

3 ObjectBase

This chapter describes what an ObjectBase is and how it fits into the MONDO architecture. We will discuss the core responsibilities of an ObjectBase, how to think about modeling and implementing it, what responsibilities it has to MONDO, and what services MONDO can provide to it and the applications that work with it. Later in this document are more chapters that describe the higher level services that MONDO provides to the ObjectBase and applications.

3.1 Introduction

An ObjectBase is a collection of objects that embody knowledge about a particular domain (subset) of the world. What needs to be represented in the computer determines the domain, and the level of sophistication required will determine the details of the DomainModel. Some possible domains are:

- Documents: The DomainModel could be a document type or multiple document types. The ObjectBase could contain one or more documents or a document fragment.
- Chemistry: The DomainModel could be chemical models and mathematical models. The ObjectBase could contain information about a particular compound or the whole CRC (if space permits).
- Properties: The DomainModel would include all the Types of properties your program needs. The ObjectBase could contain a particular configuration set of these properties (e.g. a localization set).
- Hypertext: The DomainModel could be HTML's model, XML+XLL, or a more sophisticated model. The ObjectBase could contain a web site, multiple web sites, or just a single page.
- Beans: The DomainModel could be arbitrary types of objects with a particular structure. An example would be the JavaBean model. The ObjectBase could then contain a collection of Beans that work together.
- Mixed Models: The DomainModel could be the integration of several different models. This might be for a hypertext document discussing chemistry, which has executable Beans within it to support 3D viewing of the chemical models.

The importance of MONDO focusing on the Objectbase and using a very general model is that all of these different types of knowledge are possible and easily expressible within the same architecture and application.

Purpose

An ObjectBase captures the knowledge, operations, and rules required to usefully represent a particular part of the world in a computer system. All programs have this model of the world within them, but they also have very application specific functionality. The value of explicitly identifying the ObjectBase is to clearly separate the more general knowledge the application is using from the application-specific processing it does. Information in the ObjectBase should be focused on

An ObjectBase uses objects (DomainObject) to collectively represent knowledge about the domain. DomainObjects are not separated because they are very different from other objects internally (although that may be the case) but that as part of the ObjectBase they should be focused first on general knowledge. We want the DomainObjects to represent generality that is complementing ("fighting" with) the much stronger forces for application specific needs. This generality can lead to better application design, but it also provides the ability to move DomainObjects (and their models) between different applications, languages, and industries. This generality of good information models and objects is what MONDO leverages.

By interacting with DomainObjects you either query or change the current state of the model the application is using. A query may be to "walk-through" a structured document, display a molecule in a graphic, or print an account balance. A change may be rearranging document sections, producing a new chemical model, or withdrawing cash from a checking account. In all these cases the ObjectBase is the portion that is concerned only with the information, not its specific presentation (e.g. to a UI) or processing.

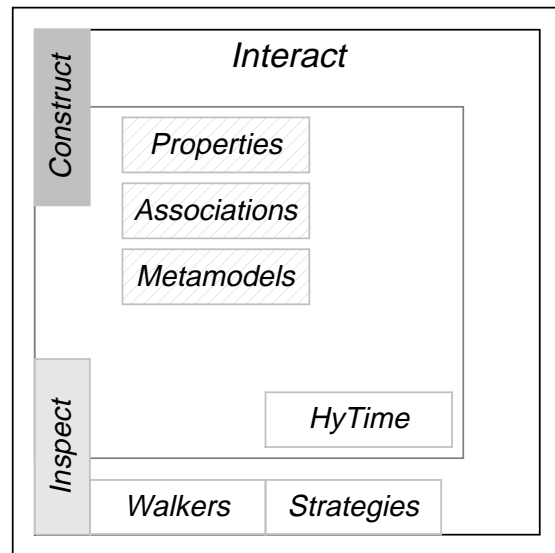
Services

An important view on the ObjectBase is what services it must provide to those around it. The service MONDO strictly requires is the ability to **Construct** objects. MONDO is primarily focused on building Objects out of Recipes. The ObjectBase must provide a way to construct objects for this to be possible. The details can vary: MONDO does not constrain how this is implemented. MONDO frameworks use generic patterns (e.g. the Factory* pattern) and implementations use specific knowledge about how to build objects in the local environment to make this possible. The specifics of construction will be left to the Building Chapter where the details are more required and the context better defined.

A second service MONDO usually requires is the ability to **Inspect** objects. To create a recipe from an object (the Describe process) requires being able to look into the state of an object. This does not mean we want to see the representation of the objects, we just need the information that will allow us to rebuild it. For our Date example this might be the 'iso' string "1997-11-10".

The rest of the services an Objectbase provides to an application can be divided into two categories: those services that improve the capabilities of the model and those services that support applications interacting with the model. The former could include reflective capabilities like properties, associations, and metamodels. The second could include Walkers and Strategies (combined they are a Visitor) for iterating through a model and performing processing on it. We could also have more domain specific services like a HyTime type of addressing and linking model on top of the association services.

MONDO does not need any of these additional services for the other components of MONDO to work. These are completely private to the sphere of the ObjectBase.



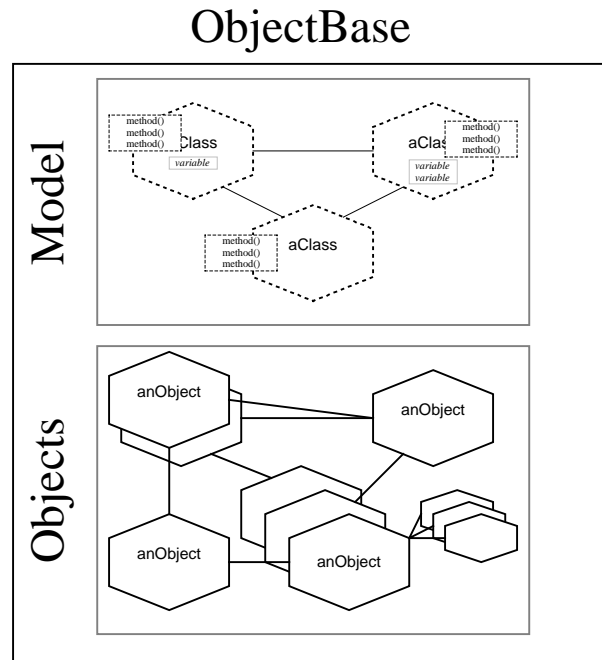
The dominant source of services in the ObjectBase will be from all the people and companies that make frameworks and tools to work with objects. This is the important benefit of the MONDO integration approach: many other people are developing high quality tools that can be accessed within an object-oriented environment. MONDO does provide designs and implementations to help with some of these specific services as part of the functionality levels on top of the base MONDO architecture. These are to help applications that may need this type of functionality or to support the actual implementations of the Inspect and Construct services. Later sections and other documents will go into more details on the MONDO functionality.

Objects and Model

An ObjectBase is a collection of objects but it can be useful to actually model those objects: to describe the constraints and possibilities of what objects are in the ObjectBase and what those objects know and can do. This is usually accomplished by determining what properties are common to sets of objects and what properties need to vary on an object-by-object basis. This separates the DomainModel from the actual DomainObjects and can make building and understanding the ObjectBase easier. This will be discussed in the next section.

* See [Gamma+HJV 95].

It is also usually required in OO programming languages to implement a DomainModel as Classes. In this case you have a clear (if not complete) representation of the Model within the functioning of the ObjectBase and a separation between the Model and the Objects.



A particular language's capabilities with Classes are not always sufficient for expressing the Types of Objects and we will cover other forms of associating Type information with Objects in later sections and chapters.

Programming language specific information

One of the example domains may have looked a bit suspicious: the Beans example. Because Java Beans only make sense for Java, this is certainly not generally useful knowledge that any application can take advantage of. Only a Java interpreter could make use of a Java Bean.

This is true, but we need to separate two things: Ability and Spirit. MONDO has the ability to record any type of information (with the appropriate implementation) and rebuild it again later. This information could be very general or it could be application specific. MONDO does not know the difference.

But the spirit of MONDO is to provide a mechanism to make knowledge available to as many people as possible. This means you should (in the MONDO spirit) first focus on the information that is general (application independent) and then consider the information that is application specific. The later can either be left up to each future application to determine or it could be provided as metadata auxiliary to, but associated with, the core general information.

This will be discussed in more detail in later sections and chapters but a simple example will be provided here. Say we decide that a document needs a Date. The document should simply record the information needed to create a Date:

```
Today (<Date iso="1997-11-20">) is the day that I am going to finish
this document.
```

But what if we want to provide an implementation of a Date that understands the 'iso' information and can then provide the day, the day of the week, and the name of the month? We could at least provide an implementation that prints a nicely formatted version (e.g. "Tuesday, November 20th, 1997"). This would be in case the system that built the recipe did not understand what a Date was.

To do that we only need to associate the Date with a Type and then associate that Type with a possible Java implementation. These would also be in recipes but they would be easily identified as auxiliary and could be ignored or used as desired. An example might be:

```
<Implementation
  type = "Date"
  class = <Class
    name = "COM.chimu.kernel.Date" version = "1.3"
    bytecodes = <Binary encoding="hex" data="cafebabe...">
>>
```

This could either be included in the original file or (more likely) be available in another file or on the internet and pulled in only if the application desires.

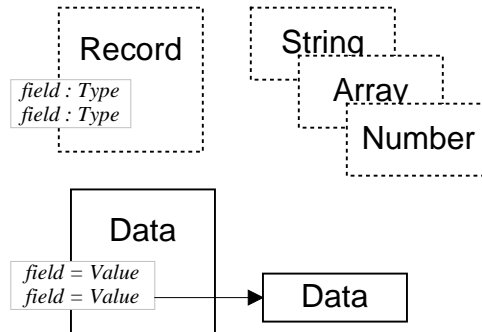
3.2 Data and Information Models

This section provides background on the different techniques for modeling information and structuring programs to use that information. It is provided to show how object-oriented information modeling relates to and subsumes previous approaches. It also defines some terms and exposes some of the notation that this document uses.

Data Structures

The simplest DomainModels are simply data structures. They structure and record data for later direct retrieval. There is no encapsulation of the state of the object and no additional behavior associated with the data type itself. The only rule is that data types must match when assigning to fields. Although these can be considered "Objects" they are shown as very "Square" indicating their simplicity.

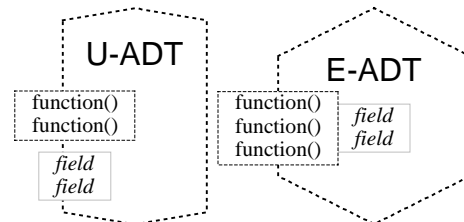
Data structures can be useful in that they are easy to understand and do not allow for more complex interpretations. They are simply structured data. This is also their big deficiency. We can not connect any rules or implications with the data except through the application itself. But if there are general rules they should be embodied in the knowledge, not in the processing of the knowledge.



Abstract Data Types

Abstract Data Types are a major step up from simple data structures. ADTs have a separate external appearance from their internal implementation. This external appearance can provide functions that are associated with the data and operations that modify the data. ADTs are very flexible and can range in appearance from unencapsulated data structures with associated functions (a U-ADT) to fully encapsulated data types where the internal representation is completely hidden (E-ADT).

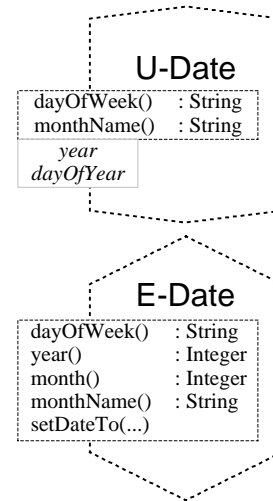
If you look at the notation you can see the transformation that is occurring. A U-ADT has the benefits of connecting rules and implications (derivable information) to the data but it still exposes the fields to direct access and users of the U-ADT will be bound to this implementation. An E-ADT completely encapsulates the fields within it and only exposes functions. This complete encapsulation supports more sophisticated modeling and better system design because we have separated interfaces and responsibilities (which the client cares about) from implementation (which only the service



provider need care about).

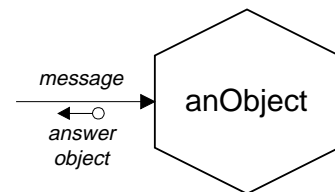
E-ADTs also support knowledge modeling better than U-ADT or simple data types. With E-ADTs clients can say what they need to know or what they want to change, but the E-ADTs in the DomainModel will determine how to provide the information or cause the information state (the objects in the Objectbase) to change. The simplest example is a Date. The unencapsulated Date might look like the U-Date example. The fully encapsulated date would look like the E-Date example.

The E-Date is a better model because we only specify what we know and what we allow to change. We do not specify how we accomplish these responsibilities. If we decide to represent dates with a different data structure (using milliseconds from a date or using Months-ADTs as values) the client does not care. We can also make sure that modifications to a Date make sense for the DomainModel (no `dayOfYear = 400`).



Objects

Object-oriented programming takes the encapsulation one step further than when we use E-ADTs. With E-ADTs, we talk in terms of what functions this computer can execute on an (encapsulated) data based on its data type. With object-oriented programming*, we talk in terms of the operations this object (the data itself) supports. We treat the object as if all implementation details are encapsulated, even how it is able to respond; each object is like a little computer.



All we can do is send messages to the object and see how the object responds to those messages. An Object responds by what it returns from the message and by how it changes the state of itself or (through it sending messages) the objects “around” it (the proximity objects within the ObjectBase). With object-oriented programming a system is built from lots of little virtual computers, each only responsible for its external interface. At every level, implementation details are hidden within the “membrane” of an object.

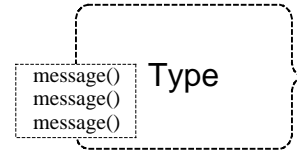
It is important to recognize and understand the benefits of Objects on their own and see how the whole paradigm is different from ADT-s at its core. That is why this section is included. But the next sections will show how modeling with objects looks like working with E-ADTs and then grows from there to be a very rich information modeling environment. This hides but does not diminish the importance of objects and pure object-oriented programming that the modeling is based on.

Object-Oriented Types and Classes

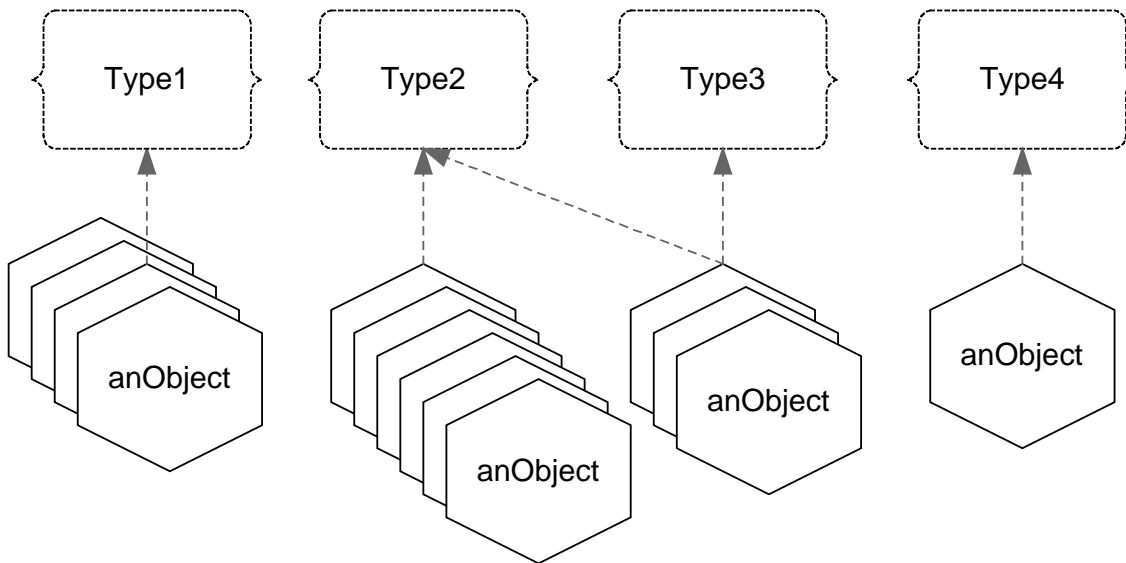
If all Objects could be different, it is a bit difficult for us humans to understand their behavior or build a system based on them. Fortunately, when building even a very complex system we want most objects to be predominantly similar to other objects. We can classify objects with similar properties as part of a Type. By then describing the Type we will be describing the much larger set of objects which conform to the Type.

* Using the term object-oriented programming as intended by Alan Kay, the originator of the word. See [Kay 96] for a full description of the origins and value of object-oriented programming.

What does a Type define? A Type defines the messages that an object could respond to and the semantics of those messages*. This is just like an E-ADT except there are no implications or restrictions on how an object implements a Type. Any number of objects could use different implementations for the same Type and a given object could implement more than one Type at a time. The Type only defines the external responsibilities of an object.



This allows several different possibilities between Types and Objects. We can have objects that only conform to a single Type (the objects under Type1 & Type4). We can have a Type that has objects that are in some way different from each other but are identical when viewed from the perspective of the Type (Type2). We can have objects that conform to more than one Type (the objects of Type2 & Type3). We can have a singleton Type: a Type that represents the behavior of a single object (Type4). Finally, could also have a Type that has no objects currently implementing it, so is “theoretical”.



At the core, Types are just a description of the common external behavior (the messages) that a set of objects supports. Types are a tool that helps us understand objects by describing their similarities. But anytime we use a tool we will want to improve it.

Object-Oriented Information Modeling

Object-oriented information modeling augments the simple Type model discussed above by describing many more characteristics of objects. These characteristics are derived: we still have only objects that respond to messages. But we have analyzed and more precisely defined the different kinds of behavior present in those objects. This results in a number of useful modeling concepts: attributes, associations, operations, subtypes, cardinality, constructors, aggregation, containment, clusters, and perspectives. These new concepts enable very precise, understandable, and expressive specifications of both simple and complex systems.

The importance of information modeling is not that it can be very expressive. It is that it can be appropriately expressive. “The goal of information modeling is to create an understandable and elegant specification of the [rules within a domain]”†. If the domain or the desired model is simple than the information model will be simple. If the domain and the model need to be sophisticated than the information model may be more complex but it can still be very understandable and elegant. Simple models are simple, complex models are possible.

* Which can formalized as a contract between the client and the object. See [Meyer 97]

† [Kilov 94 § 2.6.2]

Choosing the right type of model

MONDO does not require a specific model for your Objectbase, it only requires that the Objectbase use objects. Because object-oriented information modeling works with objects and because it subsumes all the previous modeling techniques, MONDO refers to the DomainModel as an OO information model. But this means all the options are still present, they are just more or less sophisticated version.

This means the model you choose can match your needs and if your needs grow your model can also. Later in this chapter some example models will be shown and we hope to start collecting more of these models

3.3 Example Models

(May have to wait until after services? Separate Chapter?)

Simple property-oriented model

Extended Hypertextual model

Example Company Domain Model

DTD-Based Models (Original, Modified)

SGML's Grove Model

3.4 Services:

3.5 Metamodels: Modeling Objects with Objects

An important aspect to

3.6 Validation

3.7 Summary

