute value. It defines only version of the `GetValue()` method using its template parameter `AT`. The implementation of this method will be completely generic. An `AttributeValueAutoCell` instantiation for some type will yield only one correct `GetValue()` virtual method redefinition and only for this type. Calling to `GetValue()` via a pointer to the base class and using an argument of wrong type misses this redefinition and ends up in the corresponding default version which should notify about the type mismatch.

To encapsulate the specifics of attribute values representation imposed by some *persistence mechanism* (more about it in section “Persistent Mechanisms”), YAPL developers should implement concrete `AttributeValueBase` subtype(s) for this *persistence mechanism*. That is, the common `AttributeValueBase`'s interface is the only mean to cross the boundary between the *persisted raw data* (a notion of a dataset obtained from a *persistent storage* and stored in some *persistence mechanism* specific manner) and the *persistent objects* side.

To summarize, the `AttributeValueBase` class hierarchy in collaboration with the `AttributeValueAdapter` represent the lowest level abstraction of values representation in the YAPL regardless of whether the values originate from *persistent objects* (both native and automatic storages) or from *persisted raw data*.

## Object Identifier Attributes

As was mentioned above, object identifier values are essential for numerous YAPL features like CRUD functions and the relationships management. In a runtime data model, an OID attribute is mapped to the primary key of the host *persistent storage entity*. OID attribute is the first candidate to be managed automatically, particularly when it is a surrogate attribute that has no business meaning. Like other *automatic attributes*, OIDs should be properly initialized via an automatic value generation policy.

Since, OID attributes are normally mapped to primary keys and primary keys are normally generated by *persistent mechanisms* (e.g. auto incremental primary key in relational databases) clients may not care about the OID generation issue at all and move this responsibility over to *persistent mechanisms*. YAPL developers should provide an automatic value generation policy that would not implement a generation algorithm but would simply signalize the error when requested to generate a value. YAPL should also dismiss this attribute during marshalling.

However, clients must keep in mind the peculiarities of *persistent mechanism* based OID values generation. First of all, once a *persistent object* without programmatically generated OID is persisted, it loses binding with its *persistent storage entry*. This is because there is no safe way to figure out which *persistent storage entry* was actually created to store this *persistent object*. Therefore, such *persistent object* always stays in the NOT\_BOUND\_TO\_PSE state. As a side effect, one object can be saved to different *persistent storage entries*. In some applications, driven by strong performance requirements, it is exactly the right way to go (e.g. fast generation of *persistent storage entries* by changing and persisting only one *persistent object*).

When YAPL clients definitely need custom OID generation algorithm specific value generation policies must be provided. For convenience, YAPL should provide a number of most frequently used policies like for example automatic value incrementation.

The greatest usefulness of value generation policies might be their configurability and extensibility. It might be further extended with value initialization and backup algorithms that might be also specified as policy classes. The implementation of an initialization and backup policy class may provide initial value and backup the updated value. So, for instance, such a policy can search for the maximal primary key value in the corresponding *persistent storage entry* to initialize its value. By configuring value generation policies with this value initialization and backup policy, programmatic emulation of the auto incremental primary key feature can be achieved.

In some applications OIDs can consist of a session number and a counter within the session. To use OIDs correctly the *persistent storage* needs a dedicated entity for allocating the session id's to ensure uniqueness in the OID values across the application. A value initialization and backup algorithm implemented in such application will access the session id table for OID initialization. Other ideas that come in mind for value generation policies are values generation algorithms based on some hash function etc.

## **Automatic Management of Infrastructure Attributes**

Generally speaking, infrastructural attributes like an OID attribute (a primary key) and attributes that represent foreign keys must be always automatically managed unless there are indisputable reasons to implement them as native properties and YAPL clients clearly understand the way infrastructural values are used. Automatic value generation policies which will be used for OIDs management by default should provide the *constant primary key* guarantee. The constant primary key guarantee means that once some *persistent storage entry* is created its primary key can not be changed via manipulations on OID attribute of the corresponding *persistent object*.

By keeping primary keys of *persistent storage entries* constant, YAPL avoids possible referential integrity pains. The only way to bypass the YAPL in its referential integrity efforts is to implement object identifier attribute as a native one and gain full control over OID values. There is risk that malicious client's custom management of the object identifier attribute can break referential integrity by, for example, implementing OID-getter to return values that can enforce the YAPL's marshalling mechanism to change the primary key values in the *persistent storage*. The same applies to attributes that represent foreign keys. To exclude potential referential integrity problems like references to non-existing parents the values of these attributes should not be directly changed by the application. Although there is a possibility to implement foreign key attributes as native properties, let the YAPL take care of them.

## Support for Refactoring

Developing software when system requirements constantly and rapidly change is always challenging. Implementing new requirements as well as improving the system might cause modifications of its data model. Most tough situations arise when the definition of existing entities and their attributes must be changed (i.e. attribute type or name) or when some attributes must be completely deprecated but there are some legacy applications that rely on the old data model definition. When these applications could not be modified and must remain working, the existing data model must be transparently extended. Suppose we need to change the type of some attribute from string to integer. Since we can not simply change the attribute definition, there is no choice but to add a new integer attribute. We end up with two attributes that are different representations of the same aspect. Both attributes simply hold the same value and must be always synchronized.

The YAPL allows easy application refactorings as well as data schema refactorings on the runtime data model level in any number of ways. So for example, YAPL developers can implement special `AttributeMap` subtypes that would serve as attribute proxies. Such an attribute proxy would redirect value requests (i.e. via the persistent object values access interface) to another referenced attribute map etc.

## Relationships Management

Generated in runtime, the data model object structure is an in-memory representation of an application's underlying data model. Mapping of the runtime data model to the system's object model faces the challenge of expressing the relationships between domain classes in terms of relationships between model's objects. As was mentioned in section "Mapping Object Schema to Data Schema", relationships between *persistent storage entities* are represented in a runtime data model as associations between `AttributeMap` objects of primary and key attributes. In contrast, class relationships are normally modeled as associations, aggregations or compositions and are implemented either via single references or via collections of associated objects. This is related-by-reference approach to inter-object collaborations. Here we encounter the modeling mismatch caused by the essential difference between relationships' representation in object-oriented and data modeling worlds. As a consequence, the YAPL can naturally map particular domain class to a *persistent entity* and the native properties to their *persistent attributes* but concerning native relationships mapping it is not that straightforward.

We pass RW-accessors to the runtime data model to declare how some *persistent object's* properties must be accessed. But there is no explicit way to specify, for example, that in order to obtain dependents of this *persistent object* the YAPL must traverse an existing association between its base domain class and other related class. There is no such mapping mean in the YAPL. Therefore, the information about class relationships in the domain object schema are completely dismissed on the runtime data model level. That is, for a data model all domain classes appear as unrelated artefacts and relationships between them are, let's say, re-introduced via reference-by-OID association between their *persistent classes*. We use the term *reference-by-OID* to state that related *persistent objects* have the same OID value stored in the corresponding primary and foreign key attributes.

So, the relationships model in the applications that use YAPL is overridden on the *persistent types* layer and is completely data model driven. A data schema defines corresponding runtime data model which in turn enforces *persistent objects* that represent related *persistent storage* entities to be connected by OID values even if there is no object-oriented relationships between base domain classes of these *persistent objects*.

One rationale behind relationships management which is enforced by runtime data model and based on the primary/foreign keys is the uniform reference-by-OID relationships implementation. As was outlined above, class relationships in object schemas can be implemented in number of ways: via references, pointers, wrappers, collections of different types etc. Dependent objects in collections might know their parent or might not. In fact, there is plethora of implementation choices. It is up to application developer which one to use. By using only the reference-by-OID implementation, YAPL ensures that all relationships between *persistent objects* are effectively bi-directional i.e. can be traversed in both directions from parent to child and from child to parent. The most exiting aspect of the uniform implementation is that it facilitates the generic relationships management mechanism that handles inter-object relations in fully automatic manner. This automatic mechanism is accessed via the *relationships management interface* defined in the `PersistentObject` class as part of the persistency interface (see figure 5).

Keeping in mind these advantages it may sound reasonable not to define class relationships on the *domain classes layer* but rather define them only on the *persistent types* layer as relationships between *persistent classes*. Business logic that must operate on related objects migrates then from domain classes to some external persistency dependent service layer implemented on top of the *persistent types* layer. If application developers decide to refactor the object schema in this way they must understand the specifics of YAPL's automatic relationships management. In practice, it is usually possible to change the nuts of some class, its implementation, but altering its interface is undesirable. Under such design constraints, one workaround would be to adapt the existing interface to the `PersistentObject` interface.

Now let's consider the relationships management interface. It should cover the minimal set of operations to traverse a *persistent objects* relationships graph. All interface operations involve two *persistent objects* (or their type identifiers) that might have different types. Therefore, relationships management operations are implemented as member function templates. These member function templates can generate a family of member functions that will cover any possible type-to-type combinations. The first type in this case is represented by the template parameter `DT` of the `PersistentObject` (i.e. *host type*) and the second type (*alien type*) is obtained via the relationships management operations' template parameters.

As soon as a runtime data model is created it will drive the execution of relationships management functions. Well, that means the relationships graph is not checked at the compile-time simply because it is defined by the data model which is a dynamic object structure generated at runtime. Therefore the relationships management in YAPL is dynamically driven by runtime data model. In practice, YAPL clients can compile and write flexible business logic that is not coupled to an existing data schema. But this logic will be controlled by the data model metadata at the runtime. By checking return values of relationships management functions or catching exceptions, YAPL client's code can choose different execution paths.

The method `CheckRelationship()` receives a type identifier of some *persistent class* checks whether a relationship between these type and the host type is defined in the runtime data model. The method `IsChild()` accepts two *persistent objects* of different types and checks whether they are relatives (i.e. parent and child). The `AddChild()` also accepts two *persistent objects* and establishes a parent-child relationship between them which implies setting foreign key attribute of the dependent *persistent object* to the value of the parent object's primary key. It is up to YAPL developers to decide whether to save the child *persistent object* immediately or let the YAPL client to save it manually. The method `GetParent()` retrieves the parent for a given child *persistent object*. To obtain all children *persistent objects* for a given parent object the `GetChildren()` should be used. The method should return an enumeration of *persistent objects*.

Some of the relationships management operations like for example `CheckRelationship()` or `GetChildren()` require type information about the alien type. For this purpose the class `Type2Type` from Loki library can be utilized [AAMCD]. It is a type's representative, a light type identifier that we use to transport the type information about alien types to REMI functions.

Considering the `GetChildren()` operation arises one critical question: where children *persistent objects* are actually taken from? One might expect that they stored in some collection that is probably managed via automatic storage mechanism. The answer is negative: YAPL does not manage collections of children automatically because this would be overkill. All relationships between *persistent objects* are bi-directional and hence making one *persistent object* dependent from another one does not require maintaining children collections. Children can exist independently in the object space and can be reached for example via some kind of persistent objects cache.

Another question that concerns fetching of children *persistent objects* is cascading retrieval. Cascading retrieval of a *persistent object* denotes the process of automatic recursive relationships traversal to make sure the whole relationships graph is fetched into memory. Without cascading retrieval relationships would be traversed on demand by explicitly calling the relationships management methods. YAPL developers are free to decide whether cascading retrieval should be implemented.

The YAPL naturally supports one-to-one and one-to-many relationships. The relationships management interface perfectly fit for navigation from a parent *persistent object* to its children and in the other direction from children to their parent. The implementations of relationships management methods for one-to-one as well as one-to-many relationships would be pretty straightforward. What might complicate them is the natural support of many-to-many relationships. Let's consider many-to-many relationship in gory details.

There are several methods to model many-to-many relationships depending on *persistent mechanisms*. So, in relational databases, they are normally implemented via associative table - a data entity whose sole purpose is to maintain the relationship between two or more tables. So, YAPL developers should decide whether to encapsulate from YAPL clients the way how many-to-many relationships are implemented. If they are realized via associative table, should YAPL clients be aware of this table? If yes, the relationships management interface implementation would remain simple and the whole dependents retrieval burden would be imposed on YAPL clients. Consider *persistent storage* entities from our example data schema: `staff` and `task` (see figure 2). They are related as many-to-many and there is an intermediate table `stafftask` that connects them. Runtime data model exactly represents the data schema and hence to traverse for instance which tasks a particular worker is assigned to the application YAPL clients would need to perform two hops. First, it must call:

```
STAFFTASK_RECORDS staffRecords = STAFF::GetChildren();
```

Second, for each retrieved dependent stored in the `staffRecords` enumeration the corresponding parent from the `task` entity must be retrieved by calling: `STAFFTASK::GetParent()`. From other perspective, it would be rational if YAPL clients can navigate this relationship in both directions by simply calling `STAFF::GetChildren()` or `TASK::GetChildren()`. This way would make the REMI implementation more complicated.

One of the side effects of the dynamically driven, data model-based approach to relationships management is that YAPL clients can implement business logic components that accommodate possible future data schema modifications and does not require to be rebuilt each time the data schema changes. The data model can be dynamically uploaded or reloaded in the runtime without enforcing such components to be recompiled or even restarted. This might be especially useful in the 24/7 availability environments.

## Referential Integrity Enforcement

Referential integrity (RI) is the assurance that all *persistent storage entities* that have foreign keys (i.e. are dependents in parent/child relationships) always refer either to an existing parent or to NULL. Therefore when parent objects are modified or deleted, the *persistent storage* must be automatically updated to remain in consistent state. Not all *persistent storage* types include mechanisms to support referential integrity. We can not rely on the concrete underlying data management implementation and hand over the integrity issue to the *persistent mechanism* site. Therefore, the YAPL implements its own referential integrity mechanism and joins the “object purists” camp which argues that referential integrity rules should be implemented within the persistence framework [AMRI]. The advantage of YAPL-based referential integrity support is the ability to keep both *persistent storage* and *persistent objects* memory space in consistent state when cascades caused by CRUD operations and driven by relationships between *persistent objects* occur.

There are three types of cascades: cascading delete, cascading save and cascading retrieval. Cascading retrievals were outlined in section “Relationships Management”. When some *persistent object* is saved, cascading saves implies that all its children are recursively updated or a new parent is inserted. Therefore, cascading saves can lead either to cascading updates or to cascading inserts. YAPL developers may not implement cascading saves and it is perfectly acceptable with regard to referential integrity assurance under following stipulations:

1. Parents can't change the primary key values and hence there is no need to perform cascading update of all dependent foreign keys. This is the case when YAPL provides *constant primary key* guarantee (more about in section "Automatic Management of Infrastructure Attributes").
2. Data schema allows insertion of new children with NULL foreign keys. Neither parent insertion nor parent existence check is required.

The last type of cascading that requires most attention is the cascading delete. Deleting parent *persistent objects* without taking care about children can easily bring the *persistent storage* into inconsistent state. Therefore the referential integrity must be enforced. Towards this end, YAPL cascades delete by recursively tracking the relationships graph of the *persistent entry* being deleted. It handles each relationship in accordance with the specified referential integrity policy. As was mentioned in section "Mapping Object Schema to Data Schema", each relationship between two *persistent types* defined via the runtime data model interface of the `PersistentObject` should specify the referential integrity policy (stored in `AssociationHolder` objects). Different referential integrity policies are briefly described in the table below:

RI policy	Description
RI_NO_ACTION	Deletion of a parent <i>persistent storage entity</i> is restricted when there is at least one dependent entity that references it
RI_CASCADE	Deletion of a parent <i>persistent storage</i> entity causes recursive deletion of all dependent entities
RI_NO_CHECK	Deletion of a parent <i>persistent storage</i> entity may cause referential integrity violation because no processing of dependent entities is performed. Relationships that do not require referential integrity must be defined with this RI policy

Cascading deletion begins from the CRUD method `Delete()`. First of all, cascading deletion implies starting a transaction to make sure the *persistent storage* will not get corrupted. The architecture of the cascading machinery is based on complicated concepts but the core referential integrity enforcement logic is completely generic. In essence, the integrity enforcement is event driven. That is, the first action performed by the `Delete()` method is the notification of all dependents of the *subject persistent object*. Only when the dependents are processed and all referential integrity policies approve the deletion, the corresponding *persistent storage entry* is physically erased from the *datastore* and the *subject persistent object* transits to the `NOT_BOUND_TO_PSE` state (therefore, this object remains eligible to be saved into the *persistent storage* again).



AssociationHolder objects). The referential integrity policy determines the further processing.

When the RI\_CASCADE policy is specified all children of a parent *persistent object* are retrieved (either from the *persistent storage* or from *persistent objects cache*) and committed to deletion. This presents potential performance hazard. When the data schema is complex enough in terms of chained relationships it can take a while to traverse the whole relationships graph. Another critical cascading issue addresses cycles. A cycle occurs when a cascade cycles back to the starting point. In this case the YAPL can run into an endless loop. YAPL developers should ensure it will not happen.

Here are some notes about the utilized double dispatcher's implementation which is `Loki::BasicDispatcher`. This is a dynamic engine that allows registering handler functions in runtime and ideally fits with the runtime data model concept. It keeps a runtime structure and uses runtime algorithms that help in dynamically dispatching event processing requests depending on event types. When a data model is created and particularly when a relationship between two *persistent types* is defined (via the runtime data model interface of the `PersistentObject` class), the `BasicDispatcher::Add()` should be called to register two `Event` class template instantiations generated for the base domain classes of these two *persistent types*. The `BasicDispatcher::Add()` also requires a pointer to some event handler function that accepts two `EventBase` objects. This event handler function (e.g. `ReferentialIntegrityEnforcer::EnforceRIHelper()` as shown in figure 10) would perform downcast adjustments with received `EventBase` objects and finally would call the `ReferentialIntegrityEnforcer::EnforceRI()`. When the `BasicDispatcher` is fired up to dispatch events it finds the required event handler function in logarithmic time and calls it.

As an example, figure 11 depicts the complete event processing cycle when a *persistent object* of type `DIVISION` is created and then immediately erased. This diagram also shows how during the runtime data model initialization a relationship is established between the `DIVISION` and `BRANCH` *persistent classes*.

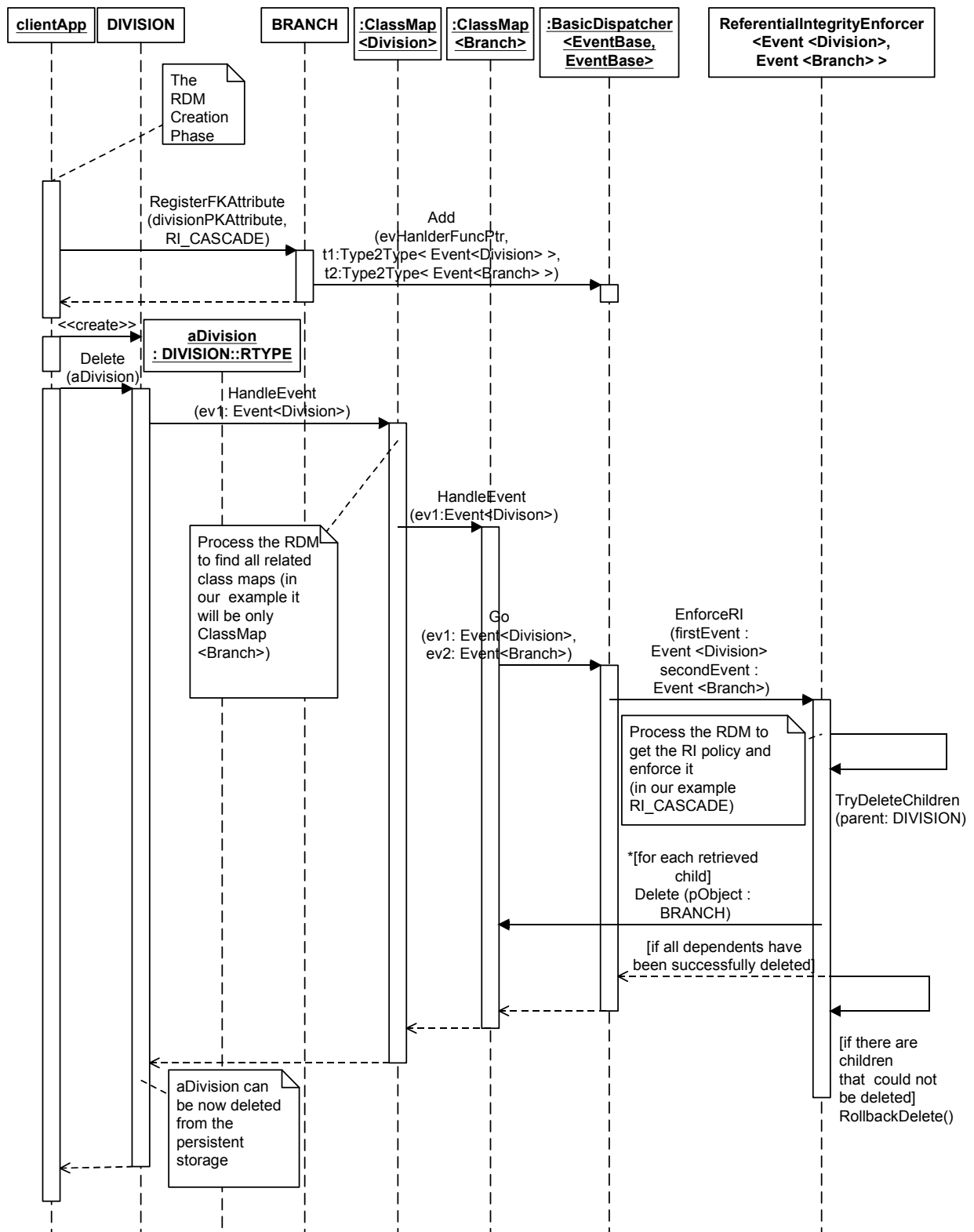


Figure 11. Sequence diagram of the RDM driven cascading deletion.

The last note about referential integrity processing is the order of integrity policies enforcement in the case when one *persistent type* participates in many relationships. The relationships should be processed in strictly defined order: first relationships with the policy RI\_CASCADE, afterwards RI\_NO\_ACTION and RI\_NO\_CHECK respectively.

## Persistent Storage Access Commands and Queries

To perform CRUD operations and to retrieve *persistent objects* selectively in accordance to user data lookup needs the YAPL must generate the *persistent storage* access commands. The semantics of a *persistent storage* access command depends on the underlying data management implementation. A command can be either a sequence of some API calls or it can be a human-readable string, as for example a statement in the Structured Query Language (SQL). SQL statements are used to insert, retrieve, modify, and delete data in a relational database.

It could be tempting to promote SQL as the basic mean to communicate with all types of *persistent storages* and allow applications to construct SQL statements for querying data. If the underlying *persistent storage* is not a relational database it should not be problem to re-process SQL string and submit the query via the *datastore* specific interface. The YAPL does not take this approach because of two major disadvantages: embedding SQL code does not live up with the entire runtime data model concept, kills flexibility and ignores attribute value types. Otherwise, clients would be able to submit SQL involving *persistent storage entities* and attributes that are not represented by a runtime data model. It would be tiresome to track and modify all SQL strings within an application once its data schema changes and lastly it would be possible to format SQL string with setting attribute values without taking into account its type.

The YAPL is about encapsulating the *persistent mechanism* specifics. Therefore, it introduces data manipulation queries (DMQ) - a proverbial level of indirection that encapsulates access to any *persistent storage* behind the uniform data querying interface. Data manipulation query elements are objects structures that consist of concrete `DataManipulationQuery` objects and instances of the `DataManipulationQueryElement` class hierarchy. YAPL clients build data manipulation queries and then pass them to the query processing subsystem that redirects them to concrete *persistences mechanisms* (how YAPL processes manipulation queries is described in section “Persistent Mechanisms”). Let’s first take a look at the figure 12 which depicts these classes.

The `DataManipulationQueryElement` class hierarchy represent well known Composite design pattern [GOF]. The advantage of this pattern is the ability to organize data manipulation queries into nested and hierarchical structures and to create fairly complex data selection conditions that satisfy virtually any *persistent storage* querying needs. On the other hand, composite lets YAPL to treat leaf data manipulation query elements as well as their compositions in uniform fashion. Instances of `DataManipulationQuery`’s subclasses specify which type of *persistent operation* (insert, update, delete, retrieve etc.) must be performed on the *persistent storage*. Objects of the `DataManipulationQueryElement` specializations represent elementary building blocks of data manipulation queries that actually specify how the *persistent storage* should be manipulated.

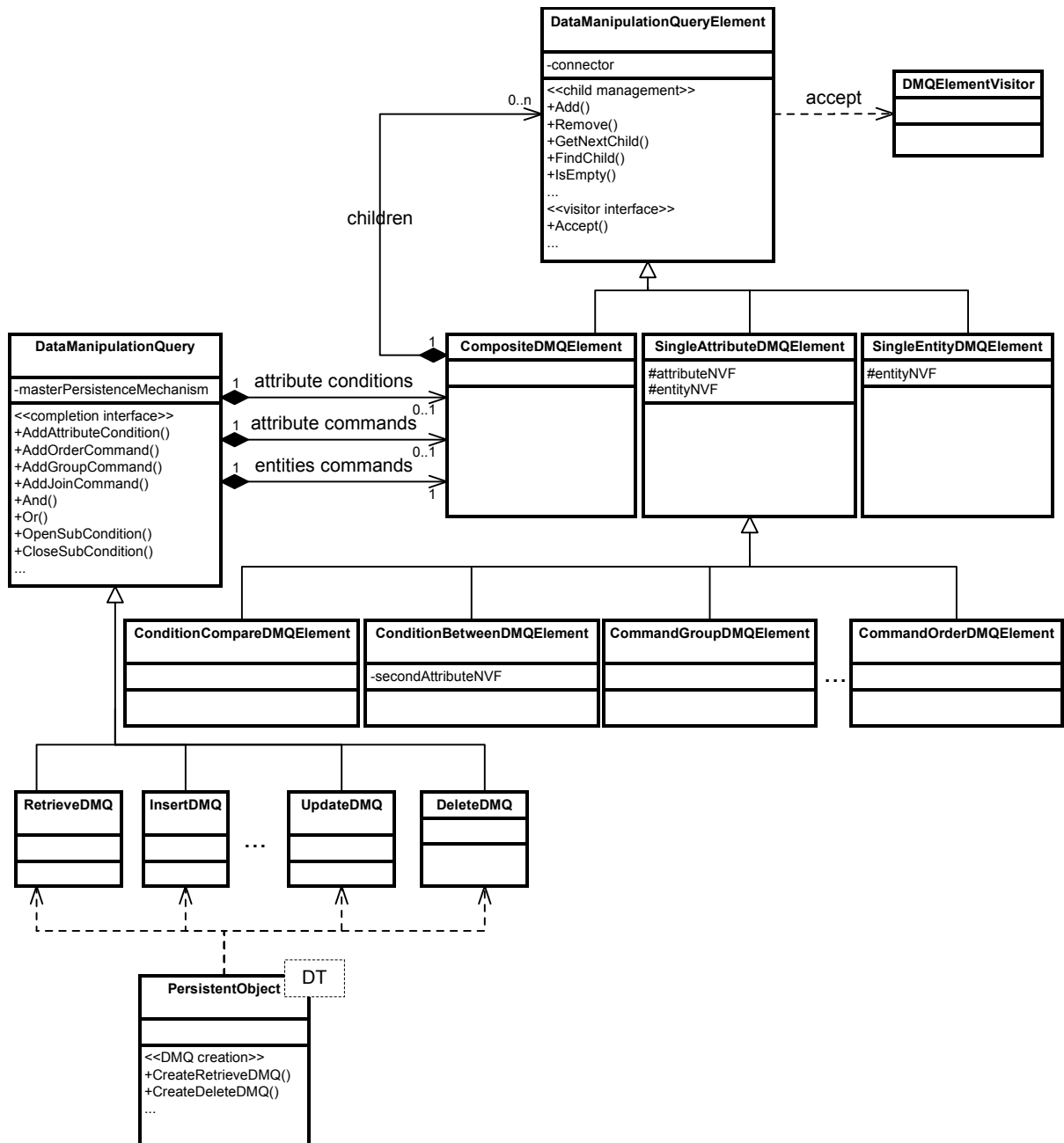


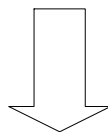
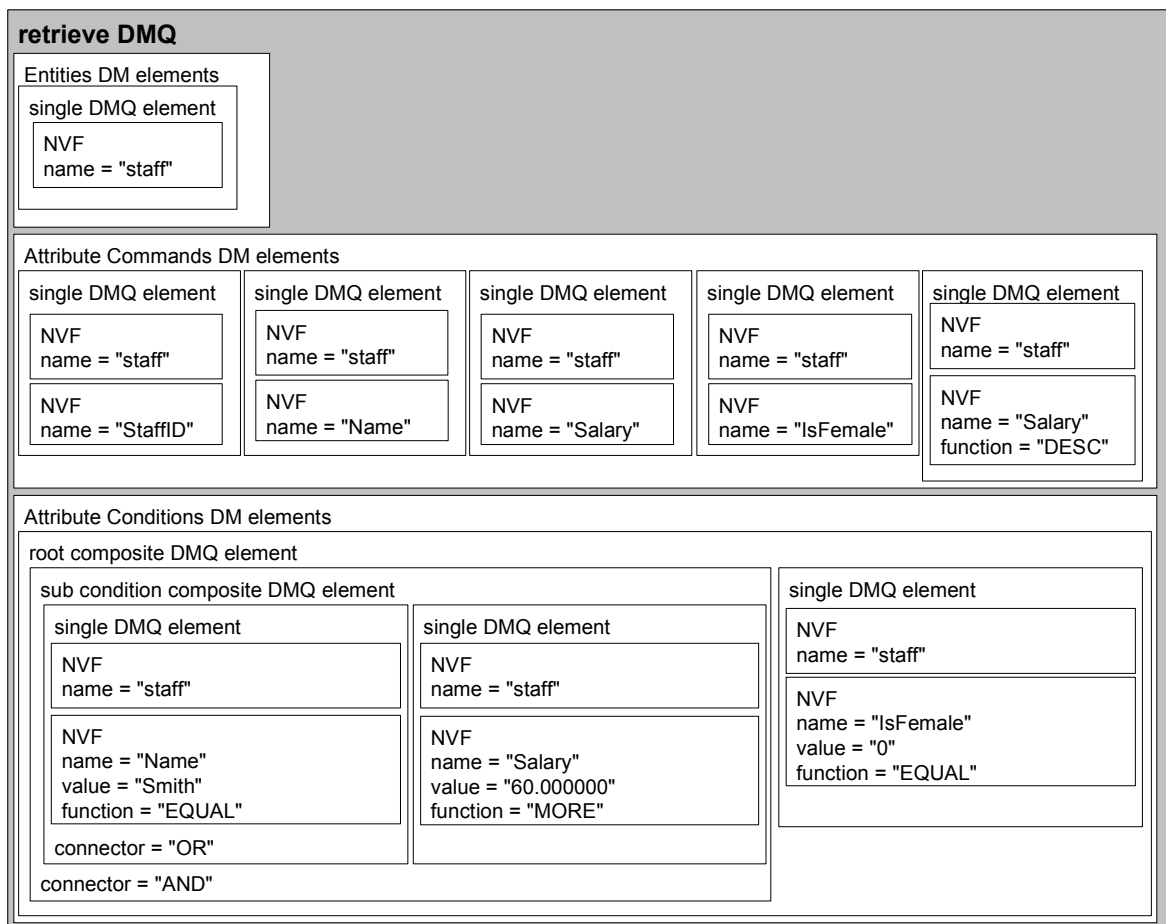
Figure 12. Class diagram of the DMQ infrastructure.

In general, `DataManipulationQuery` is a container that is filled with data manipulation query elements. These elements will be split into three categories: one for data manipulation query elements that define subject *persistent storage entities*, the second for specifying their attributes along with commands that must be executed on them (e.g. aggregate functions AVG or SUM) and the third one will specify the conditional part of a data manipulation query. With regard to the composite nature of `DataManipulationQueryElement` objects organization, the `DataManipulationQuery` will actually store three objects of type `CompositeDMQElement` representing the root objects of each data manipulation query elements category.

The data manipulation query elements store the information about the subject entity and its attribute(s) by holding NVF (“name value function”) objects which are an universal atomic

descriptors that specifies either an entity or an attribute and encapsulates a name, a textual representation of some value and a *persistent function* that must be applied to the entity or the attribute it describes. For example, when an NVF describes a value-to-attribute assignment (required for save or update *persistent operations*) it will contain an attribute name along with its new value and the *persistent function* will be "ASSIGN". Other examples of *persistent functions* are aggregation functions (e.g "AVG", "SUM" etc.), joining, grouping, ordering directives and comparison predicates (e.g. "LESS", "EQUAL" or "BETWEEN"). Various types of data manipulation query elements require different number of NVFs. So the SingleEntityDMQElement holds only one NVF that describes a single *persistent storage entity* when ConditionBetweenDMQElement requires three NVFs: one for the host entity and two others for both left and right attribute values that define the comparison interval.

To illustrate the above data manipulation query description, a structure of a simple DMQ and its SQL equivalent is shown in figure 13.



```
SELECT staff.StaffID , staff.Name , staff.Salary , staff.IsFemale FROM staff
WHERE ( ( ( staff.Name = "Smith" ) OR ( staff.Salary > 60.000000 ) ) AND ( staff.IsFemale = 0
) )
ORDER BY staff.Salary DESC
```

Figure 13. Schematic representation of a DMQ structure.

To manipulate on *persistent storage*, YAPL clients should first obtain a data manipulation query object. Constructing data manipulation queries for *persistent operations* normally implies two steps: creation of a basic data manipulation query and its completion with manipulation query elements according to application specific data selection conditions and processing directives. The interface for creation of data manipulation queries is integrated in the persistency interface (see figure 12). It is used to build data manipulation queries that should manipulate on the *persistent entity* (DMQ-target entity) represented by the template parameter DT of the `PersistentObject` class template. All methods of the data manipulation query creation interface should return queries that can be immediately executed without any completion with query elements.

Note however, that only such *persistent operations* as retrieve and delete be committed by YAPL clients directly. That is, construction of `InsertDMQ` as well as `UpdateDMQ` objects can not be controlled by YAPL clients. The CRUD method `Save()` method is responsible for building the appropriate data manipulation query depending on the *subject persistent* object state. Direct construction of `DeleteDMQ` may be required for performing some batch data processing but YAPL clients should keep in mind that committing deletion data manipulation queries bypasses the referential integrity enforcement. Normally clients should first retrieve *persistent storage entries* and then delete them by calling the CRUD operation `Delete()` on their *persistent objects*.

The second step of building a data manipulation query is actually the specifying of selection conditions as well as ordering, grouping, joining etc directives by adding concrete data manipulation query elements. This is performed via the completion interface of the class `DataManipulationQuery`. Example functions of this interface are shown in figure 1. The data manipulation query depicted schematically in figure 13 can be built and completed by this C++ code:

```
//initialize the selection pattern
STAFF::RTYPE pattern = STAFF::Create();
pattern->SetName("Smith");
pattern->SetSalary(60);
pattern->SetIsFemale(0);

//create a basic DMQ
RetrieveDMQ* dmq = STAFF::CreateRetrieveDMQ();

//complete the DMQ

dmq->OpenSubCondition();
dmq->AddAttributeCondition (pattern, attributeHandleStaffName, EQUAL);
dmq->Or();
dmq->AddAttributeCondition (pattern, attributeHandleStaffSalary, MORE);
dmq->CloseSubCondition ();

dmq->And();
dmq->AddAttributeCondition (pattern, attributeHandleStaffIsFemale, EQUAL);
dmq->AddOrderCommand(staffTypeIdentifier,
attributeHandleStaffSalary,DESC);
```

Let's now scrutinize this code. The `AddAttributeCondition()` specifies selection conditions. For example the first call to this method conveys the selection condition "Name = Smith". Programmatically it is expressed in terms of arguments passed to the method

`AddAttributeCondition()`: the argument `attributeHandleStaffName` (an *attribute handle*) specifies that the value of the column “Name” (see the table “Staff” in figure 2) should be EQUAL to the value of the native property “name” of the object `pattern`. Since the `AddAttributeCondition()` accepts a *persistent object*, this method as well as other completions methods should be implemented as member function templates that obtain the *persistent types* from its own set of parameters. The method `AddOrderCommand()` instructs the data manipulation query to order the retrieved data by the column “Salary”. Since ordering commands do not require attribute values, there is no need to pass concrete *persistent objects* and only the type identifier of the *persistent class* `STAFF` is specified. The calls to methods `OpenSubCondition()` and `CloseSubCondition()` create a sub-condition according to the nested conditions expression. The methods `And()` and `Or()` are used to specify the corresponding logical connectors between the selection conditions.

Most methods of the `DataManipulationQuery`'s completions interface need attribute handles of *persistent storage* attributes. Therefore, YAPL clients need access to attribute handles used by the runtime data model interface (i.e. method `RegisterAttribute()` in `PersistentObject`) to be easily accessible. It is up to YAPL clients and developers to decide how to manage the attribute handles and provide them to all system components. One particular solution will be represented in section “Deployment Architecture of YAPL-based Systems”.

One important issue to emphasize here is the *persistent object* based conditions value initialization. Building a selection condition that compares a single attribute to a given value requires knowledge about this attribute's type. Clients can acquire this knowledge by taking a look on a particular *persistent class* and its properties' types. Since each property of a *persistent class* that is mapped to some *persistent attribute* is accessible via RW-accessors, it is reasonable to use RW-accessors for initialization of data manipulation queries as well. Therefore, a temporary *persistent object* is created (the object `pattern` from the above code listing) to serve solely as a selection pattern (*pattern persistent object*). This object is then passed to `AddAttributeCondition()` member function template along with the attribute handle that instructs the YAPL which attribute defines the condition. The specified attribute handle is then used to obtain the attribute value from the *pattern persistent object*. This yields a fully initialized NVF that is the core of a newly created `SingleAttributeDMQElement` object that is added to the manipulation query being completed.

Most significant advantage of the *persistent object* based conditions value initialization is the centralized attribute values handling performed only via the runtime data model. So, when for example the name of some *persistent attribute* is modified, YAPL clients should not change the code that builds data manipulation queries involving this *persistent attribute* – they only need to reconfigure the mapping of the *persistent attribute* to the corresponding RW-accessors.

Since building of data manipulation queries is a costly procedure, the YAPL should allow “reusing” of already completed and executed data manipulation queries. Changing the NVFs of their elements should be allowed as well. Considering the example above, it should be possible to re-execute the `dmq` object multiple times with different selection conditions (e.g. values of the columns “Name” or “Salary”).

The class `DataManipulationQuery` should provide an intuitive completion interface that will allow YAPL clients to express any data manipulations needs. Though YAPL specifies the general idea of the data manipulation query completion concept, elaborating the uniform data querying interface that will cover all *persistence mechanisms* supported by the YAPL is a hard issue. Depending on the *persistence mechanism* which will execute a data manipulation query, YAPL clients should be able to specify different selection condition expressions like attribute-compared-with-value, attribute-compared-with-attribute, attribute-within-values-interval, attribute-compared-with-sub-query-result as well as different data manipulation directive (i.e. grouping, ordering and joining). It is up to YAPL developers which operations to include in the completion interface and how to elaborate them. The evolution of the completion interface is directly reflected on the `DataManipulationQueryElement` class hierarchy. That is, as different selection conditions and data manipulation directives need to be implemented, YAPL might need to integrate specific subtypes into this hierarchy.

Data manipulation queries need to be handled in different contexts and for different purposes as for example execution by different *persistence mechanisms*. A query is completely defined by elements it contains and therefore accessing and iteration over them is critical for the investigation of the query's structure. Concrete data manipulation query element types define many distinct and unrelated to each other operations (e.g. NVF getters). To work with data manipulation query elements in generic fashion via the `DataManipulationQueryElement` interface and, on the other hand, to preclude "polluting" this interface with all concrete operations and to keep it closed for modifications as well as opened for extensions Robert Martin's Acyclic Visitor design pattern [RMAR]. The pattern eliminates cyclic dependencies. In accordance to the Acyclic Visitor pattern the class `DataManipulationQueryElement` is a visitee and defines the `Accept` operation that allows the instances of concrete `DMQElementVisitor`'s subtypes to enter the data manipulation query element. Each element class will have an associated abstract visitor type. Acyclic Visitor works perfectly with the composite object structures. A concrete visitor that is sent to a root data manipulation query element of type `CompositeDMQElement` is automatically sent to all its children. This enables transparent iteration over collections of query elements.

Introducing the visitor infrastructure to the data manipulation query elements hierarchy prescribes a strict and well defined approach to query processing. A processing algorithm should be implemented as a concrete data manipulation query element visitor. Sending this visitor to query elements triggers the execution of the algorithm.

## Retrieving Data from Persistent Storage

So far we have studied how data manipulation queries are constructed. Most frequently used queries are data selection requests that must extract some *persistent storage entries* and bring them into object memory. Such queries are represented by objects of the class `RetrieveDMQ`. The class `RetrieveDMQ` not only defines the data selection criterion but also contains the results of its execution in form of *raw persisted data*. This *raw persisted data* can be unmarshaled into *persistent objects* and they in turn can be obtained in sequential or random access manner via the `RetrieveDMQ`'s interface. Therefore, a `RetrieveDMQ` object also plays the role of a *data cursor* that can traverse forward and backward in the result dataset as well as remember its current *persistent entry*. This class provides the data cursor interface (see figure 14) that performs not only unmarshalling of *persistent objects* but also

returns a single scalar value (i.e. result of some aggregation data manipulation query). If the number of entries is large, it is expensive to unmarshal all *persistent objects* at once. Hence, for performance and memory consumption reasons, the RetrieveDMQ allows not only unmarshalling of all *persistent objects* at once, but also allows navigating over the dataset by getting only one *persistent object* at a time.

The operations of data cursor interface will need to unmarshal data into *persistent objects* of different types, and therefore they should be implemented as member function templates that obtain the *persistent types* from its own set of parameters. On the other hand, they should perform some runtime *persistent type* check to disallow unmarshalling of *raw persisted data* into *persistent objects* of the wrong type.

Let's now unveil the RetrieveDMQ's internal structure to understand how *persisted raw data* is stored and how it is transported into *persistent objects*. Take a look at the figure 14.

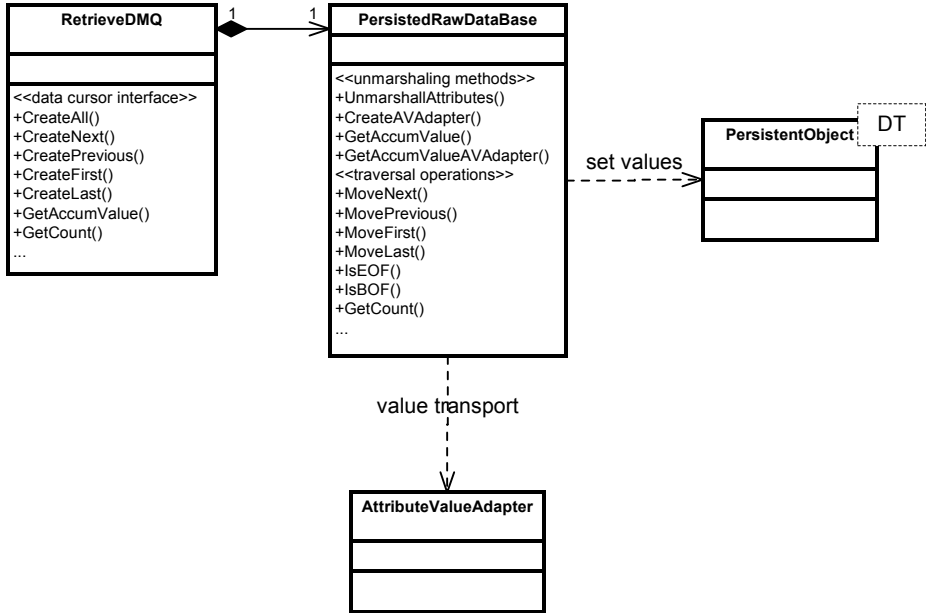


Figure 14. Class diagram of the data retrieval mechanism.

The class `PersistedRawDataBase` is an abstraction of a records-based dataset that can be traversed in both directions and remembers its current record. Its subclasses are concrete *persisted raw data* representations implemented for each concrete *persistent mechanism* type. But `PersistedRawDataBase` is not only passive dataset holder. Its method `UnmarshallAttributes()` actually performs unmarshalling and does all the heavy lifting required for setting a record's fields to a *persistent object's* properties. The values are set to *persistent objects* via the `AttributeMap::SetValueToPersistentObject()` function of the persistent object values access interface. An attribute value adapter object which this method requires as a parameter, is created by concrete `PersistedRawDataBase's` subclasses. Their implementations of the virtual function `CreateAVAdapter()` create attribute value adapters and initialize them with concrete attribute value for a specified field. In this regard, the `PersistedRawDataBase's` class hierarchy is designed according to the Template Method design pattern [GOF] where the `CreateAVAdapter` is the abstract *primitive operation* of the template method `UnmarshallAttributes()`.

The class `PersistedRawDataBase` can also return one single scalar value via the function `GetAccumValue`. It is up to specializations now to store the single value (for example some `PersistedRawDataBase` subtypes may interpret a single value as the first field of the first record). The method `GetAccumValue` is also a *template method* which lets subclasses redefine the abstract function `GetAccumValueAVAdapter`.

In the `RetrieveDMQ`'s unmarshalling mechanism the class `AttributeValueAdapter` carries out its function of the transportation mean. It brings the attribute values from *persisted raw data* to *persistent object*'s properties via the corresponding *attribute maps*.

## Persistent Mechanisms

A *persistence mechanism* (PM) is any technology that can be used to permanently store data for later update, retrieval, and/or deletion [AMPL]. Possible *persistence mechanisms* include flat files, databases (relational, object-relational, hierarchical, network), object-bases etc. Different *persistence mechanism* types are accessible via different APIs. The design goal of the YAPL is to support all major persistence mechanisms. To enable the smooth framework evolution and at the same time preserve the backward compatibility with existing applications the YAPL introduces the concept of a *generalized persistence mechanism*. The YAPL provides the uniform object persistence and data selection interfaces that are independent from concrete persistence mechanism types. Details and APIs of particular *persistence mechanisms* are by no means exposed to YAPL clients. They use only persistency interface and data manipulation query objects. This is one aspect of the *generalized persistence mechanism* concept. It addresses the software maintainability issue and assures that the application code remains unchanged when the underlying *persistence mechanism* is switched. Another aspects concern the ability to switch the *persistence mechanism* at runtime and the ability to submit the same data manipulation query object to different *persistence mechanisms*.

Of course, the benefits of *generalized persistence mechanism* concept do not come for free due to the additional level of indirection in the data manipulation query representation. This level of indirection sacrifices a bit of overall YAPL performance but strongly favors the YAPL generality.

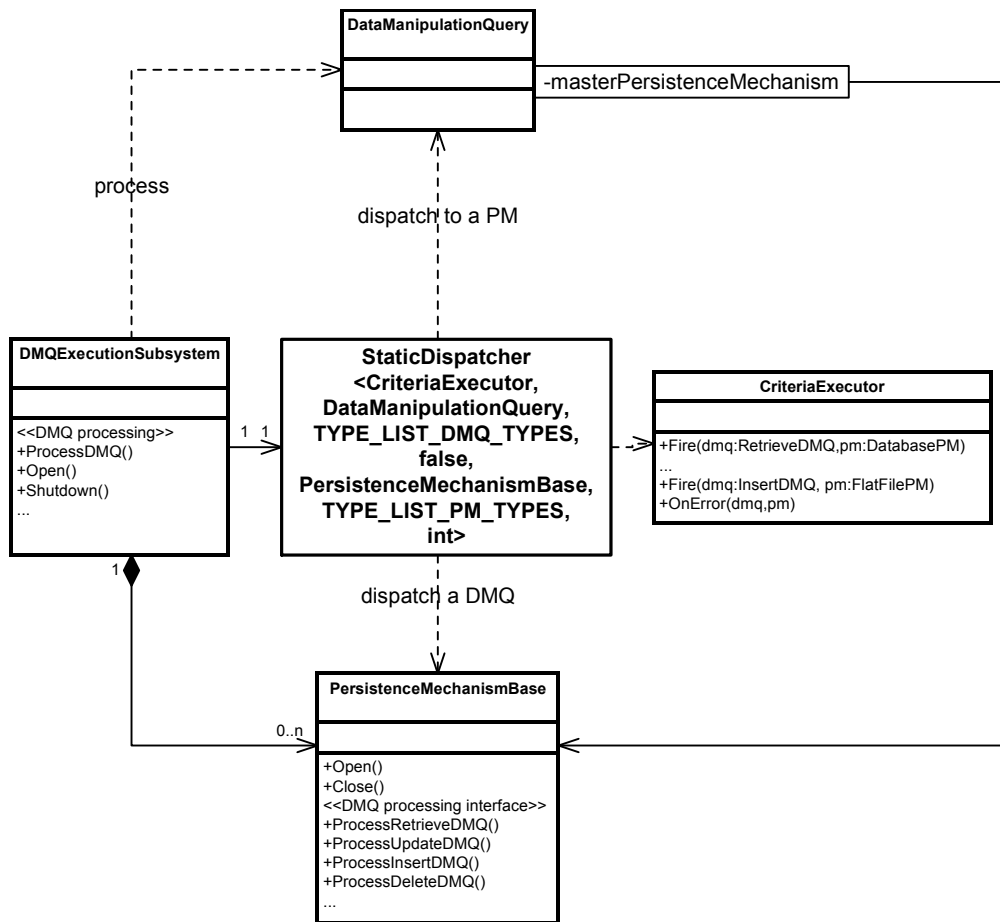


Figure 15. Class diagram of the generalized persistence mechanism infrastructure.

The YAPL achieves the independence from concrete *persistence mechanisms* by effectively decoupling the data manipulation query representation from its execution and result dataset representation. Data manipulation queries are submitted for execution via the interface of the class `PersistenceMechanismBase` which is the abstraction of the generalized *persistence mechanism*. The `PersistenceMechanismBase` declares one abstract operation for each data manipulation query type. All those operations are denoted collectively as the data manipulation query processing interface (see diagram in figure 15). There is one subclass of the `PersistenceMechanismBase` for each supported *persistence mechanism*.

Instances of concrete `PersistenceMechanismBase`'s specializations are not directly available for clients but are hidden behind the higher-level façade class `DMQExecutionSubsystem` which should be available as a singleton. As the name implies, this interface is the façade to all the query execution subsystem realized by *persistence mechanisms*. The goal of this class is to provide one uniform operation for submitting of all types of data manipulation queries to all available *persistence mechanisms* as well as to provide centralized operations for *persistence mechanism* management.

A pair of operations `Open()` and `Shutdown()` serve for persistence mechanisms initialization and shutdown respectively. Before some *persistence mechanism* can be used it must be first opened. The *persistence mechanism* opening operation implies creation of a concrete *persistence mechanism* object, initializing it, adding it to some internal collection of available *persistence mechanisms* and finally returning its *persistence mechanism handle*. A

*persistence mechanism* handle is required for *persistence mechanism* referencing because YAPL clients should not obtain pointers to created *persistence mechanism* objects.

Once a *persistence mechanism* is opened, its handle is used to bind the runtime data model and data manipulation queries to *persistence mechanisms*. For this purpose, `ClassMap` and `DataManipulationQuery` objects are associated with *persistence mechanism* handles of their master *persistence mechanisms* (see figure 12 and figure 7). By associating these objects with another handles, the YAPL enables flexible runtime data model reconfiguration in terms of the underlying *persistence mechanism* and allows dynamical binding of *persistent types* to different *persistence mechanisms*.

Let's now consider how the class `DMQExecutionSubsystem` enables transparent and uniform processing of data manipulation queries by any kind of a *persistence mechanism*. As we know, `DMQExecutionSubsystem` provides only one centralized entry point `ProcessDMQ()`. The double dispatch of data manipulation query objects and *persistence mechanism* instances based on their dynamic types is the core mechanism behind the `DMQExecutionSubsystem::ProcessDMQ()`. Double dispatch (particularization of multiple dispatch) is the mechanism that dispatches a function call to different concrete functions depending on the dynamic types of two objects involved in the call. The Loki's class template `StaticDispatcher` can be utilized as the double dispatch engine that performs the type deduction algorithm and then fires a function in another class [AAMCD]. This class (`CriteriaExecutor`) defines several overloads of the `Fire()` method for all possible combinations of `DataManipulationQuery` subtypes and `PersistenceMechanismBase` specializations. Each implementation of the `CriteriaExecutor::Fire()` method is the place where a data manipulation query finally meets concrete *persistence mechanism*. The specified data manipulation query object is simply forwarded to the specified *persistence mechanism* instance via the appropriate function of the manipulation queries processing interface. Using the `StaticDispatcher` makes the `DMQExecutionSubsystem::ProcessDMQ` completely generic and closed for modifications according to open-closed principle [OOCAPP]. When new *persistence mechanism* types are added this function never have to be changed or even recompiled. It uses *persistence mechanism* handles specified by the submitted manipulation query object to find the corresponding *persistence mechanism* objects in the collection of available mechanisms and then leverages the double dispatcher by calling the `StaticDispatcher::Go()` method.

In contrast to the `DMQExecutionSubsystem::Open()` operation which opens only one *persistence mechanism* per call, the `DMQExecutionSubsystem::Shutdown()` operation shutdowns all available and opened *persistence mechanisms*.

## Relational Databases

Relational database management systems organize data into related rows and columns and use the SQL as the communication language. RDBMS is without doubt the most widely used *persistence mechanism*. Therefore, the YAPL most likely will need to support it. First of all YAPL developers should decide which legacy library for programmatic access to RDBMS (e.g. Microsoft ADO or ODBC) to use. Then they should adapt this library into the YAPL according to the *generalized PM* concept. Let's assume YAPL developers have some legacy library e.g. `LegacyDBAPI` for low-level access to relational databases. We present a pattern design which shows how this library can be integrated into the YAPL. We also show how

abstract data manipulation query objects are converted to SQL statements. This pattern design is easy to follow so that YAPL developers can similarly implement support of other kinds of *persistence mechanisms*.

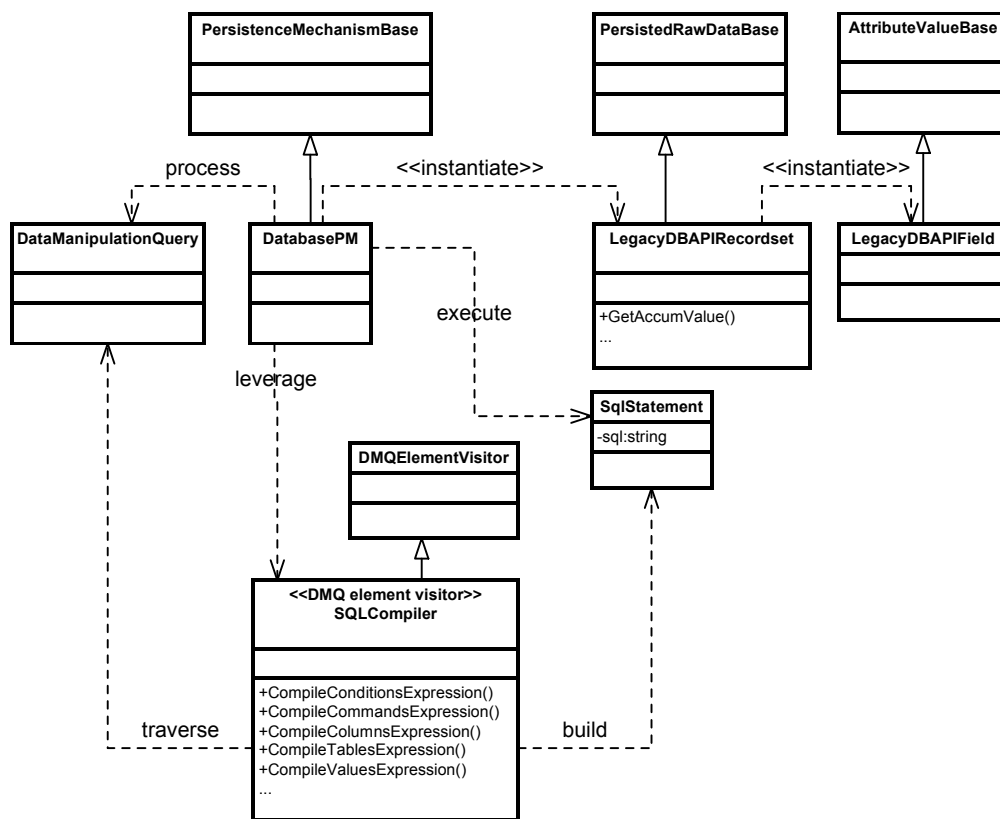


Figure 16. Class diagram of RDBMS access infrastructure

The support of relational databases is represented by a set of interrelated classes (see diagram in figure 16). The master class of this classes bundle is of course the PersistenceMechanismBase's specialization DatabasePM. This type hides the SomeLegacyDBAPI based implementation of relational databases access behind the generalized *persistence mechanism* interface. The DatabasePM's implementation of the base Open() should establish connection with database server and perform all necessary database initialization.

The primary responsibility of the DatabasePM is the generation of SQL statements and its submitting to the underlying database server. The DatabasePM's implementation of the data manipulation query processing interface is just the glue code that takes a query object, creates a SQL statement and forwards it to the low-level SomeLegacyDBAPI. The DatabasePM should generate SQL from a DMQ by sending special data manipulation query element visitor to this query object (e.g. class SQLCompiler in figure 16). This visitor will traverse a query object and will build corresponding SQL statement.

Objects RetrieveDMQ class converted to SQL "SELECT" queries might yield *persisted raw data*. Normally, database access libraries represent *persisted raw data* returned from database servers in form of some recordset. To encapsulate the details of the LegacyDBAPI's recordset implementation a corresponding PersistedRawDataBase's specialization LegacyDBAPIRecordset is introduced. Its implementation of CreateAVAdapter() creates attribute values of type

LegacyDBAPIField. The class LegacyDBAPIField represents a single value of some database attribute. The cooperation of these classes hides the specifics of relational databases according to the generalized *persistence mechanism* concept.

## Deployment Architecture of YAPL-based Systems

Let's consider the structure of a YAPL-based system in terms of deployment components that form executable software run by concrete operating systems. The core YAPL functionality including building and processing of data manipulation query objects as well as management of all supported *persistence mechanisms* should be deployed as a separate component. This component should expose some interface to control *persistence mechanisms* and to create and execute manipulation queries. For example, the interface of DMQExecutionSubsystem class can be exposed. This component is actually referred to as *persistence layer*.

Before objects can be persisted *persistent types* should be defined for their classes, the corresponding runtime data model must be generated and the *persistence layer* must be initialized. The model construction and *persistence mechanisms* initialization might be managed by a separate deployment component that we denote as the *persistence runtime*. The *persistence runtime* represents the *persistent types layer* (figure 4). It utilizes the *persistence layer* and depends on the *domain classes layer* since *persistent classes* are generated from domain classes. The *persistence runtime* can be used by system components that implement application's business logic based on the *persistent types*. Following the proposed approach to elaborate a standalone *persistence runtime* component, most YAPL applications will end up with the deployment architecture depicted in figure 17.

From the overall applications perspective, both *persistence runtime* and the *persistence layer* will provide the ultimate persistency interface that smoothly integrates into existing system architecture as an intermediary layer that resides between the domain classes and the higher level services. This is the most reasonable way to achieve the transparent object persistence.

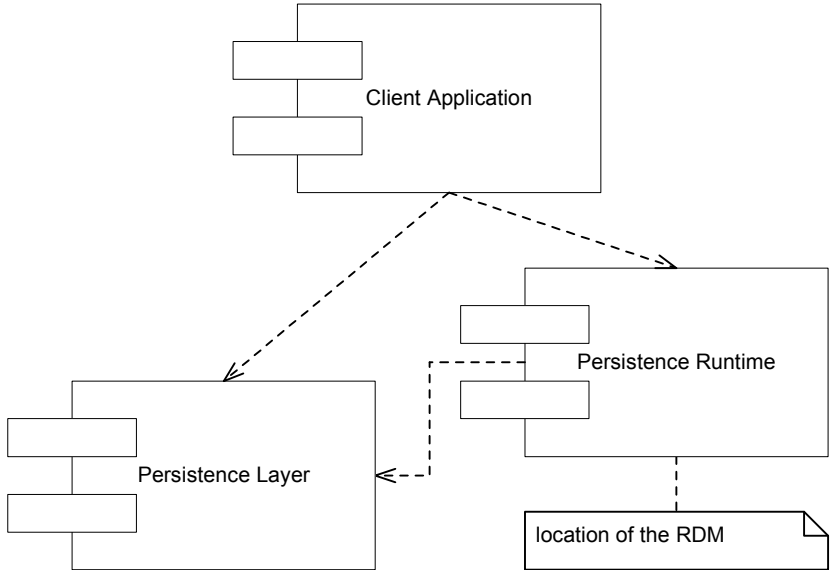


Figure 17. Component diagram of a typical YAPL application for Microsoft Windows.

It is up to YAPL clients to decide whether a persistence runtime repository is required. Simple applications can combine runtime data model definition and business logic in the same component and may not have standalone *persistence runtime*.

We have outlined that one of the core responsibilities of a *persistence runtime* is the initialization of runtime data model. In the most cases the whole model will be created at an application start-up and will be remain unchanged. More complicated scenarios would perform partial dynamic model loading or re-initialisation reflecting the data schema changes at runtime.

In general, a runtime data model is an object structure that reflects a data schema. For complicated data schemas involving hundreds of entities it might be tiresome and error prone to write the model initialization code manually. In this case YAPL clients may consider some automatic code generation facility.

To summarize, the YAPL does not dictate the overall system architecture to the YAPL-based applications. But to craft powerful and flexible systems some common advices might be obeyed. The bottom line is that for complex multi-component applications a common *persistence runtime* should be elaborated.

## Persistent Objects Cache

One of the most essential YAPL features that serves as basis for another critical features is the *persistent objects cache*. We have already mentioned it in many sections of this document. The *persistent objects cache* can dramatically improve the lifecycle management of *persistent objects* as well as *relationships management* in terms of objects sharing. The *persistent objects cache* should be implemented to enable automatic data consistency enforcement. YAPL clients should not carefully track which *persistent objects* are in memory making sure there are multiple *persistent objects* that refer to the same *persistent storage entry* but have different states etc. The *persistent objects cache* would also allow easy *persistent objects lookup* and accessibility which is required for *persistent object* notifications during referential integrity enforcement. Such a notifications mechanism would consult the *persistent objects cache* to obtain and notify all in-memory *persistent objects* as soon as their corresponding *persisted storage entries* are physically deleted. Those *persistent objects* must be marked as “dead” (any further CRUD manipulations on them must be disallowed).

## Implications of Object Schema to Data Schema Mapping

According to the transparent object persistency concept, the domain object schema should not be affected by the persistency aspect. But making a domain class persistent still does not come for free. There are some implications on the design of domain classes. To implement mappings you will probably need to add code to the business classes, code that might impact your application. To access a native property via the RW-accessors a domain class must provide strictly defined public getter/setter. Application developers should keep on mind that RW-accessors are pointers to member functions. The copy initialization mechanism of the `PersistentObject` expects the operator= and copy constructor to be at least accessible. In ideal case, the domain class is suited to be persistent but sometimes the modifications are absolutely unavoidable. Consequently, encapsulation can be broken, domain specific design rules might be violated and so on.

Another critical impact on domain object schema derives from the YAPL relationships management. YAPL takes the data schema driven approach to relationships modeling and therefore uses the primary/foreign key values to manage relationships between *persistent objects*. If a domain classes does not use the automatic storage feature for primary/foreign key values management it must add it own properties for storing the key values and must define getter and setter to be use as the RW-accessors.

Let's consider objects interoperation interface defined in the related domain classes. Such interface might have object collaborations operations like for example `Division::AddBranch`, `Branch::GetDivisions`. On the other hand, the class `PersistentObject` provides extensive relationships management interface (described in section "Relationships Management"). That is, if YAPL clients are willing to continue using the native objects interoperation interface they have no choice but to adapt them to the `PersistentObject`'s relationship management interface. One way to do this is to implement an adapter class according to the adapter design pattern. The Adapter design pattern intents to convert the interface of a class into another interface that the clients expect [GOF]. Hence, the adapter class will inherit from the *persistent class* being adapted. The adapter will override the native interface from the target domain class and will commit to the required method in the `PersistentObject`. Another option is to change the related-by-reference association to related-by-id one. Doing this, the domain class will implement properties for storing the primary/foreign key values and will define getter and setter. This implies additional persistency-related application extension efforts and changes to the objects interoperation interface in order to make its methods subjects to overriding (e.g. by declaring them virtual).

One more issue is building of data manipulation queries. Queries use *persistent objects* to initialize query parameters (*pattern persistent object*). When creation of some *persistent object* is a costly operation the type safe query parameters setting based on *pattern persistent object* might not be a choice. As a workaround, a string-based parameters setting can be allowed.

## Conclusions

The YAPL core architecture is flexible and open for further modifications, new features can be easily implemented and smoothly integrated. In this document we have described the core aspects of a persistence framework. However, as you may have noticed, such crucial topic as transactions support is omitted. In general YAPL developers can easily implement transactions support by chaining data manipulation query objects and adding commit and rollback mechanisms.

## References

[AMDMP] Scott W. Ambler, UML data modeling notation, available at <http://www.agiledata.org/essays/umlDataModelingProfile.html>

[AMPL] Scott W. Ambler, The Design of a Robust Persistence Layer For Relational Databases, available at <http://www.ambysoft.com/persistenceLayer.pdf>

[LCSA] Lakos, Large Scale C++ software design, Reading, MA: Addison-Wesley, 1994.

[COMPJDO] David Jordan, A Comparison Between Java Data Objects (JDO), Serialization and JDBC for Java Persistence, available at <http://www.jdocentral.com>

[CTMPL] David Vandevorde, Nicolai M. Josuttis, C++ Templates: The Complete Guide. Reading, MA: Addison Wesley, 2002.

[OOCAPP] Robert C. Martin, Designing Object-Oriented C++ Applications Using The Booch Method, Englewood Cliffs, NJ: Prentice-Hall, 1995.

[EXCC47] Herb Sutter, Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Reading, MA: Addison-Wesley, 2000.

[AAMCD] Andrei Alexandrescu , Modern C++ Design: Generic Programming and Design Patterns Applied, Reading, MA: Addison Wesley, 2001.

[GOF] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison Wesley, 1995.

[STR] Bjarne Stroustrup, The C++ Programming Language, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[AMRI] Scott W. Ambler, Implementing Referential Integrity and Shared Business Logic, available at <http://www.agiledata.org/essays/referentialIntegrity.html>

[RMAR] Robert C. Martin, Acyclic Visitor, available at <http://objectmentor.com/publications/acv.pdf>