

## 2.1 Modeling and Implementing, Objects and Recipes

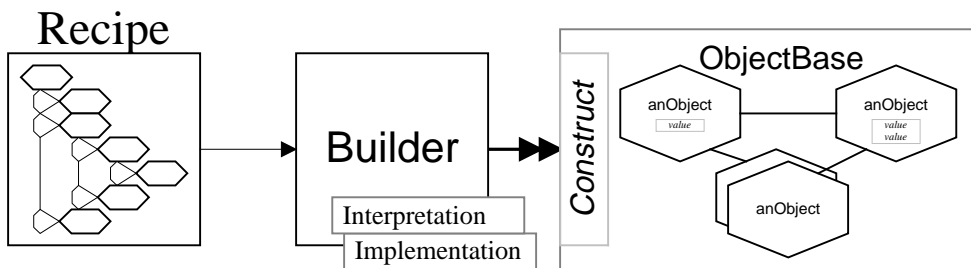
Recipes describe how to build knowledge though creating objects. An important aspect to working with knowledge is to be able to model it. So far, we have assumed the model of our knowledge preexists and only exists in the ObjectBase's DomainModel. Our other choice is to explicitly describe the Model outside of the ObjectBase and then configure the ObjectBase based on that model. With this choice, information will describe its own model (or models) and we will be provided with a lot more capabilities to automatically and universally understand that information.

Similarly, so far we have assumed the implementation of the DomainModel resides solely within the ObjectSpace. Part of this assumption is from a crucial aspect to MONDO: we should focus on capturing knowledge in an implementation independent manner. We can keep this characteristic while augmenting our capabilities with explicit, possible, implementations. We will encode the knowledge of how to implement our model and loosely link them together. This will allow an application to automatically extend its ObjectBase's functionality with sophisticated implementation if an implementation is provided and the application desires to use it.

All of this functionality is easily described and implemented in MONDO because of two powerful characteristics: closure and completeness. MONDO has closure because all transformations within it lead to new objects that MONDO can work with: either recipes or objects. MONDO is complete because both objects and recipes are Turing complete\*. This will enable us to describe Models and Implementations in the same terms as basic knowledge itself, and to extend MONDO's behavior by applying MONDO to itself.

### Interpretation and Implementation of Recipes

A recipe describes how to construct objects. The type of object built from the recipe is completely determined by the ObjectBuilder and the ObjectBase. The recipe describes the instructions but does not say anything about what the result should be. It is up to the ObjectBuilder to **interpret** the recipe as a particular **model** and it is up to the ObjectBase to **implement** that model via Classes or whatever appropriate mechanism.



As an example we could have:

```
<Period
  start = <Date iso="1997-10-01">
  end   = <Date>
>
```

The interpretation of Date and Period is completely up to the ObjectBuilder. It could choose to build an SGML Grove out of it or Java specific classes. The recipe is describing sufficient information to build knowledge but we can only learn about the structure of that knowledge through analyzing the recipe. There is no more explicit information.

\* With sufficient core implementations.

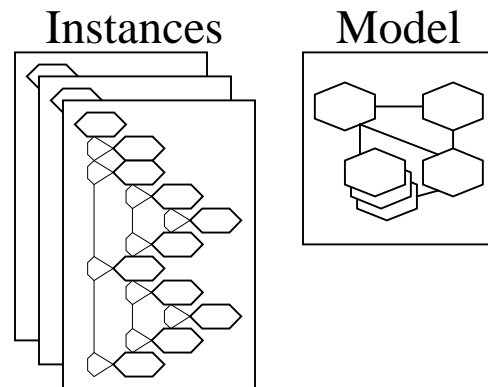
## Recipe Models

The deficiency with the previous example is that it is only minimally self-describing. It tells us what the recipe is, but not what it is allowed to be. It does not describe its own model except through what we can glean from its internal consistency. In some cases this may be fine. We may know that the ObjectBuilder needs no further information because we have a particular type of builder in mind. This might be the case for a properties file (Maps of Strings) or some other well-defined information structure.

Frequently, though, it is better to have the information describe its own model<sup>\*</sup>: the expected types and properties of the objects it is building from a recipe. We will use the term **instance** for the original recipe/objects and **model** for the objects that describe common properties of instances. There are many advantages to providing a model:

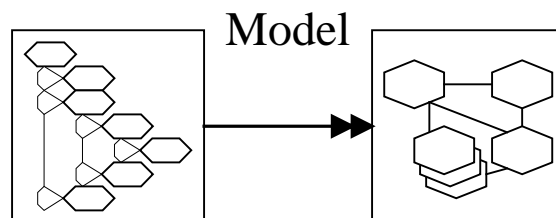
1. Models may be easier to understand than a (probably much bigger) instance.
2. A single model can be used to describe multiple instances, which will make it easier to understand the many from their common aspects described in the model.
3. Models can describe common property information that would otherwise have to be attached to each instance. This explicitly common information can be more intelligently understood by applications.
4. Models can be used to validate whether an instance is compliant with the model. This is very useful for helping ensure the recipe describes (at a semantic level) what we expect it to describe.

The first two points are probably the most important. It is easier to work with a single simple model and the many instances that must conform to that model than it is to work with many unrelated instances. This affects both our ability to comprehend the information and our ability to controllably change it. The third point (common properties) is partially a restatement of the first two, but it is also a reminder that explicit information is easier for an application to deal with. The final point (validation) will be discussed in an upcoming section.



## Modeling Recipes as Recipes

So how do we encode the Model? Models are simply DomainObjects that describe other DomainObjects. So we should be able to use the same approach to construct them as we do with any other DomainModel: use a Recipe. We simply create a recipe for the Model that is separate from our instance recipe.



For our example this might look like:

```
<Model types=(
  <Type
    name = "Date"
    description = "A Gregorian Calendar Date"
    properties = (
      <Property name="day" type=<TypeReference name="Number"> >
      <Property name="month" type=<TypeReference name="Month"> >
      <Property name="year" type=<TypeReference name="Number"> >

      <Property name="iso" type=<TypeReference name="String"> >
    )
    constructors = (
```

<sup>\*</sup> This will become plural ('models') in an upcoming section.

```

        <Constructor {}>
        <Constructor {"iso"}>
    )
>
<Type
  name = "Period"
  description = "A range of dates"
  properties = (
    <Property name="start" type=<TypeReference name="Date"> >
    <Property name="end" type=<TypeReference name="Date"> >
  )
  constructors = (
    <Constructor ("start" "end")>
  )
>
)>

```

In the above we describe the Types “Date” and “Period” sufficiently enough for a human to understand their properties and a computer to model them.

## Models and Objects and Recipes

The Model is not describing the Instance recipe itself as much as the resulting Objects from that Instance recipe. We can see that the resulting objects will have certain properties and whatever other modeling information we will need. The part that is most important to the recipe itself is the “constructor” information. This determines what construction possibilities are valid for this model and, with it, what recipes can be used to construct these types. A Date can be constructed without any parameters and a Period can be constructed with only a “start” and “end” parameter. If we assume a ‘<Date>’ recipe produces a ‘Date’ type of object, we now know the rules for valid parameter and ingredients for that ‘<Date>’ recipe. We can now validate recipes that are supposed to be for this Model.

## Model Completeness

We may choose to try to provide a complete description of the information model within our recipe or we may leave some of the Model up to the ObjectBase to determine precisely. This depends on our knowledge of the Model and all its possible uses. We could also choose to create different Models that are based on each other with different levels of detail.

## Connecting Instances to Models

We now have a recipe that describes a Model and we have a recipe that describes an Instance of that Model. We have several choices for how to connect the instance to the Model. First, we could simply combine them together into one large recipe:

```

<UseModel model=<Model types=(
  <Type
    name = "Date"
    description = "A Gregorian Calendar Date"
    //...
  >
  <Type
    name = "Period"
    description = "A range of dates"
    //...
  >
Model> UseModel>
<Period
  start = <Date iso="1997-10-01">
  end = <Date>
>

```

The UseModel recipe will tell the ObjectBuilder to use the model object (our Model from above) as the Model for the subsequent recipe. This is a straightforward approach but is probably not what we want. The model was meant to be useable by multiple instances and to be looked at on its own. Embedding the model with the instance prevents these two abilities. The better choice is to keep the model recipe separate and to just refer to it:

```

<UseModel fileName="exampleModel.OML">

```

```
<Period
  start = <Date iso="1997-10-01">
  end   = <Date>
>
```

In this form the UseModel recipe will build the object in the file and then use that object as the Model for the current instance. Now we can have multiple instance files use the same model and we can study and augment the model by studying a single file.

## Wide-scale sharing of Models

The important part of sharing a Model is the Model object itself. In the previous example we shared models by having a common recipe file that would create the same Model object. We can do this over a wider area by treating the Internet as our file system:

```
<UseModel
  url= "http://www.chimu.com/projects/mondo/models/exampleModel.OML"
>
<Period
  start = <Date iso="1997-10-01">
  end   = <Date>
>
```

Both of these UseModels are simply convenient “macros” for common uses of more basic functionality in MONDO. We simply need to create a Model object and use it as the model of this instance. All the above variations are equivalent to:

```
<UseModel
  model= <BuildAnObjectInAParticularWay>
>
```

Where, the BuildAnObject is replaced with (for example):

```
<Build url=<URL "http://www.chimu.com/projects/.../exampleModel.OML">>
```

## Referring to Model Objects

Building a Model object directly from a file is conveniently direct but certainly not the only approach. We can also use the inherent ability of MONDO (and objects in general) to refer to other objects by Reference or Proxy. For our example, we can create a reference to the Model object we desire and let the ObjectBase (i.e. other objects) figure out what object we are referring to and fetch it.

```
<UseModel
  model=<FindReference
    <Reference id="www.chimu.com/mondo/examples/exampleModel">
  >
>
<Period
  start = <Date iso="1997-10-01">
  end   = <Date>
>
```

The difference may not be very noticeable, but we have separated the name/identifier of the Model object “...exampleModel” from the actual location that the model is stored “...exampleModel.OML”. This allows us to change model locations or even our approach for building the model object (from a file on demand to prebuilt from a database or serialization). The only important part is that <FindReference> returns an object.

We may have different types of references also. If our system builds Model objects from SGML DTDs we could use a DTD reference:

```
<UseModel
  model=<FindReference
    <DTDReference sysid="-//HaL and O'Reilly//DTD DocBook//EN">
  >
>
<Book
  title = {MONDO Design Document}
  preface = {<P{This document describes MONDO, a generalized
    architecture for encoding, modeling, and processing
    information.}P>}
  parts = (...)
```

```
>
```

And the equivalent of an SGML Catalog file would be:

```
<DtdCatalog entryReferences=(
  <Pair key = "-//HaL and O'Reilly//DTD DocBook//EN"
        value = <Reference url=<URL
                "http://www.chimu.com/projects/.../docbook.oml"> >
  )DtdCatalog>
```

The sole responsibility of a DtdCatalog object is to resolve DTD-References into objects. That is all the client cares about. The DtdCatalog can choose to build all the objects from the references when it is created or it can build them just as they are needed. It can also cache and reuse the objects for multiple calls to the catalog instead of “pure build” approach, which will always build a new Model object.

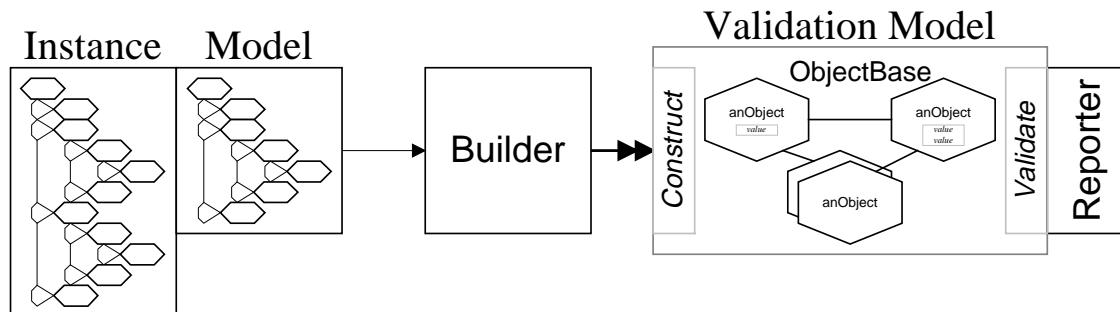
Using identifiers and references is clearly a good way to go for widespread and stable Models. The recipe instance simply refers to a Model object by name/id and then the rest of the system (e.g. the reference and catalog) is responsible for tracking the reference down and returning the actual object.

## Validating against the Model

There are two major approaches to validating an Instance against its Model: separate validation and inherent validation.

### Separate Validation

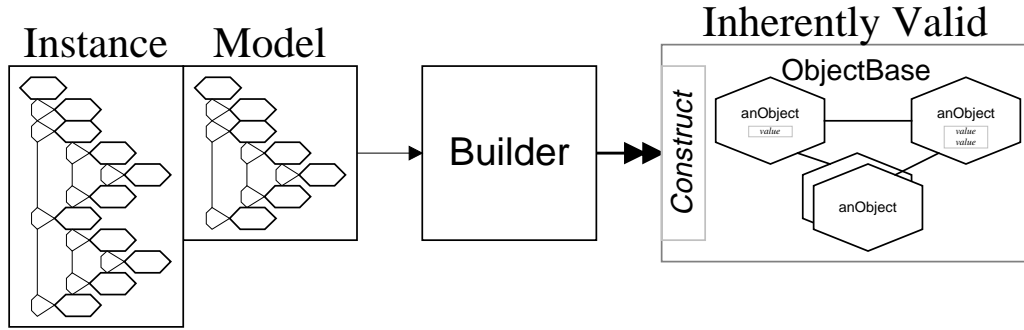
With separate validation, we have a model that is particularly designed to keep all the model information and use it to validate the particular instance. In this approach, validation is a separate process or stage from building our normal working DomainModel.



This approach allows us to have a simpler DomainModel than our ValidationModel. We may want to verify that a Date recipe has all the properties our Model says it should have, but when we implement the DomainModel we will be using a system supplied class (e.g. java.util.Date) and it knows nothing about our Model. The disadvantage is that the ValidationModel is not the same as our DomainModel, so the two models have to be managed separately (but also see combining the approaches below).

### Inherently Valid

With inherent validation, our normal DomainModel is intelligent enough to verify whether a particular instance is valid and to only modify the ObjectBase if the instance is valid. This is similar to a database transaction: the database is only changed if all the data is correct by all the rules of the database model.

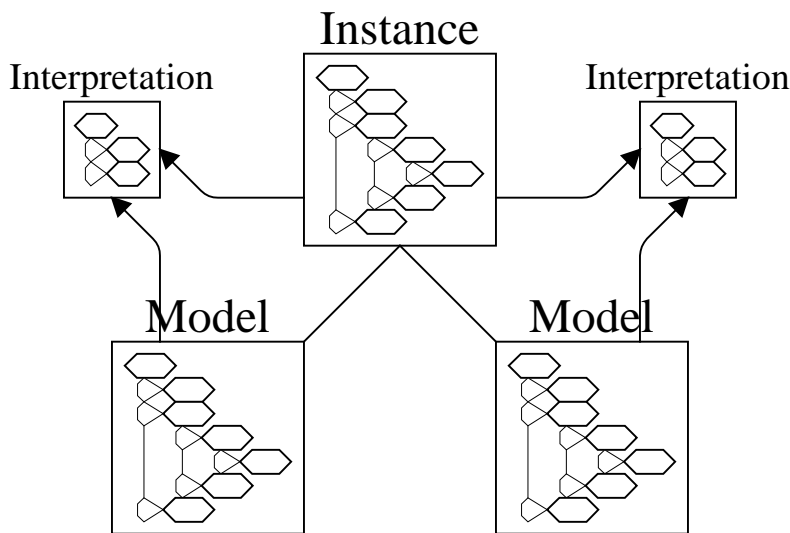


### Combined Approach

We can also choose to combine the approaches: we can have an inherently valid domain model but also have some extra information attached to it when a full validation is desired.

### Multiple models and Interpretations

Any recipe can have multiple models that apply to it. A recipe may be written with a particular model in mind, but that does not prevent other models from applying either concurrently or in the future. This indicates that separating the recipe from the commitment of which model applies to it can be a good idea.



This separation can be accomplished by using **interpretations**. An interpretation is the combination of a recipe instance with its model. Both the recipe instance and the model are written independently of each other so they can be combined in different ways. Each of these combinations simply identifies the recipe instance and the model that will be used for it.

```
<UseModel fileName = "exampleModel.oml">
<Build fileName = "example.oml">
```

This now gives us three choices: separate interpretations, combined instance and interpretation but separate model, and combined instance and model. Using separate interpretations provides the maximum flexibility. Combining the interpretation into the recipe file but keeping the model separate provides slightly less flexibility but is good for many applications. It is also easy to fix (make more general) by simply removing or disabling the <UseModel> within the recipe. Putting the model physically into the instance recipe provides no flexibility and makes reuse/sharing of the model impossible. One of the first two choices is generally the best.

## Language Independent Implementation

The previous examples enabled the recipes to describe their content and the application could take advantage of that information to validate the encoding or provide some Type intelligent behavior. How can we take the next step: supporting more extensive DomainModel behavior that is application independent?

More extensive behavior will require real code to be loaded by the application, so at some point we will have to couple to a specific development language and environment. The spirit of MONDO is to encode information into as general a format as possible. Generality is accomplished by separating the general information from the specific. This is what we did with the different levels of the information for this example: objects, model, and now implementation. Generality is also accomplished by keeping all types of information -- even application/language specific information -- encoded in the same format. This allows applications that do not “deeply” understand the information to still work with it on a simpler level (e.g. formatting and printing it).

### Java Encoding

Our example will use Java as its first implementation language. Our example has two types, ‘Date’ and ‘Period’, which we will need implementations for. For Date we will use the supplied Date implementation in the ‘java.util’ package. So all we need to do is identify the class:

```
<JavaClass
  name="java.util.Date"
>
```

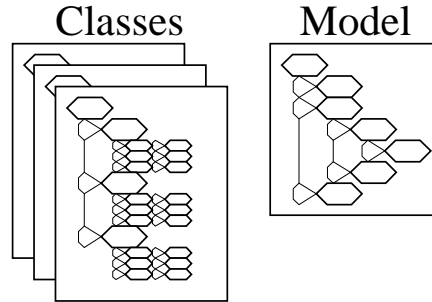
For Period we will use a ChiMu class, “COM.chimu.kernel.basicTypes.Period”. The situation with Period is different from “java.util.Date” because our Period class may not be present on a particular machine. To actually encode a Period class we will need to give the application enough information to build the code for the class. We have a number of choices for providing this information: source code, compiled code (e.g. DLLs), bytecodes (for Java, Smalltalk, etc.), or even through detailed code-recipes. The right choice may depend on the language and the desired portability for that language.

A good choice for Java is to use bytecodes for the implementation information. This allows concise expression of the class information and is easy to decode. Our class might look like this:

```
<JavaClass
  name      = "COM.chimu.kernel.basicTypes.Period"
  version   = "v0.1"
  vmRequired = "1.1"
  description = "This is a simple Period which uses java.util.Dates
                as its start and end values"
  bytecodes = <Binary encoding=hex
                [[cafebabe0003002d000e07000907000c010015284c6a6176612f6c616e
                672f4f626a6563743b2956010004436f646501000d436f6e7374616e7456
                616c756501000a457863657074696f6e7301000f4c696e654e756d626572
                5461626c6501000e4c6f63616c5661726961626c65730100095365747570
                506f6f6c01000e5365747570506f6f6c2e6a61766101000a536f75726365
                46696c650100106a6176612f6c616e672f4f626a65637401000574657374
                31020100010002000000000010401000d000300000001000b0000000200
                0a]]
  >
>
```

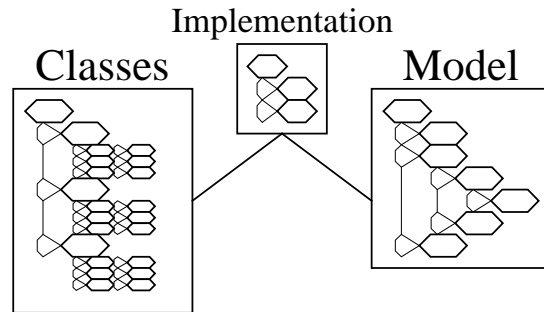
This MONDO class recipe has a couple of interesting qualities. Foremost is that it is readable to both Java and non-Java systems. A non-Java system may not understand the bytecodes, but it can understand everything else and work with the information usefully. The recipe is also more useful to a Java system than a normal ‘.class’ file. For example, the recipe allows a Java program to identify whether this class can run on the current VM and whether we already have a newer (and compatible) version loaded. The Java program can decide this all before loading the Class itself.

So we have an approach for representing language specific implementations in as general and self-describing a format as possible. For any given model we can have multiple of these implementation classes that will function on different platforms and with other differing characteristics. The classes themselves are self-describing so we can make some choices based on their knowledge. Our next issue will be to encode what possibilities exist between the Models and the Classes and to help our particular application make a choice from them.



### Implementations

Just because we have a possible implementation should not alter the original information of our instance recipe and model recipe. These are still good general descriptions of information. But we want to be able to associate our Model with the possible implementations so an application can take advantage of them. If an application wants to use our Java implementations it should be able to find them and pull them in.



This association we can represent via an Implements relationship between the Type and the Class. For example:

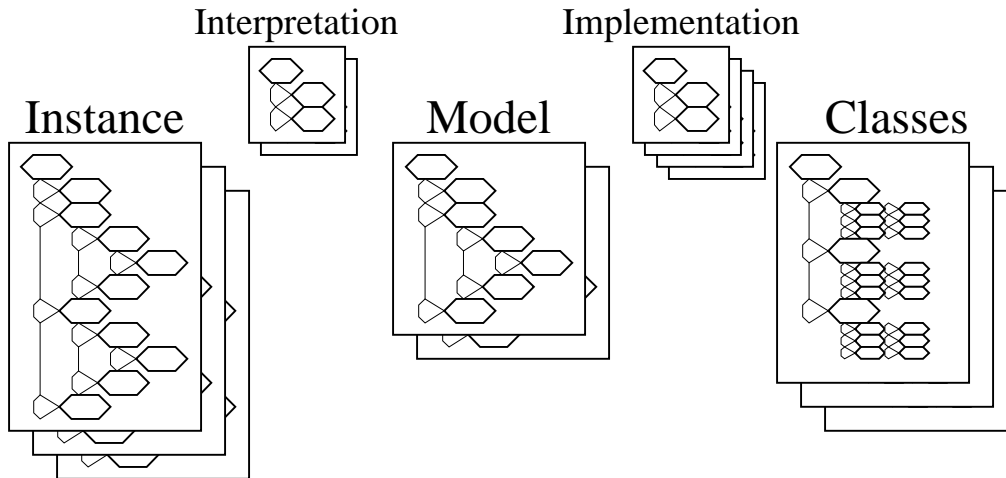
```
<Implementation
  type      = <TypeReference name="Date">
  language  = "Java"
  class     = <ClassReference name="java.util.Date">
>

<Implementation
  type      = <TypeReference name="Period">
  language  = "Java"
  class     = <ClassReference name=
              "COM.chimu.kernel.primitives.Period">
>
```

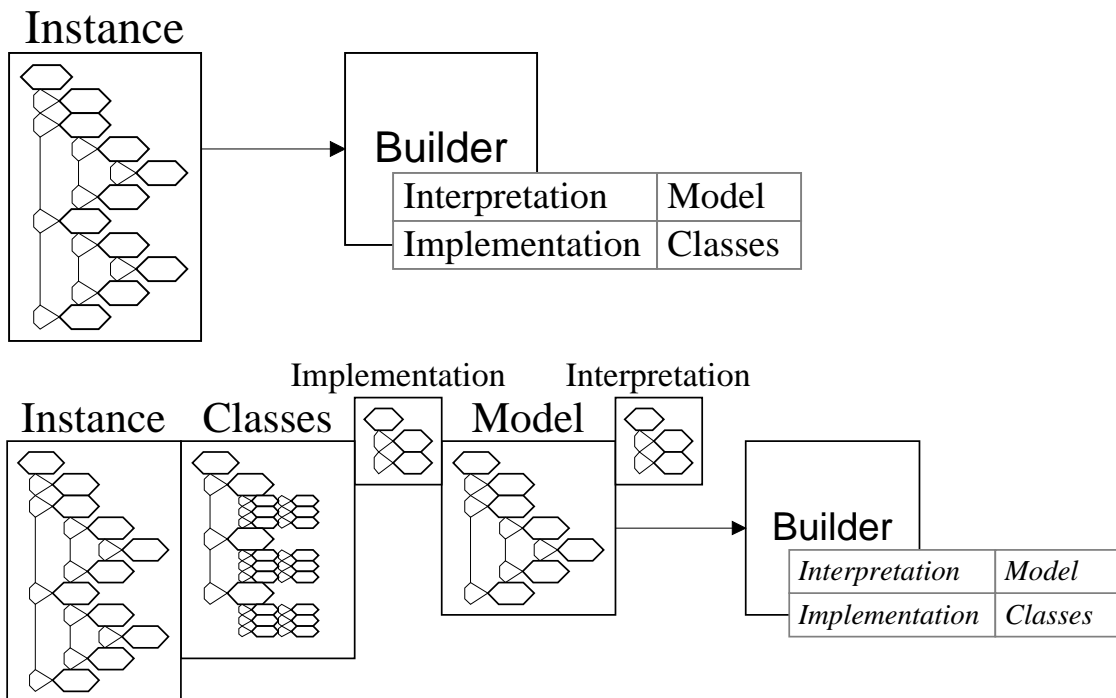
This would associate the objects representing the Types with the objects representing the Java Classes. The Implementation must include enough information that a particular application can choose whether it can and wants to use that implementation. In the above case all we explicitly have is the language characteristic, but we also can verify that all the Classes are able to work with the particular VM. We may also need to bundle classes together because they must work together.

### Possibilities

We have progressed from just having a single Recipe instance to having recipes with possibly multiple interpretation models and with each model having multiple implementations.



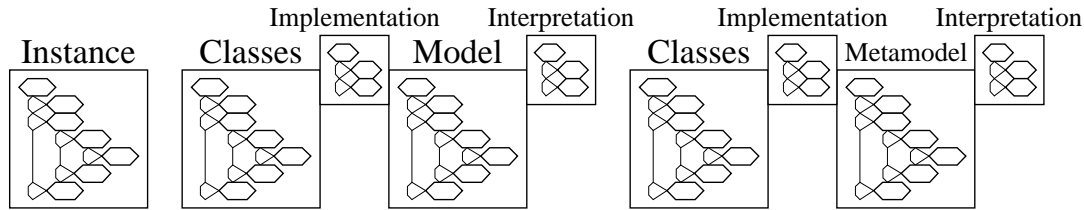
This full complexity is not required; it is just possible if we need it. We have the ability to progress from an application where the builder assumes all the properties of an instance (i.e. its Interpretation and Implementation) until we reach the point where everything is provided explicitly to that builder.



Note that the “everything” version still has the Builder with the same responsibilities, they are only significantly lessened. This is because the builder is still making assumptions, even though it is being given much more information on interpreting and implementing the instance recipe. Some of these assumptions result from insufficient information in the Model and Classes. Others result from the Metamodel problem.

## Metamodels

But even when “everything” is provided to the Builder it is not really everything. We still have built-in assumptions about how the Model and Classes will be interpreted and implemented. We need Models (Metamodels) for our Models and Implementation Classes for those Metamodels. These have to either be assumed as part of the Builder and DomainModel or we would again have to provide recipes for them.



The Modeling process can go on ad infinitum. At some point we need to explicitly encode an interpretation so we have a turtle to stand on.

## Summary

This section showed how MONDO can:

1. Model recipes using Model objects (and create Model objects from recipes).
2. Loosely or tightly connect Instances to Models.
3. Support direct or indirect retrieval of Model objects.
4. Store implementations in a generally understandable (and platform executable) format
5. Support thinking about the implementation before committing to it.
6. Loosely or tightly connect models to implementations.
7. Continue the Modeling at the Metamodel and MetaMetamodel
8. Support the right level of modeling and validation for information
9. Allow the model to grow or migrate without affecting the instance information

All of these abilities can be useful to applications and MONDO lets you choose what capabilities you need for your specific information and applications. They are also simple to understand within the closure and completeness of the MONDO concepts. Recipes build objects, objects can describe other objects, so recipes can describe other recipes.

## Models

Models are important because they allow common characteristics of recipes to be described and enforced. Recipes can be validated against a model and shown to comply or not. A model can also provide the application with more information about a recipe that is useful for working with it. The model can pool common properties that are recipe independent or can augment existing recipes with extra information. Validation and this extra information are all in terms of the DomainModel so applications can have very good control and expressiveness for the rules and modeling considerations. The DomainModel gets smarter in its own terms.

Because Models are just Objects with a particular purpose, we can take advantage of all the normal abilities we have with objects inside MONDO. We can have references to Models in our recipe instances and those references can either be direct (build this recipe) or indirect (fetch the object with this name/id). Model objects can be built from any number of encodings (DTD or MONDO). Or the objects may have been cached, deserialized, or fetched from a database.

## Implementations

MONDO can also be used to enhance the DomainModel itself with new functionality. We can provide information about possible implementations and an application can decide whether it can and wants to use the implementations to enhance its own implementation. This can provide the base for very interesting capabilities on top of MONDO: language independent objects and agents.

## Conclusion

Simple closure and completeness produces a lot of functionality\*. By keeping everything that MONDO does in terms of recipes and objects, a lot of capabilities come for free or can be leveraged from well know designs, patterns, and implementations. Generic Modeling and Implementation are two of these capabilities.

---

\* As programmers of LISP, Smalltalk, and similar languages well know.