The Objectiva Architecture

Francis Anderson (francisa@altinet.net) and Ralph Johnson (johnson@cs.uiuc.edu)

Introduction....2
Notation....4
Billing Overview....6
A Framework for Business Objects....9
The Objectiva Business Model Domain....10
Country....12
Operational Level....16
Region....16
Entity....22
Data Value....25
Node....28
Entity Context....32
Knowledge Level....33
Entity Type....33
Attribute....38
Entity and Data Major Components....42
Continuous Data....45
Discrete Data....49
Complex Data....55
Discrete Collection Attribute....60
Data Type....63
Framework Development....66
Relationship....67
Use Cases....70

Introduction

Objectiva is a black-box framework for telecommunications billing. "Black-box framework" means that it lets you build applications primarily by reusing existing classes and does not force you to create new ones. "Telecommunications billing" means a system that produces bills for a telephone company. Objectiva makes it possible to quickly produce billing systems for all kinds of telecom services, including cellular, PCS, local number portability, conventional local and long distance, and satellite services. It also makes it possible to quickly customize an existing system to respond to changing conditions and to provide new services. It is a "convergent billing" system that makes it possible for a single billing system to handle any kind of telecommunications service.

A billing system has many parts, some technical in nature, and some that solely implement the business rules of billing. The purpose of the Objectiva architecture is to organize the parts of the system as effectively as possible in order to maximize their reuse. In Objectiva, as in Smalltalk, everything is an object, with all the jargon that one expects in order to be fully object-oriented: inheritance, encapsulation, polymorphism, responsibilities, collaboration, etc.

The Objectiva Architecture consists of ten domains:

- some are technical in nature (Hardware, System Software, Common Services, Operations Model and the User Interface);
- some are oriented more towards behavior (Business Process and User Conceptual Model);
- some are oriented more towards structure (Business Model and Domain Model Engine);
- one is a set of tools (Development Services).

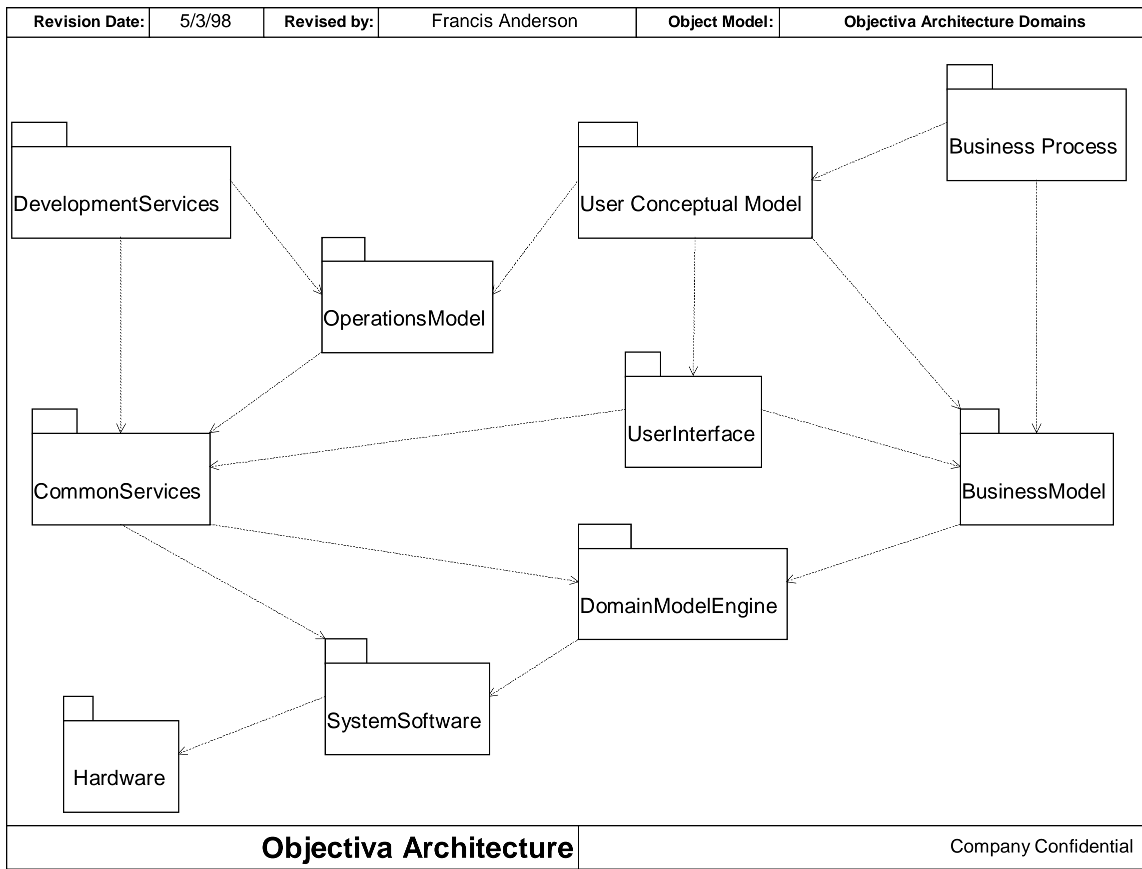

Figure 1: Objectiva Architecture Domains

The Objectiva Architecture Domains are depicted in Figure 1as a UML Package Diagram. In one sense, the domains can be thought of as the layers within Layered Architecture [POSA], but this implies only one specific relationship (messaging) between the domains. It is better to think of the dependencies as depicting how we build the architecure, rather than just how the domains communicate. Once we have hardware and system software, we build a domain model engine, from which we construct a business model and common services to the system software. These common services are applied to the different environments in the operations model, a special set of services being required for the development environment. The business model requires an interface for the various user roles and metaphors defined by the user conceptual model, which implements the business process.

Although these domains area at a very high level of modeling, they have a direct representation in the implementation of Objectiva. Figure 2 shows that the domains are directly implemented as *ENVY* Configuration Maps. This is one of the goals of Architecture: to provide configuration at a sufficiently high level that component-based development becomes possible.

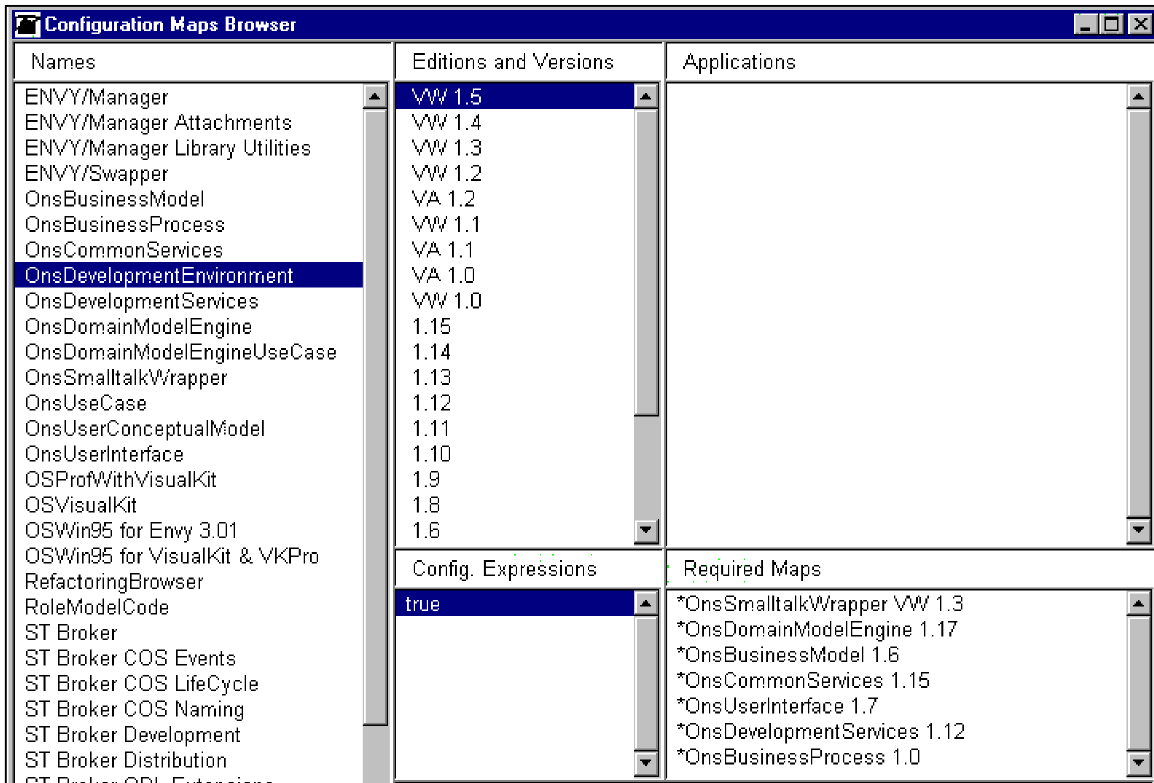

Figure 2: Objectiva Domains as *ENVY* Configuration Maps

To begin with, we will concentrate on the Business Model, and how it uses the Domain Model Engine (DME) to express business rules. The DME is the most abstract of the domains, and lies at the heart of the Objectiva Architecture.

Objectiva keeps track of a company's customers. This includes their addresses and other contact information, the agreements that each customer has with the Enterprise (which can change frequently), the network events that cause a charge (like a local or long distance call, a page, or an e-mail message), taxes, discounts, invoices sent, and payments received. It manages equipment that is being rented or purchased, which means not only charging for it, but keeping track of its location and managing an inventory of equipment available for rental or purchase, and scheduling repairs on equipment that is broken. Objectiva manages products, which are combinations of the various pricing plans that a company is offering to customers. It connects to other systems for accounting, to get network events, and to load subscriber information on the switch. Objectiva is a complex information system, but it is made up of a fairly small number of highly reusable classes. This is a key to its flexibility and power.

This paper describes the architecture of Objectiva, and assumes you know little about telecommunications billing, but a lot about modern OO design, patterns, and Smalltalk.

Notation

Objectiva uses several hundred classes, and we'll be looking at many of them. All the classes added by Objectiva have a prefix of "Ons", for Objectiva Name Space. Thus, class names all look something like OnsNetworkAuthorization. It can get tedious to read a document with names like that, and it seems unnecessary, since the names were chosen to read well. So, class names are broken into their individual words, the "Ons" is omitted, and a special font is chosen. Thus, the class named "OnsNetworkAuthorization" will be written as NETWORK AUTHORIZATION. Smalltalk code will of course use the real Smalltalk class names. This should make it easy for you to go from the document to the code, but should keep the document easy to read.

Billing Overview

Billing means both calculating charges for an individual event, and computing discounts and taxes on the total. In both cases, the billing is event-driven:

- For an individual event, the trigger is externally generated through the subscriber's usage of the network, or on the completion of an order (e.g. installation charge).
- For the computation of monthly charges (e.g. subscription charge) discounts and taxes, the events are internally generated through the cyclical closure of a period. Agreement period closure, which triggers the calculation of monthly recurring charges and discounts, and account period closure, which triggers the calculation of taxes and preparation of an invoice, should be thought of as being independent events. Frequently, however, the two events are combined, since this makes their explanation through a paper invoice much simpler.

Billing is accomplished by rating an event (calculating the charges), and posting the results (a posting with charges) to an account. The rating of network events is the most computationally intense part of Objectiva because it is done for most transactions. It is common for a system to rate a million network transactions a day, and some rate hundreds of millions. Cycle closure is the second most computationally intense part of Objectiva. Together, these two activities account for more than 99% of the transactions in a typical telecommunications billing system.

Rating can be complex, and there are many schemes (price plans) for rating. The charges for a call can depend on the location of the caller and receiver, the agreements of the caller and receiver (think of MCI's "friends and family" program), and the time of day. A cellular call can have airtime, roaming, long-distance, and landline charges. In contrast, ship to satellite communications has a number of different terminal types, each of which offers a different combination of voice, fax, telex, and e-mail service. The rating of a call not only depends on the type of call, but also the type of equipment on either end. Objectiva must handle all these rating schemes.

However, rating makes up only a small part of Objectiva. Most of Objectiva is involved with representing more universal concepts like Business Party (Organization), Region, Network, and Account. These are represented in the architecture as "business objects." Most of Objectiva is devoted to representing business objects, editing them, maintaining relationships between them and constraints on their attributes, storing them in a database, and keeping historical information on them. Because Objectiva supports these standard functions so well, the rating algorithms are easy to change.

The first step in billing is to load information that rarely changes from outside sources. Some of this information is about regions and networks, such as the fact that the 217 area code is in Illinois and Illinois is in the United States. Some of this information describes legal phone numbers or other "network authorization" numbers. In any case, this information is provided partly by various international telecommunication organizations,

and partly by online maintenance. Nevertheless, it has to be loaded into an Objectiva-based billing system.

The second step in billing is loading information about the service provider (the Enterprise) such as the various price plans offered through customer agreements and the rating rules for each price plan. This information is constantly changing. However, for the sake of discussion, we will assume it is done before billing starts.

Once a billing system is set up, it performs six steps over and over. The steps can be taking place simultaneously, but the results of one step affect the following steps.

1. Edit service order. Setup a new customer, or change the agreement with the customer.
2. Rate CDR Batch. Calls and other network events that affect billing are grouped into batches of CALL DETAIL RECORDS. Processing these events consists of making NETWORK EVENT objects from the externally supplied data, rating these events, and then charging them to the appropriate account.
3. Agreement period closure. Some transactions (such as monthly fees for equipment charge, and the calculation of discounts) occur periodically based on the customer's agreement. Periodically these transactions must be created and charged to the customer's account.
4. Account period closure. Generate taxes and any other charges that depend on the total invoice, and produce the invoice for a customer. Sometimes Objectiva prints the invoices directly, sometimes it hands them off to a legacy system that prints them.
5. Payment. Receive payment.
6. Adjustment. Change old billing information and make all adjustments.

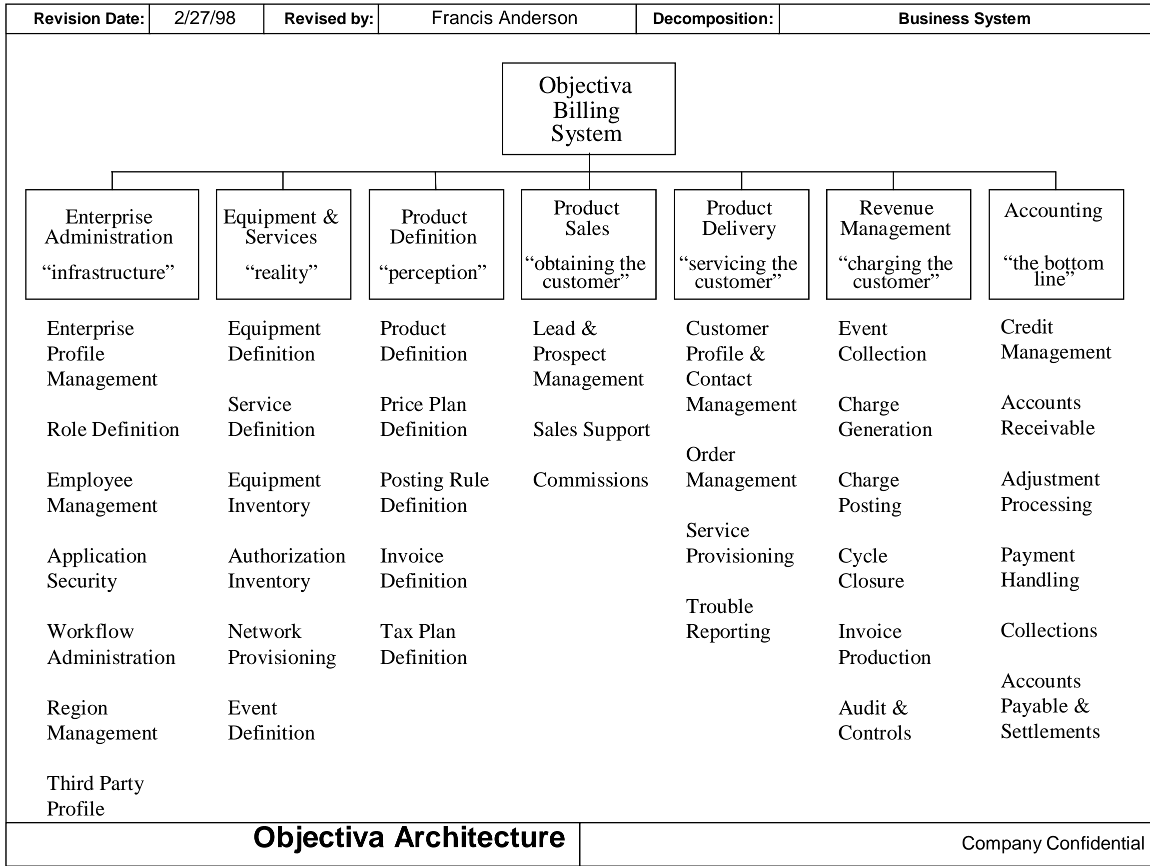

Figure 3: Business Process Domain

The Business Process Domain (Figure 3) shows the decomposition of Objectiva into business systems and subsystems. It depicts the functionality and requirements of the system with which business people are comfortable, and it is the result of a significant amount of business analysis.

This analysis is necessary for the development of a successful business system, but it is not a technical activity. The Business Process domain is the responsibility of the business, and standard Business Process Reengineering (BPR) and quality improvement techniques should be applied to its optimization. Only if the business defines the critical process and product indicators within the Business Process domain, can the Business Model provide the required feedback to keep the overall system operating at peak efficiency and effectiveness.

In short, billing consists of processing transactions and generating invoices. Billing is difficult because the kinds of transactions and the rules for processing them are constantly changing, and are different for every company. The key to Objectiva is hard-coding the part of billing that does not change and making the part that does change very flexible so that it can be tuned to the Business Process that a particular Enterprise needs to be most competitive.

A Framework for Business Objects

Billing requires lots of information. For example, taxes depend on the location of the customer. There might be city taxes, state taxes, and federal taxes. The cost of a call also depends on where the call originated. Therefore, the system must know the location of both the call and the customer. The charge to a customer depends not only on the transaction, but also on the agreement that the customer makes with the Phone Company. Thus, modeling the information needed to rate a phone call takes a lot of objects.

The high-level structure of these objects is always the same, but the details are always different. For example, rating a network event (e.g.. phone call) involves the following:

1) finding the network authorization for the event (e.g.. the phone number of caller),
2) finding the price plan for the network authorization,
3) creating a posting for the event,
4) iterating over each part of the price plan and finding out how much it will add to the charge for the event,
5) adding these charges to the posting,
6) applying the posting to the billing account.

But different kinds of telecommunication services have different kinds of network events, network authorizations, price plans, and charges.

Even the simplest objects can vary. For example, you might think that a phone number is a phone number, and wonder why Objectiva uses a fancy name like “network authorization” for something that seems pretty simple. But a phone number is just a special case of a network authorization. In general, a network authorization states the right by which a call was able to use the network. It is usually either the originating or terminating party in a call. From the network authorization, we can determine the billing authorization which keeps track of the pricing plan used for rating the call, knows who gets the invoice, the features they are willing to pay for, and dates that it is effective. Some of this information is part of the core billing functions, but other information is not. The core is unlikely to vary while the rest varies a lot. For example, a regular phone line might support call forwarding and might allow certain kinds of incoming or outgoing calls to be barred. Satellite service does not support these features, but supports noise muting. None of these are used directly by rating algorithms, though they might be used to choose a billing plan or to “provision” the switch.

The Objectiva Business Model Domain

As shown in Figure 1, the Business Model depends on the DME. This is due to the fact that the DME is used to store those business rules that can be expressed as knowledge level instances. This is the crux of the notion of a black box framework – to increase functionality by adding instances of objects, rather than lines of code.

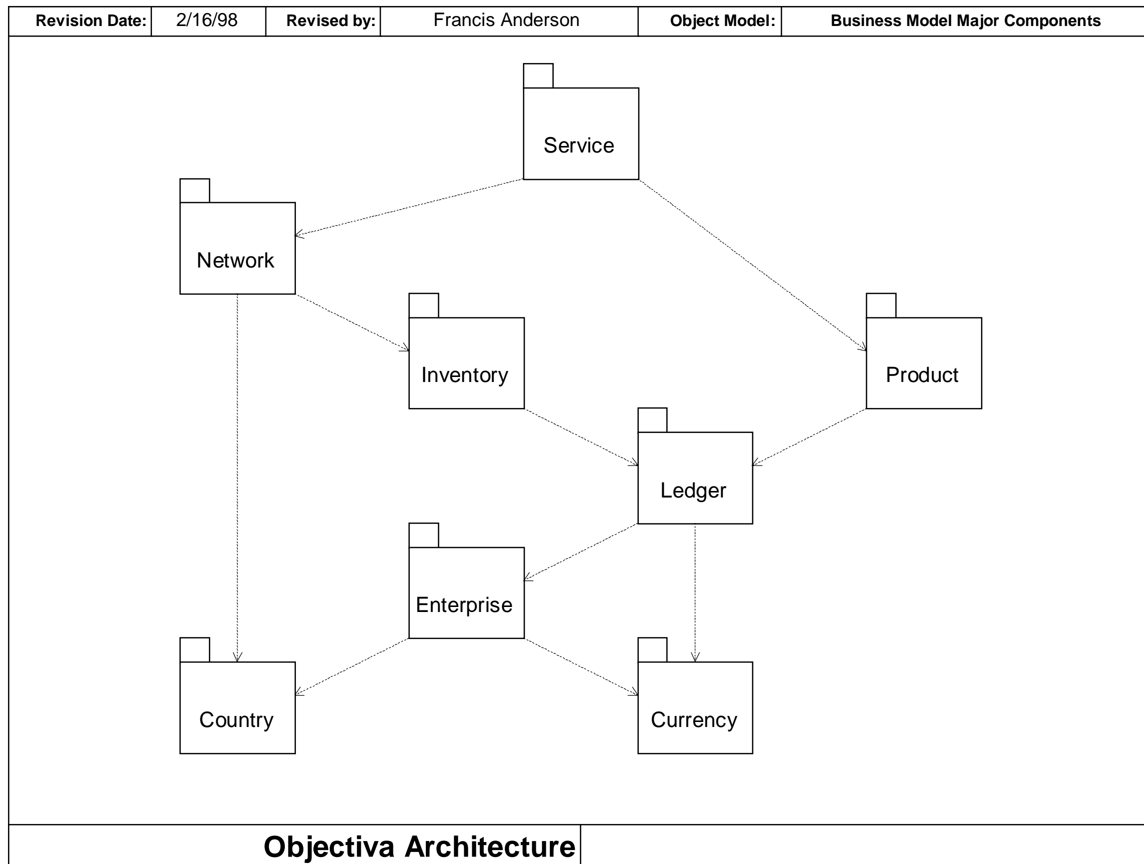

Figure 4: Business Objects

The Business Objects (Figure 4) are the major components of the Business Model required for Telecom Billing. Each of the Business Objects is implemented as an *ENVY* Application (e.g. COUNTRY APP); the dependencies between the Business Object packages are implemented as *ENVY* prerequisites.

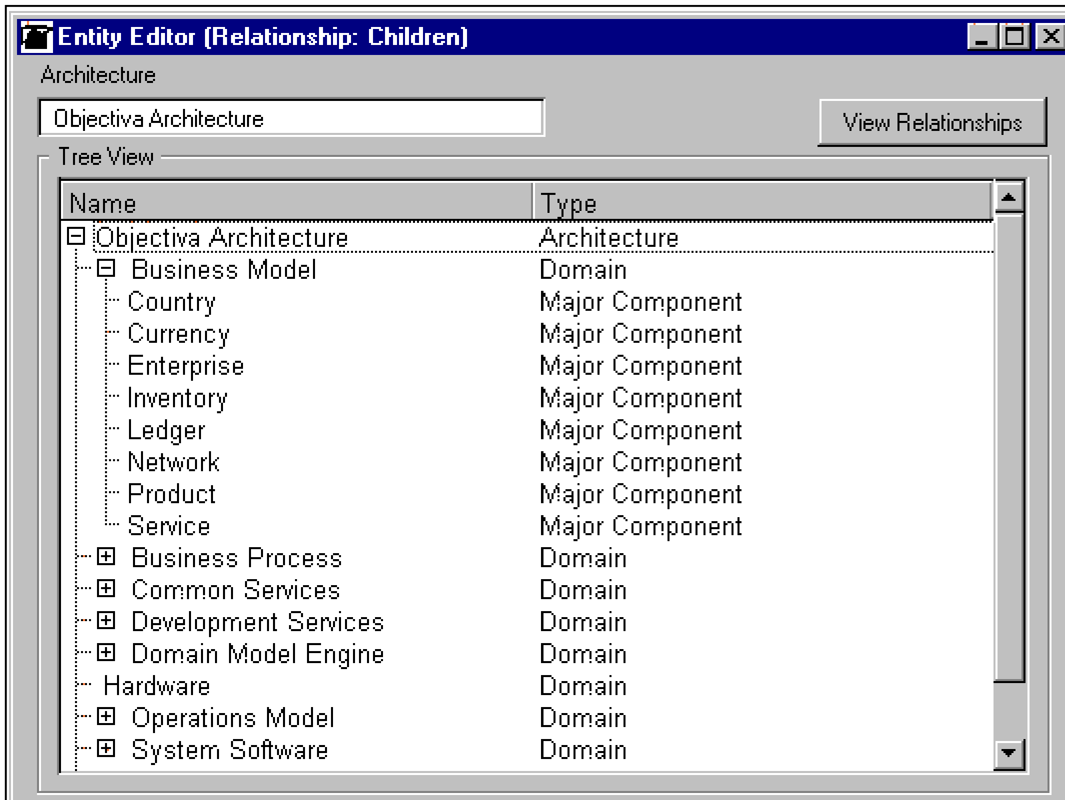

Figure 5: Objectiva Architecture Entity Editor

The Objectiva Architecture Entity Editor (Figure 5) shows the decomposition of the architecture into domains, and the Business Model Domain into its Major Components (Business Objects). We will be using this outline view in the Entity Editor to look at a number of different types of entity within Objectiva.

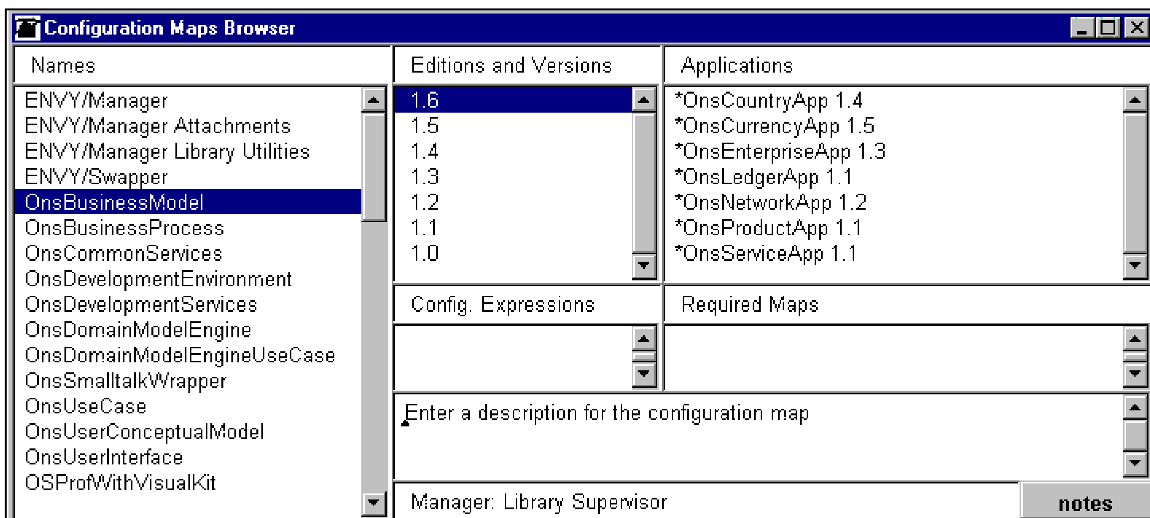

Figure 6: Business Model *ENVY* Configuration Map

The Business Model *ENVY* Configuration Map (Figure 6) shows each business object implemented as an *ENVY* Application.

Country

The first business object to describe is Country, and we will be using it as the example to describe how Objectiva's Domain Model Engine (DME) implements the Active Object Model pattern by building on the Entity major component of the DME.

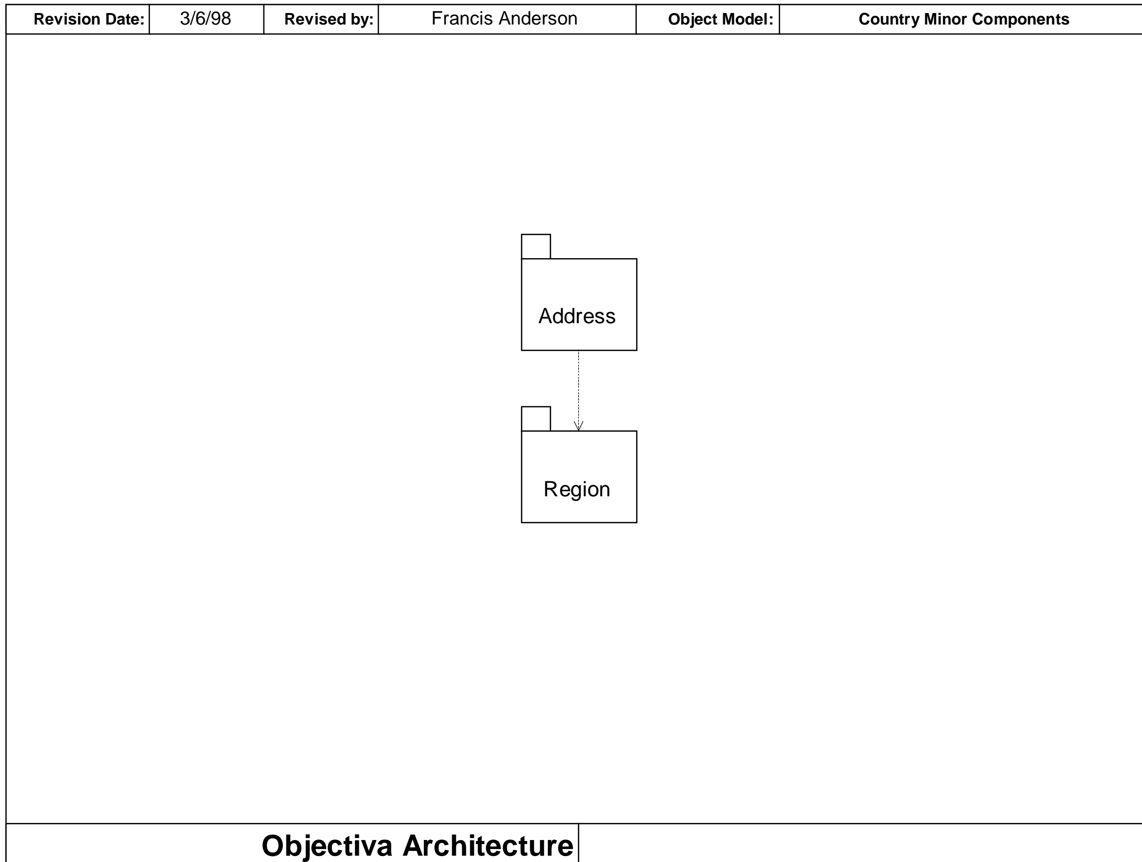

Figure 7: Country Minor Components

The Country Minor Components (Figure 7) are Region and Address, but if Country has these Minor Components, why were they not depicted in the Objectiva Architecture Entity Editor (Figure 5)? A Minor Component is implemented as an *ENVY* subapplication (e.g. REGION SUB APP) within the Major Component application (COUNTRY APP). This association relationship can be derived by naming within the Smalltalk image, and is displayable via the Relationships button on the Entity Editor.

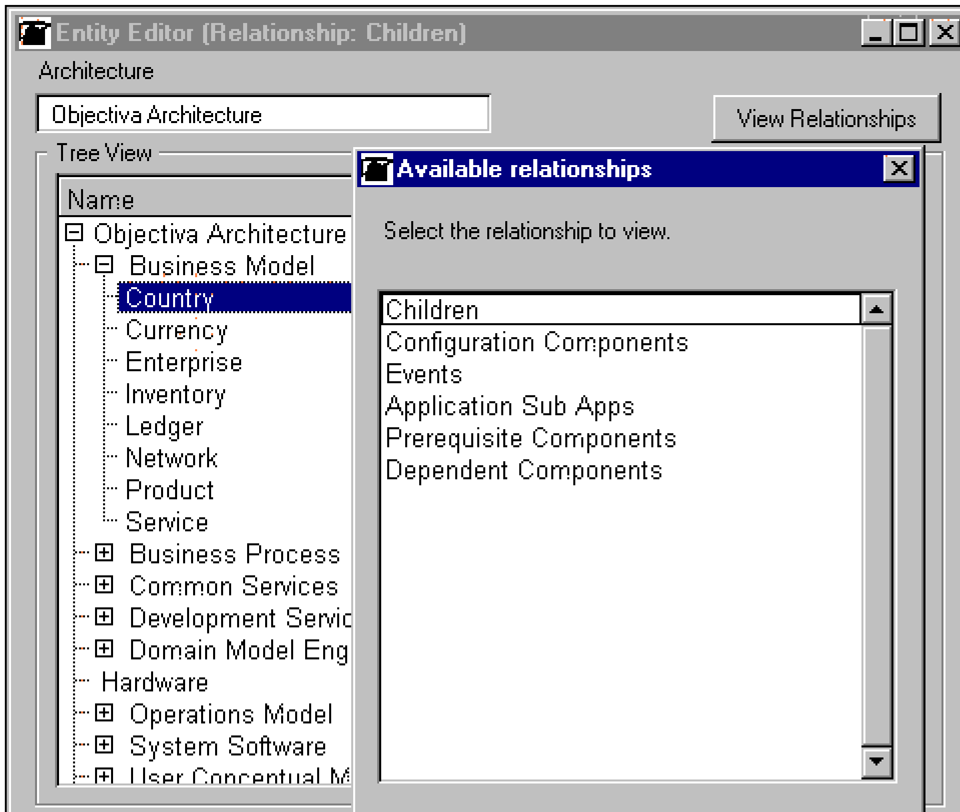

Figure 8: Major Component Relationships

Having selected the Configuration Components relationship (Figure 8), the Entity Editor is redisplayed, but instead of viewing the architectural composition, the configuration composition (physical ENVY structure) is displayed (Figure 9).

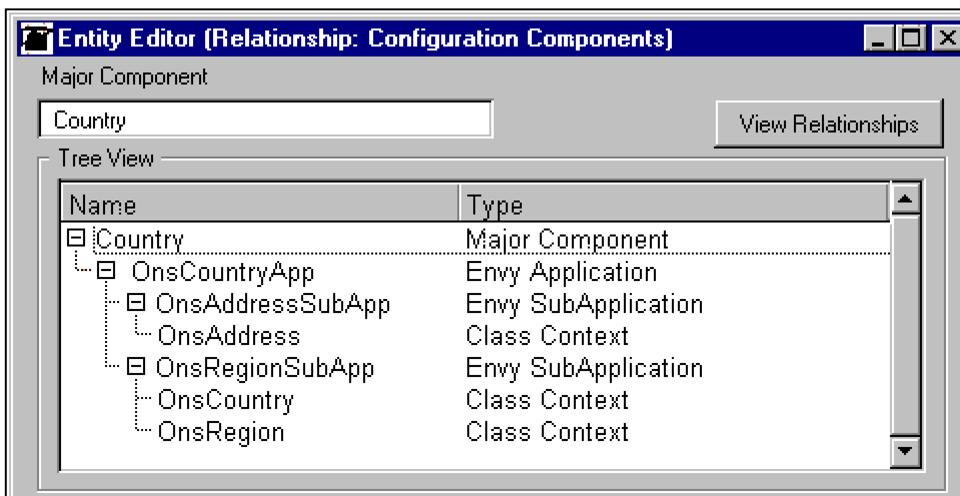

Figure 9: Country Major Component Configuration Components

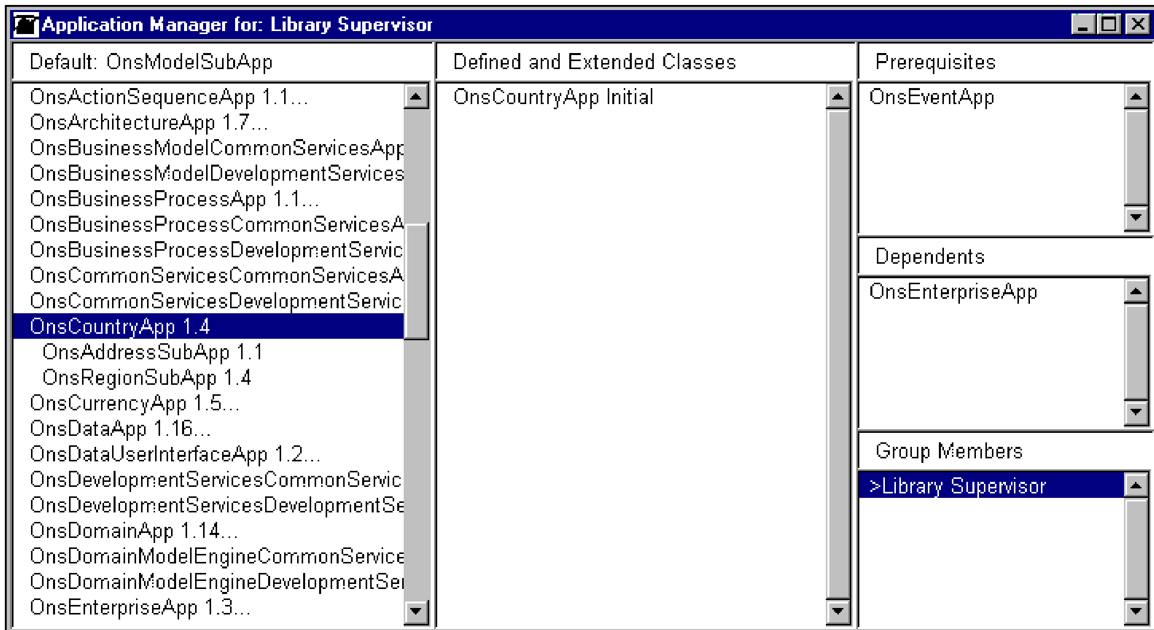

Figure 10: Country *ENVY* Application

The Country ENVY Application (Figure 10) shows the dependencies from the UML Package Diagram implemented as *ENVY* prerequisites. Since Event is a component of the DME, shows that the Business Model depends on the DME. This information is also available from the Entity Editor by selecting the Prerequisite Components (Figure 11) or Dependent Components (Figure 12) from the Major Components Relationships (Figure 8).

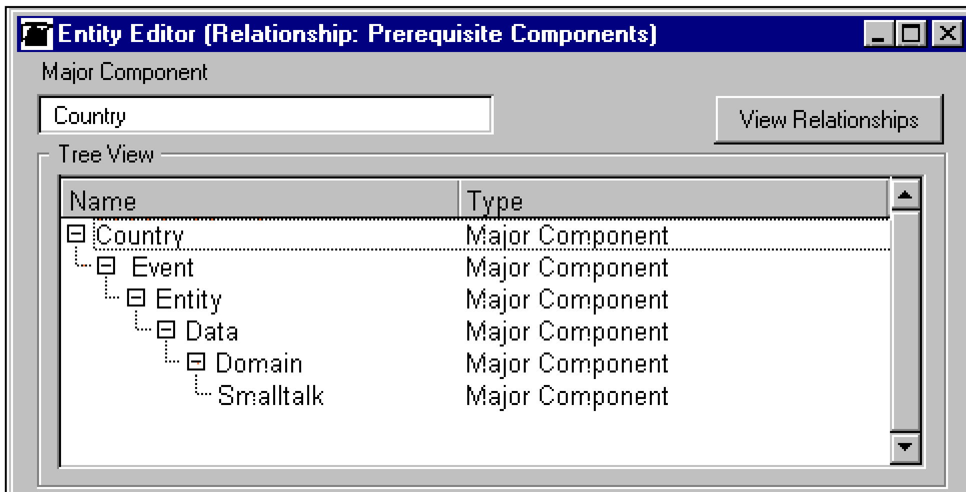

Figure 11: Country Prerequisite Components

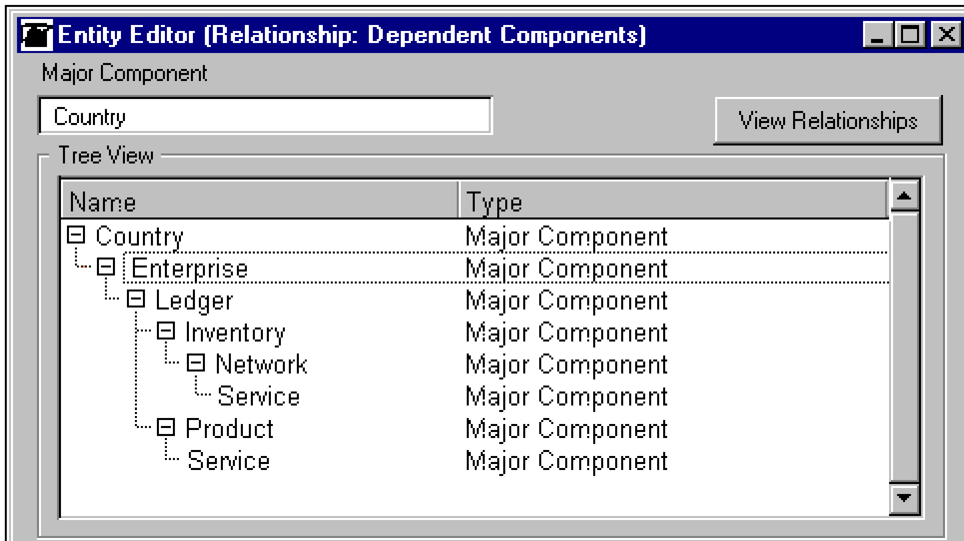

Figure 12: Country Dependent Components

In the naming of the Objectiva business objects, we have emphasized aggregation, as opposed to generalization: thus, Country, even though it is a kind of region, is chosen as the name of the business object, since it is the "big" region. When using the divide and conquer approach, one must decide where to place each component; Objectiva groups components together as much as possible because they are part of a larger component, rather than because they are the same kind of thing as a more general component. We feel that this encourages more stable and cohesive architecture, which is not dependent on the much more arbitrary design decision of inheritance.

Operational Level

To support a telecommunications billing application, Objectiva must know about a number of different kinds of region. The kinds of region that are supported depend on the Country in which Objectiva is deployed. The countries are the roots of the forest of regions. In the Operational Level, we concentrate on how the various regions (United States, Texas, Area Code '972', etc.) are represented; the Knowledge level describes how the rules governing the types of Region are expressed.

Region

The Country Minor Components (Figure 7) shows us that the primary component of the Country business object is Region. Having first applied aggregation to the naming of the business objects (Country, Currency, Enterprise, etc.), we now apply generalization to the naming of the minor components within them (Region, Address, Posting Rule, Business Party, etc.).

The first order of business within most object-oriented applications is to gain a handle on a node within a graph of objects, which can then be navigated by following pointers. This involves the execution of a query, for which Objective provides a "Finder" interface.

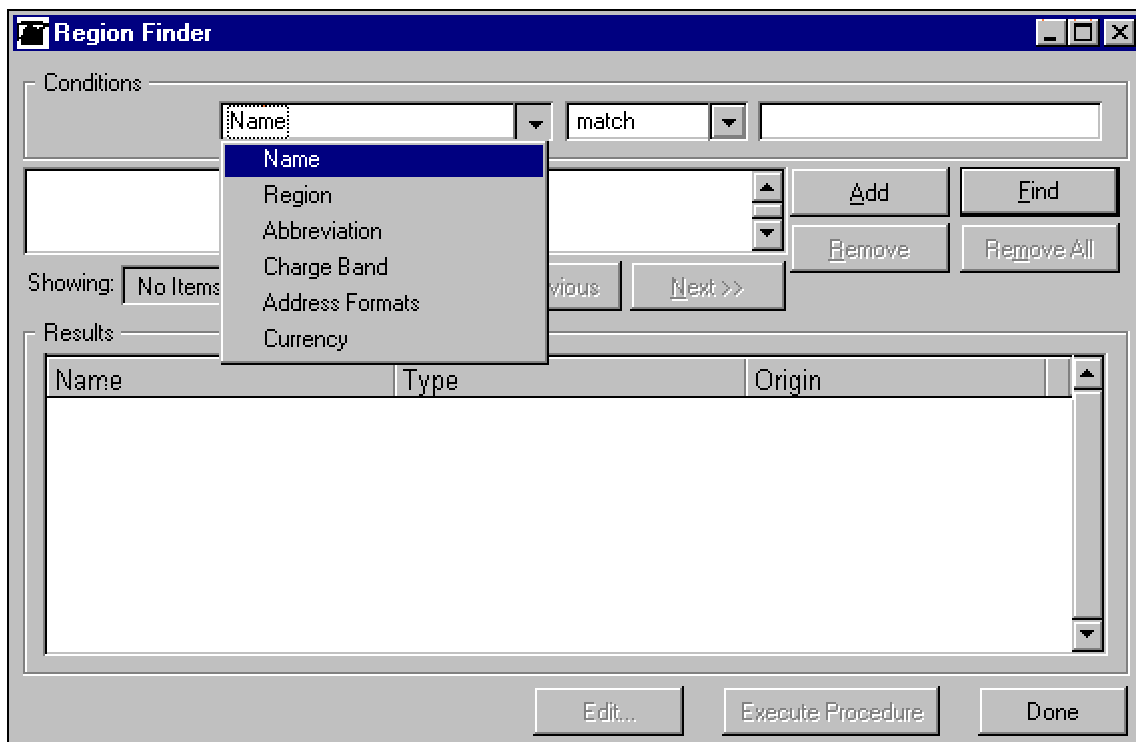

Figure 13: Region Finder

The Region Finder (Figure 13) enables us to compose a query for a Region, based on the type of region that we are looking for, or any of the attributes of a Region.

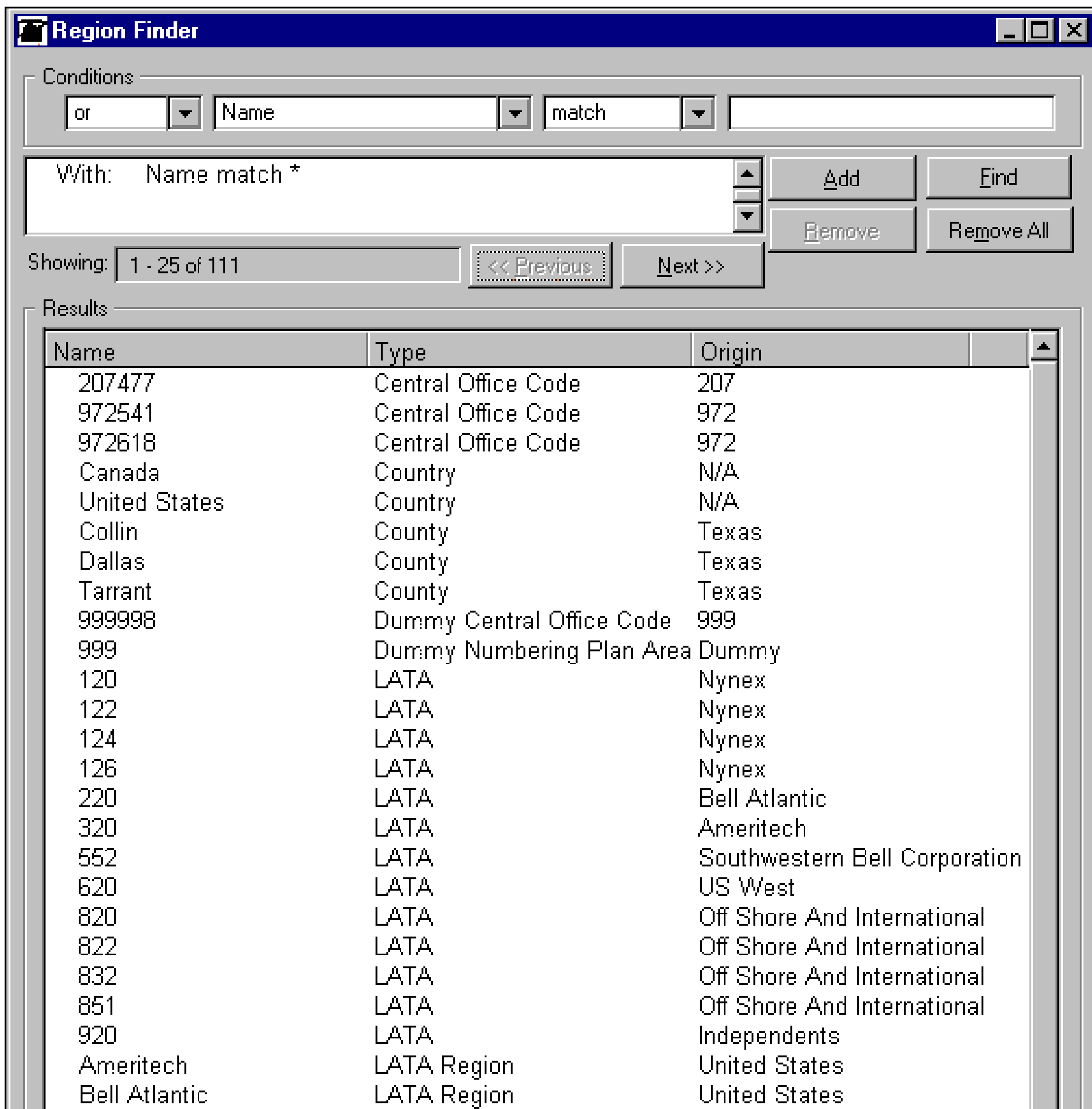

Figure 14: Results of Wild Card Search on Region Name

The Results of Wild Card Search on Region Name (Figure 14) shows the regions populated in Objectiva for testing purposes sorted by type and name. The origin displays the "parent" of the region. As countries, Canada and the United States have no parent.

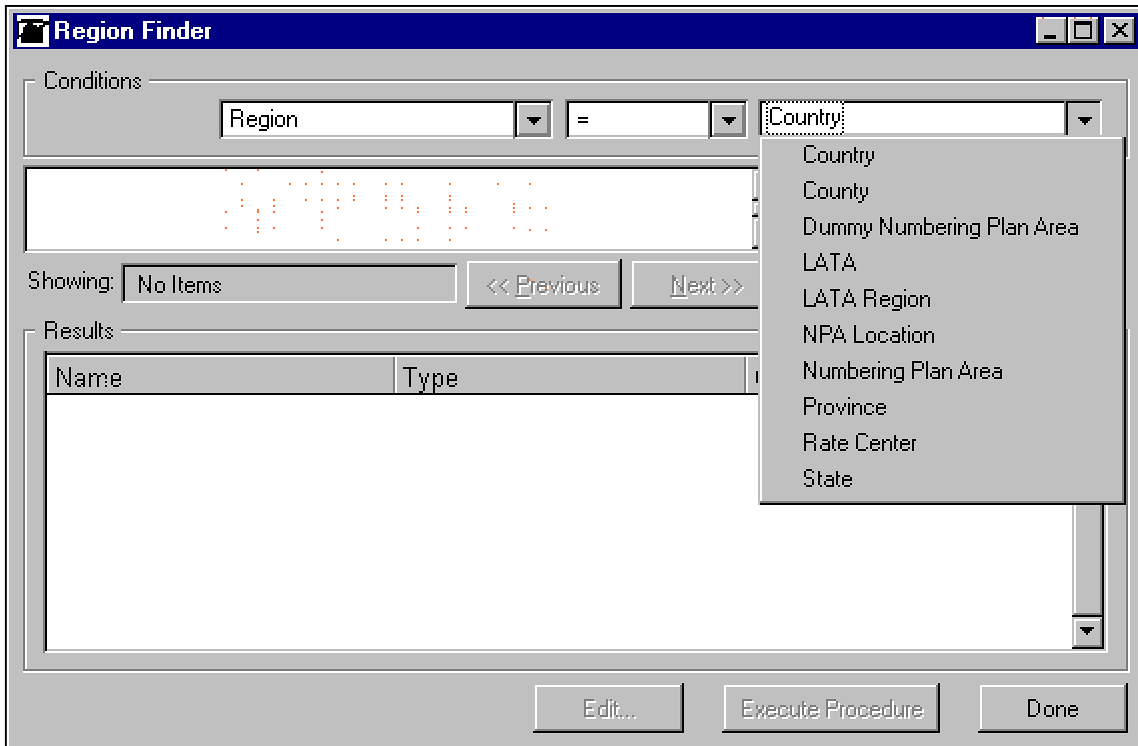

Figure 15: Region Types

In most cases when doing a search, the type of region will be known, and we will select a region to edit

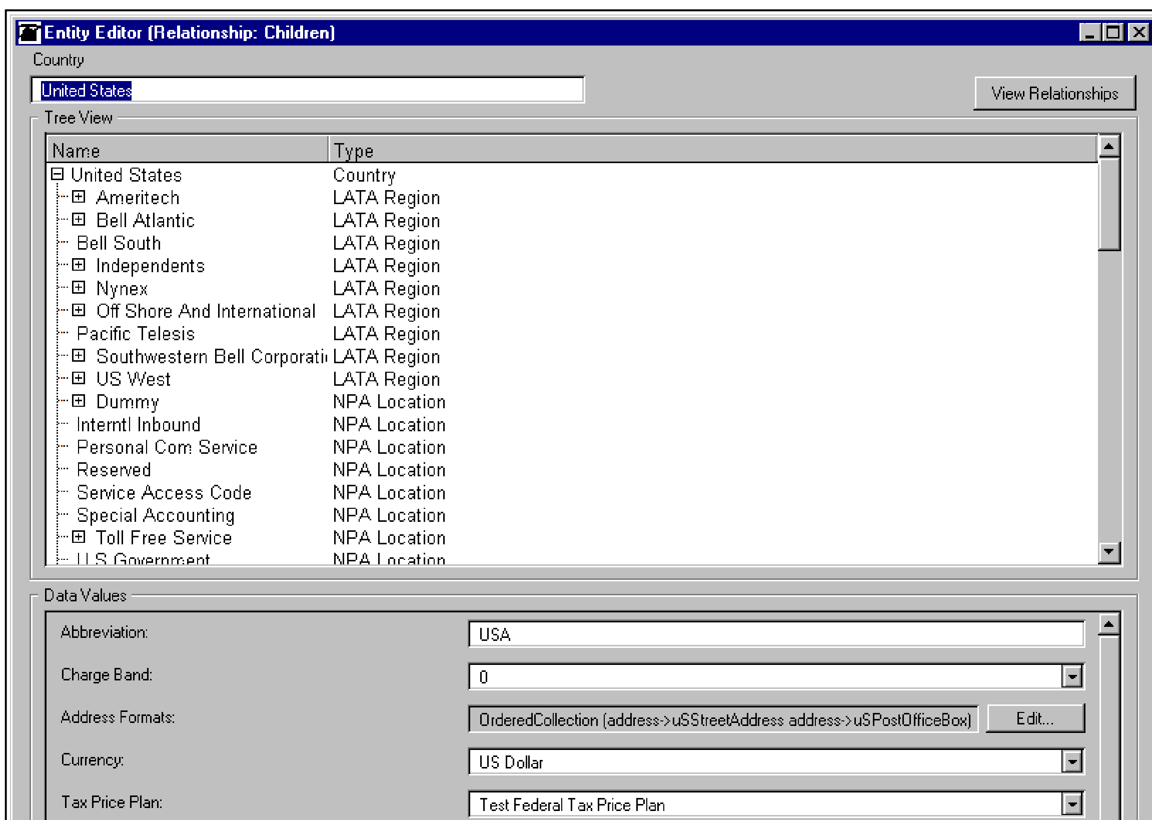

Figure 16: The United States Entity Editor

The United States Entity Editor (Figure 16) shows those attributes required by a Country, but other Region types may only require abbreviation and / or tax price plan. Note that the attributes have different data types, which we will discuss in detail later:

- abbreviation is a data entry string;
- charge band is selected from a list of available strings;
- address formats is a collection selected from a list of available objects;
- currency and tax price plan are selected from a list of available objects.

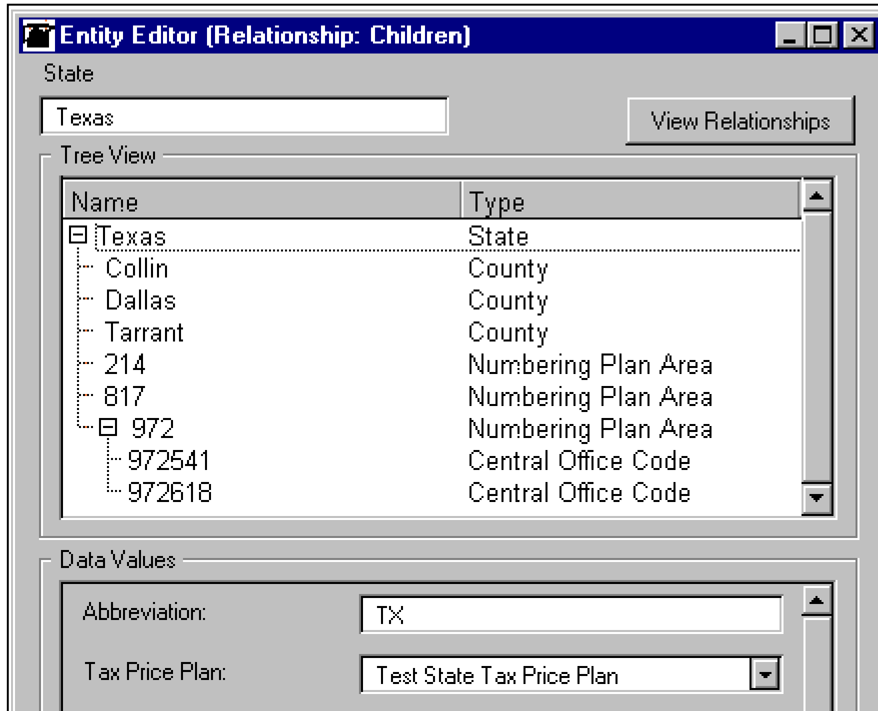

Figure 17: Texas

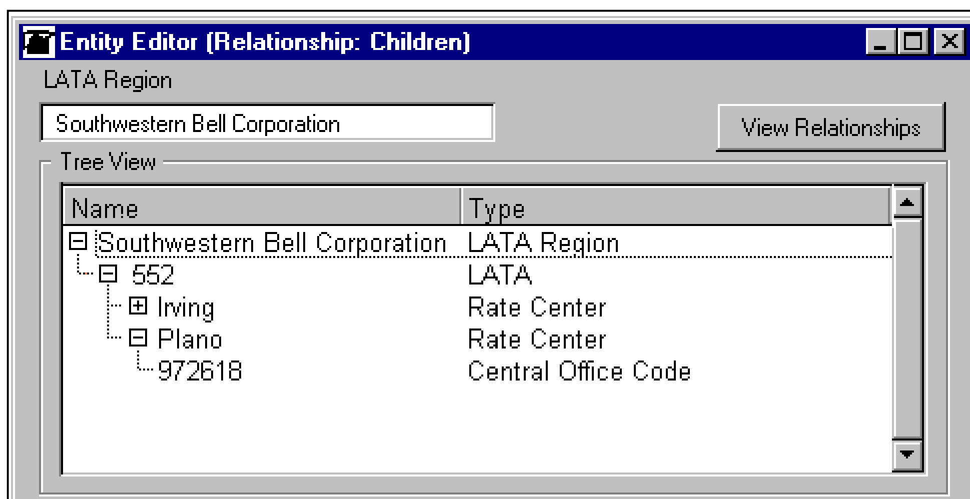

Figure 18: Southwestern Bell LATA Region

The Texas and Southwestern Bell LATA Region Entity Editors (Figures 17 and 18) show the sharing of a Central Office Code (972618) by a Numbering Plan Area (972), which is

composed of Central Office Codes, and a Rate Center (Plano), which includes Central Office Codes. Composition and inclusion are two different forms of aggregation, which we will discuss later.

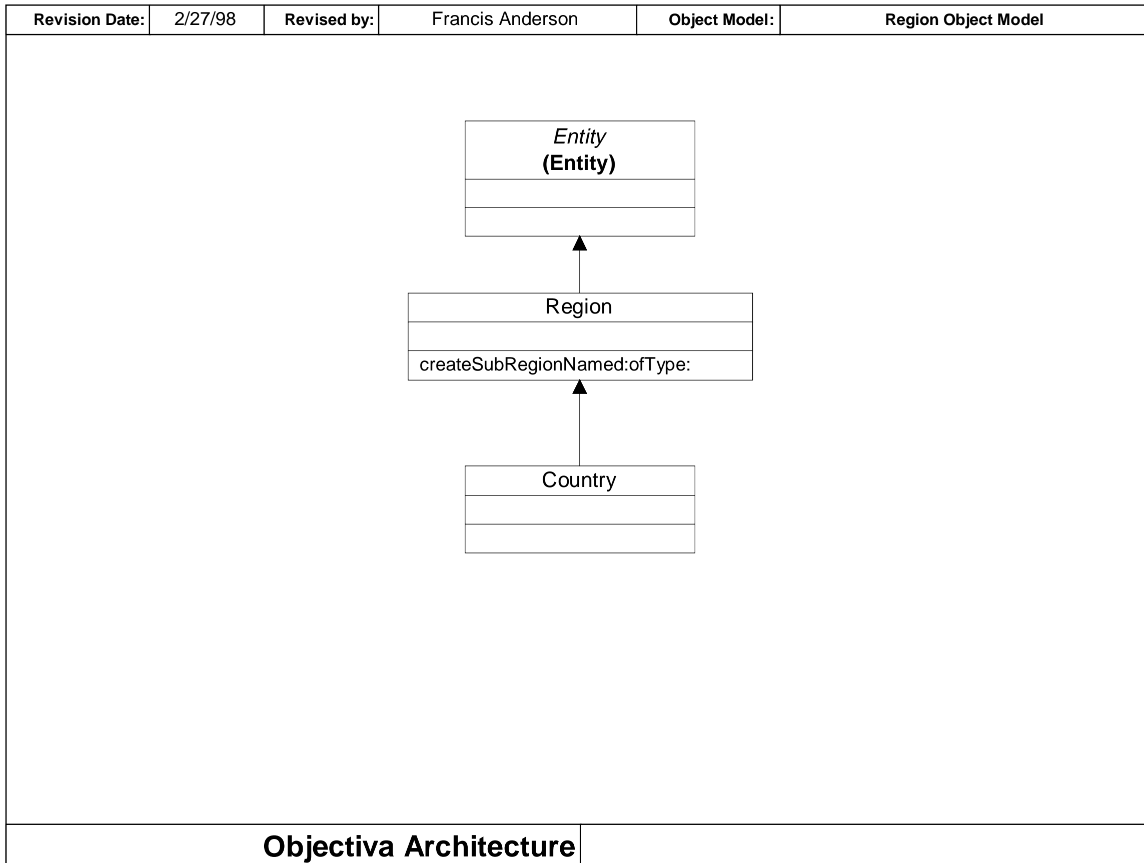

Figure 19: Region Object Model

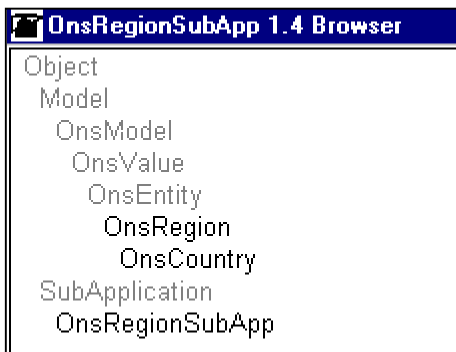

Figure 20: Region *ENVY* Subapplication

The Region Object Model (Figure 19) is implemented by the Region subapplication in *ENVY* (Figure 20). The reuse of the Entity framework allows REGION to consist of only 18 instance lines of code (LOC), and COUNTRY of only 7 class LOC. Of course, this is only in the Business Model domain, the Entity major component of the DME consists of 1,436 instance LOC, and 141 class LOC, and the ENTITY EDITOR in the User Interface domain consists of 84 instance LOC, and 74 class LOC. The test Country domain data in the Development Services domain consists of 102 class LOC to populate the knowledge

level, and 145 class LOC to populate the test data in the operational level. Each of these is built upon previously built frameworks, of course, but the point is that only 25 additional LOC are required to implement the base Region functionality.

Other applications (e.g. Dispatching) will place additional requirements on Region such as handling polygons of coordinates. These applications will add classes, structure and behavior to the Region object model, but as has been demonstrated, a significant amount of structural support is obtained by being a subclass of ENTITY. We will now look at how this is achieved.

Entity

Most of the important operational level classes in Objectiva are subclasses of ENTITY, which is part of the DME. The Region examples above start to demonstrate some of the power of the Entity framework. As we progress through the descriptions of the business objects, we will be further detailing the capabilities of the DME.

From the United States Entity Editor (Figure 16), we see that an entity has a name ('United States'). From the Tree View section of the editor, we see that an entity has a type (Country), and that it is related to other entities of various types (LATA Region, NPA Location and State). Finally, from the Data Values section of the editor, we see that an entity is described by assigning values ('USA') to attributes (Abbreviation).

Somewhat surprisingly, the only piece of information the entity holds on to for itself, is its name. It delegates the responsibility for holding on to the rest of the information to its context, and the rules governing it to its entity type.

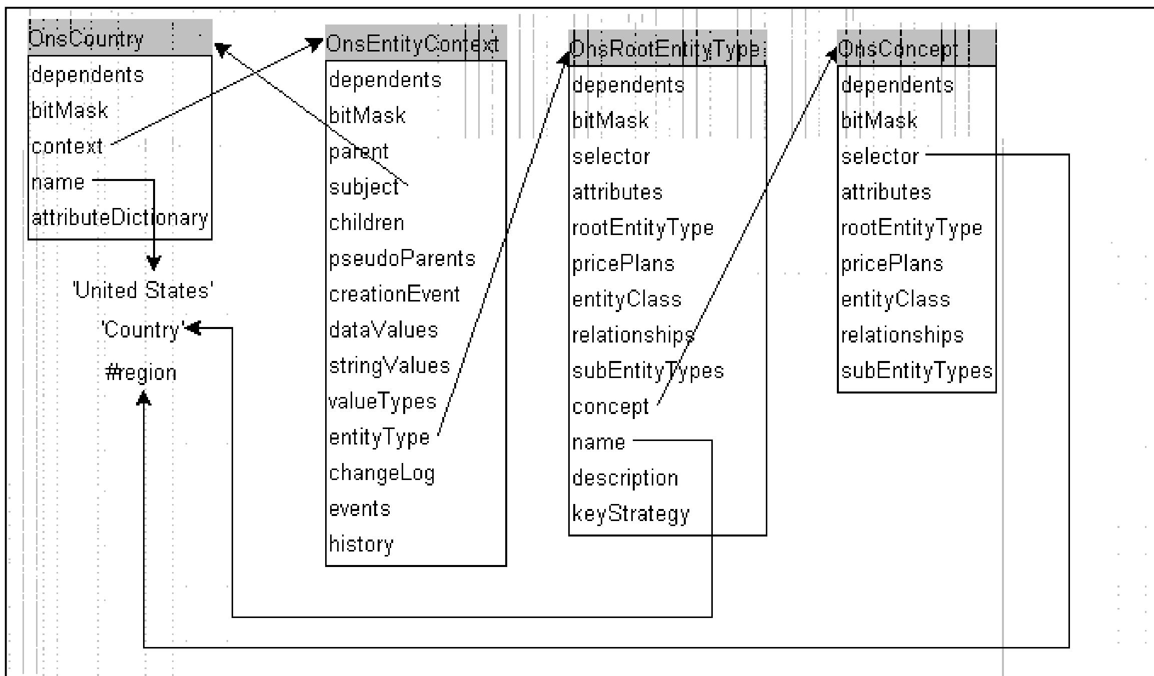

Figure 21: Instance Diagram of the 'United States'

The Instance Diagram of the United States (Figure 21) shows most of the structure is in ENTITY CONTEXT and ROOT ENTITY TYPE, with little falling into ENTITY itself. Also, note the nested application of the TypeObject pattern between ROOT ENTITY TYPE and CONCEPT. 'United States' (ENTITY) is an instance of type Country (ENTITY TYPE); Country (ENTITY TYPE) is a subtype of Region (CONCEPT). We will cover this later when describing the Knowledge Level.

The attributeDictionary variable is a temporary measure to enable an image-based query mechanism, particularly for architecture metrics, and it raises the design question of when to use an instance variable in ENTITY or a data value in ENTITY CONTEXT. As a

general rule, association relationships and queryable continuous data values (e.g. name) should be stored in instance variables.

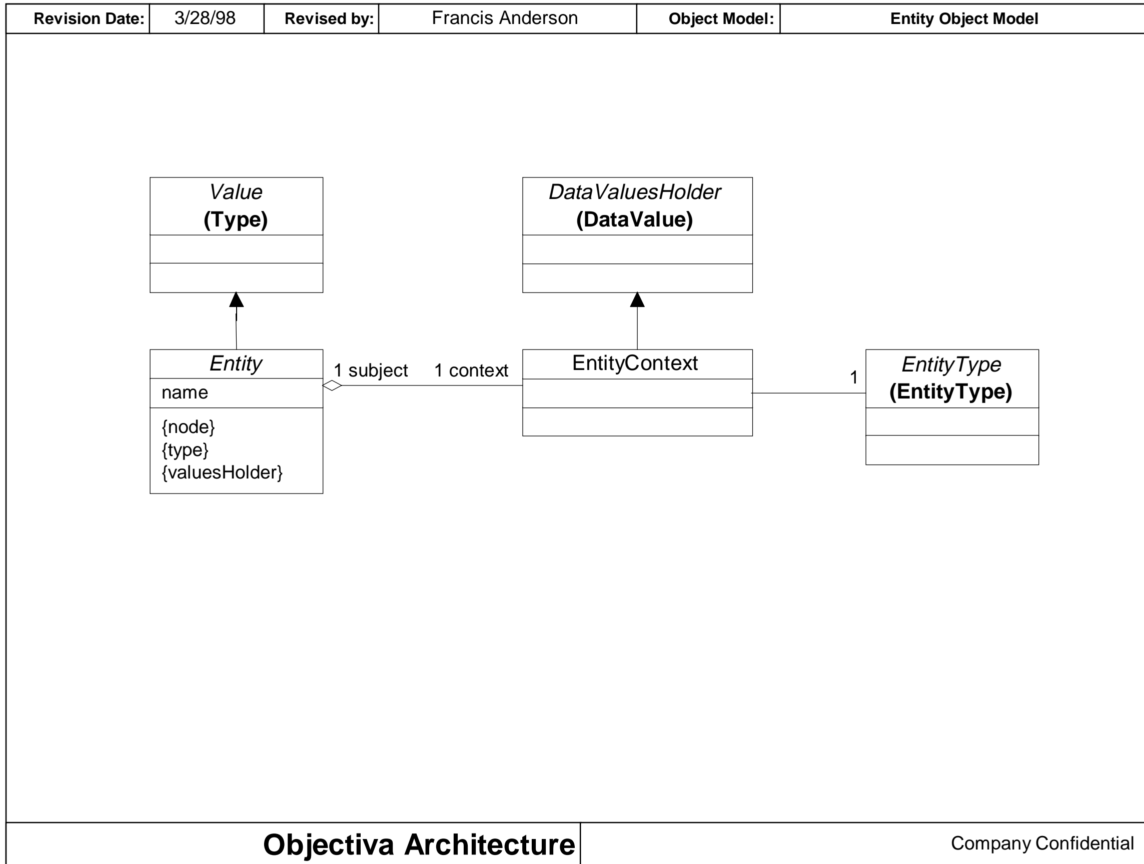

Figure 22: Entity Object Model

From the Entity Object Model (Figure 22), we see that ENTITY and ENTITY TYPE play the expected roles in the Type Object pattern. In this case, the classification relationship has been reified as ENTITY CONTEXT, which is a subclass of DATA VALUES HOLDER. An entity's context holds on to its type, its data values, and its parent / child (aggregation) relationships. ENTITY CONTEXT also tracks the changes to these values over time, through three different historical mechanisms: a change log, an event collection, and a collection of historical editions of itself. We will be describing these historical capabilities later, but they are mentioned now because it is the context's responsibility for tracking the history of its subject entity that determines its implementation via delegation rather than inheritance.

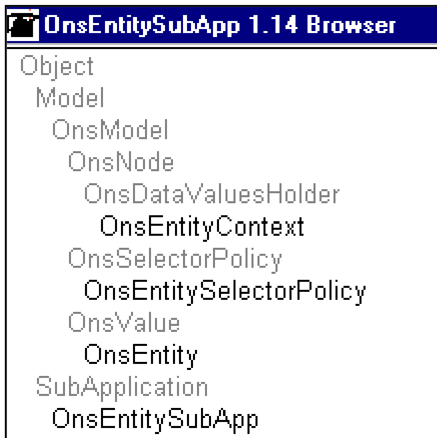

Figure 23: Entity ENVY Subapplication

The Entity Object Model (Figure 22) shows that the only variable added by ENTITY CONTEXT is entityType; actually, it also adds changeLog, events, and history, but we will cover these later. So, let us look at how ENTITY CONTEXT stores its dataValues, which it inherits from DATA VALUES HOLDER in the Data Value minor component.

Data Value

Instead of representing the properties of an ENTITY as instance variables, an ENTITY CONTEXT holds them in a collection named dataValues. This is a very flexible solution, in that a property may be added to an ENTITY with no change required to either code or the physical schema.

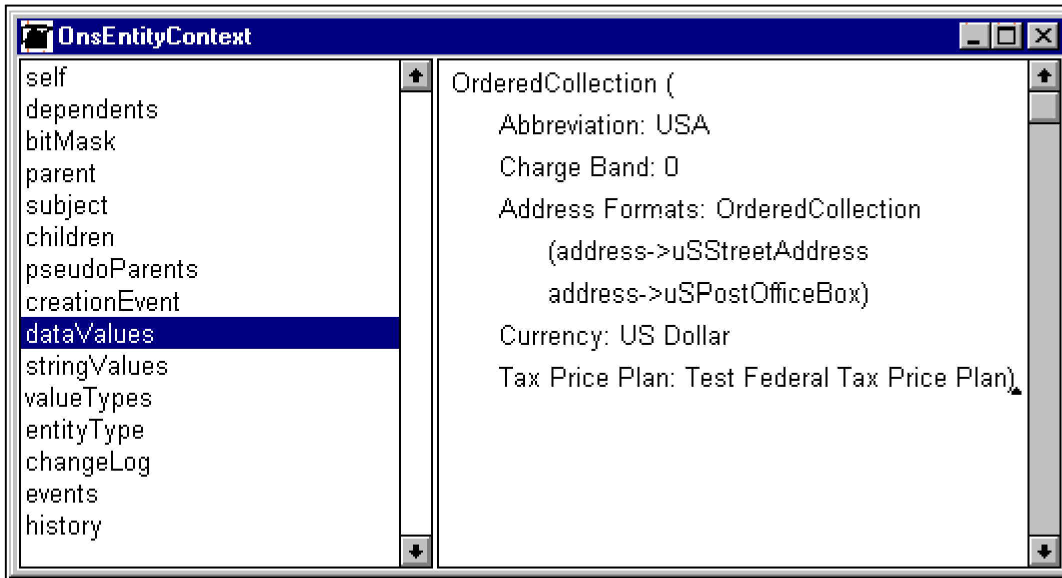

Figure 24: Inspector View of United States Data Values

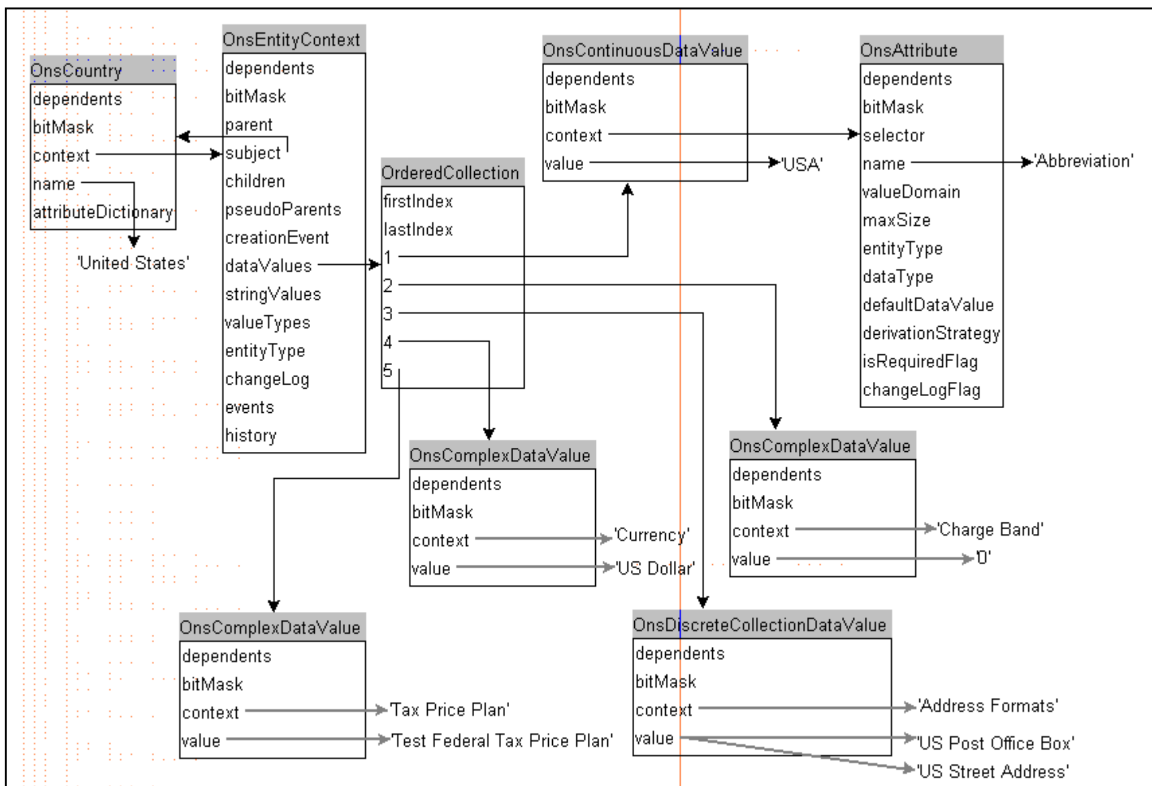

Figure 25: Instance Diagram of Data Values

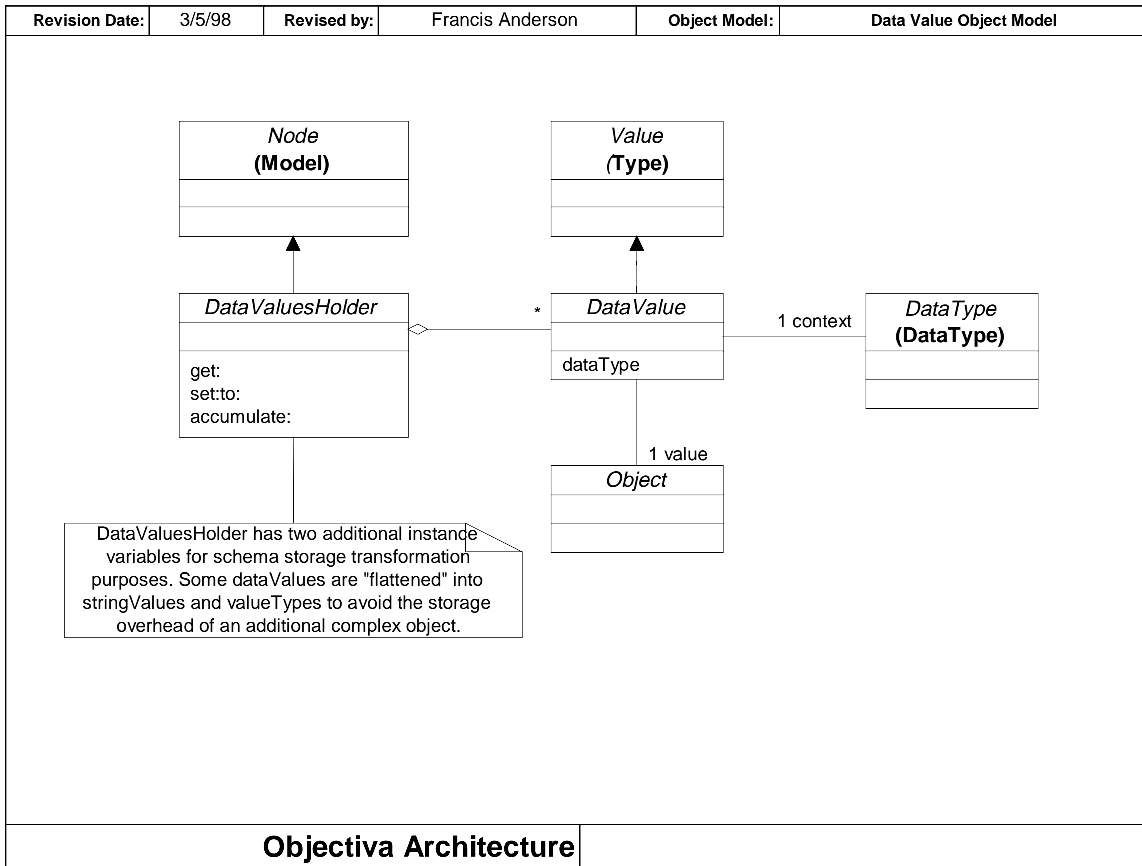

Figure 26: Data Value Object Model

The Data Value Object Model (Figure 26) shows how DATA VALUES HOLDER has an aggregation of many DATA VALUE, which places a value OBJECT in the context of a DATA TYPE. All these classes are abstract. This, of course, is a second example of the Type (DATA TYPE) Object (DATA VALUE) pattern, and Fowler's Operational and Knowledge levels. For now, we will concentrate on the operational level DATA VALUES HOLDER and DATA VALUE. We will discuss the relationship to DATA TYPE later.

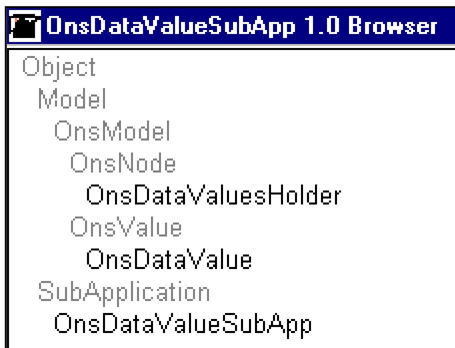

Figure 27: Data Value *ENVY* Subapplication

In the case of a CONTINUOUS DATA VALUE (United States Abbreviation 'USA'), we are dealing with a String value ('USA') in the context of an ATTRIBUTE (Abbreviation) of an ENTITY TYPE (Country). As a general rule, we do not wish to store continuous values as

persistent objects, since they cannot be shared, and, as simple values, only have meaning within the context of one DATA VALUES HOLDER. Objectiva provides the schema Major Component in the Common Services domain, which performs storage transformation on objects prior to making them persistent. Prior to invoking storage transformation, objects are told to "flatten" themselves.

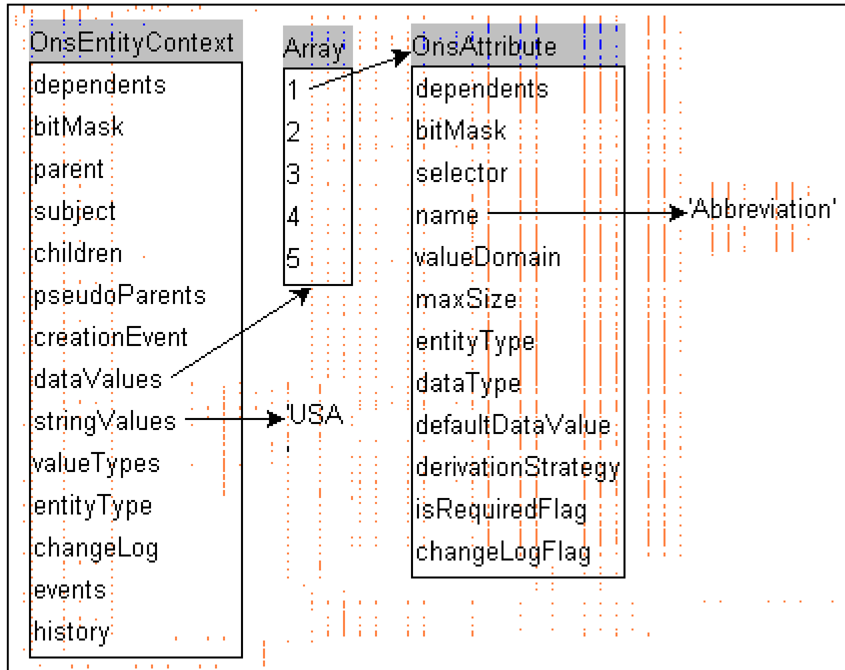

Figure 28: Instance Diagram of Flattened Continuous Data Value

The Instance Diagram of the Flattened Continuous Data Value (Figure 28) shows how an entity context flattens its continuous data values by storing the value in the stringValues collection, and replacing the continuous data value with the attribute in the dataValues collection. The dataValues OrderedCollection is replaced by an Array and the stringValues StringCollection is replaced by a carriage return delimited String.

In this section, we have described how ENTITY uses ENTITY CONTEXT to store the values of simple DATA TYPES (ATTRIBUTES), a capability ENTITY CONTEXT inherits from DATA VALUES HOLDER. Next we will describe how relationships to other ENTITYs are also stored in ENTITY CONTEXT, a capability it inherits from NODE.

Node

When we look at the United States Entity Editor (Figure 16), we see that the screen is divided into two halves: "Tree View" and "Data Values". The data values describe the entity; the tree views define its relationship to other entities. The United States consists of states (e.g. Texas), which consist of counties (e.g. Collin, Dallas, and Tarrant) and Numbering Plan Areas (e.g. '972'), which consist of Central Office Codes (e.g. '972628'). Texas (Figure 29) becomes a node in a tree, or graph, of regions.

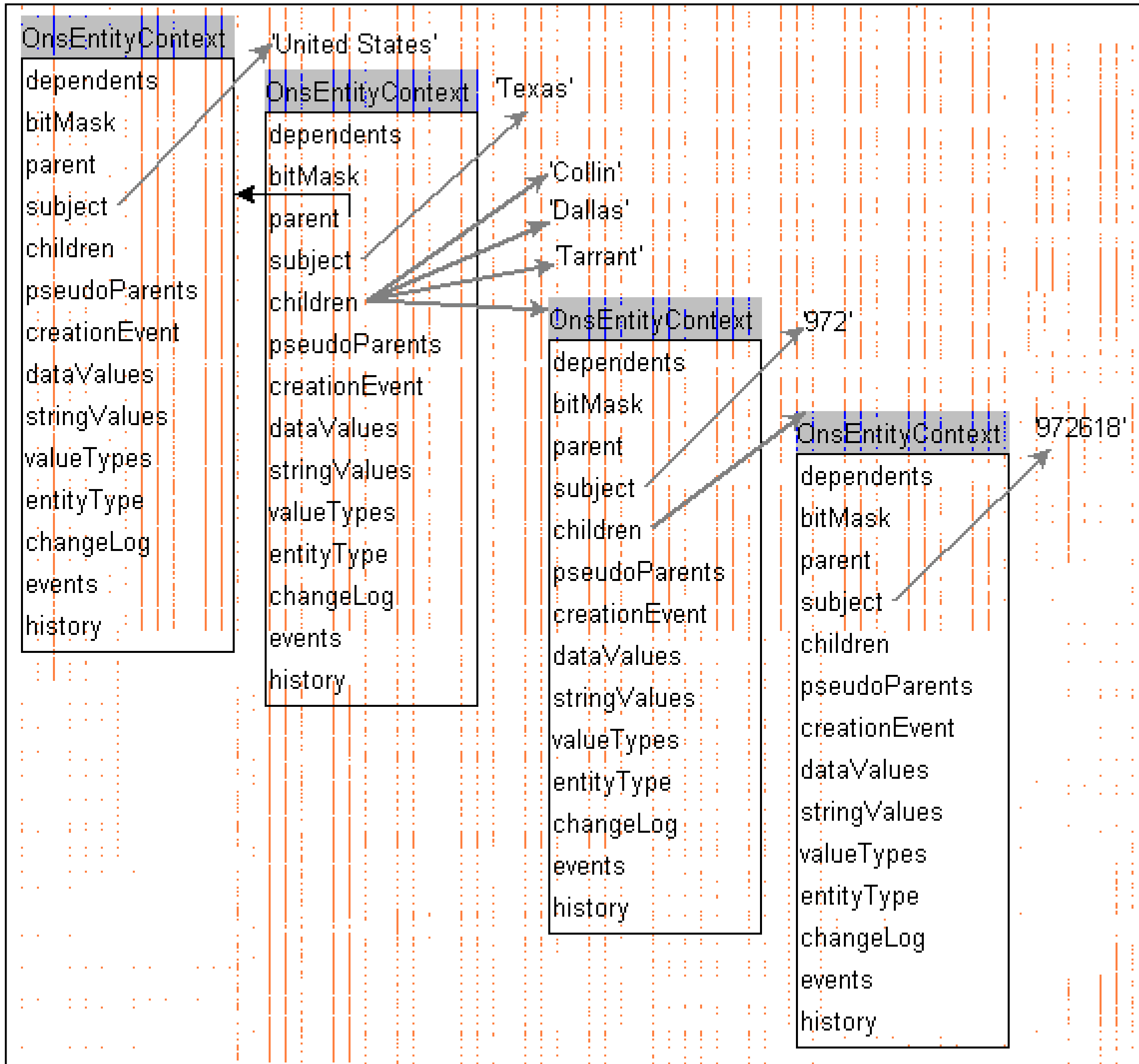

Figure 29: Instance Diagram of Region Contexts

DATA VALUES HOLDER inherits this capability from NODE, which provides a generalized implementation for the values of relationships, similar to the manner in which data value provides a generalized implementation of the values of attributes. Node is an extension of the Composite pattern [Gamma 95], except that a black-box framework approach is taken, implemented by delegation, rather than a white-box approach, implemented by inheritance.

A graph of nodes may be a tree or a directed acyclic graph (DAG), depending on the number of parents that a child is allowed. It is very hard to think of an entity that is not part of at least one tree. In Objectiva, an entity delegates the responsibility for keeping track of its position in multiple trees to its context.

A node keeps track of its links to other nodes in a graph with the instance variables children, parent, and pseudoParents:

- children may be empty, in which case the entity is a leaf; or children may have members, in which case the entity is a composite. The United States Entity Editor (Figures 13 and 14) shows some of the children of the United States (Alabama, Arkansas, etc.).
- parent may be nil, in which case the entity is the root of a tree and provides overall context; for example, countries are the root regions, so the parent of the United States is nil.
- pseudoParents is used to represent a DAG structure, in addition to a tree structure. This handles the situation depicted in Figures 11 and 12, where the central office code (COC) ‘972618’ is part of both the numbering plan area (NPA) ‘972’ and the rate center ‘Plano’.

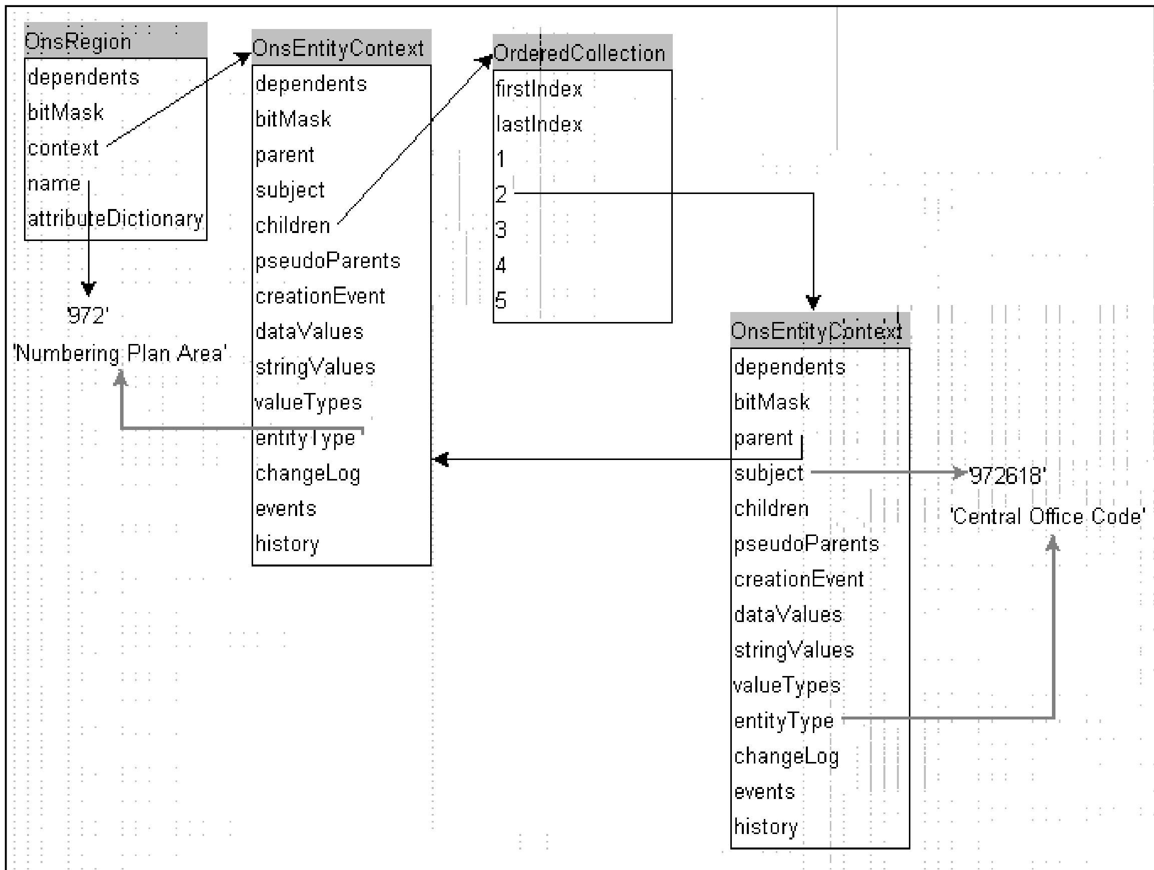

Figure 30: Instance Diagram of a Composition Relationship

The Instance Diagram of a Composition Relationship (Figure 30) shows the implementation of the relationship between NPA ‘972’ and COC ‘972618’. This

relationship is a composition, and a strong form of composition at that, since ‘972’ is propagated from the NPA into the name of the COC. Only an NPA can create a COC; if an NPA is deleted, all its COC children must be deleted too; a COC can only be part of one NPA.

Thus, since a child may only be part of one composition, the back pointer from the COC context to the NPA context is stored in the parent variable.

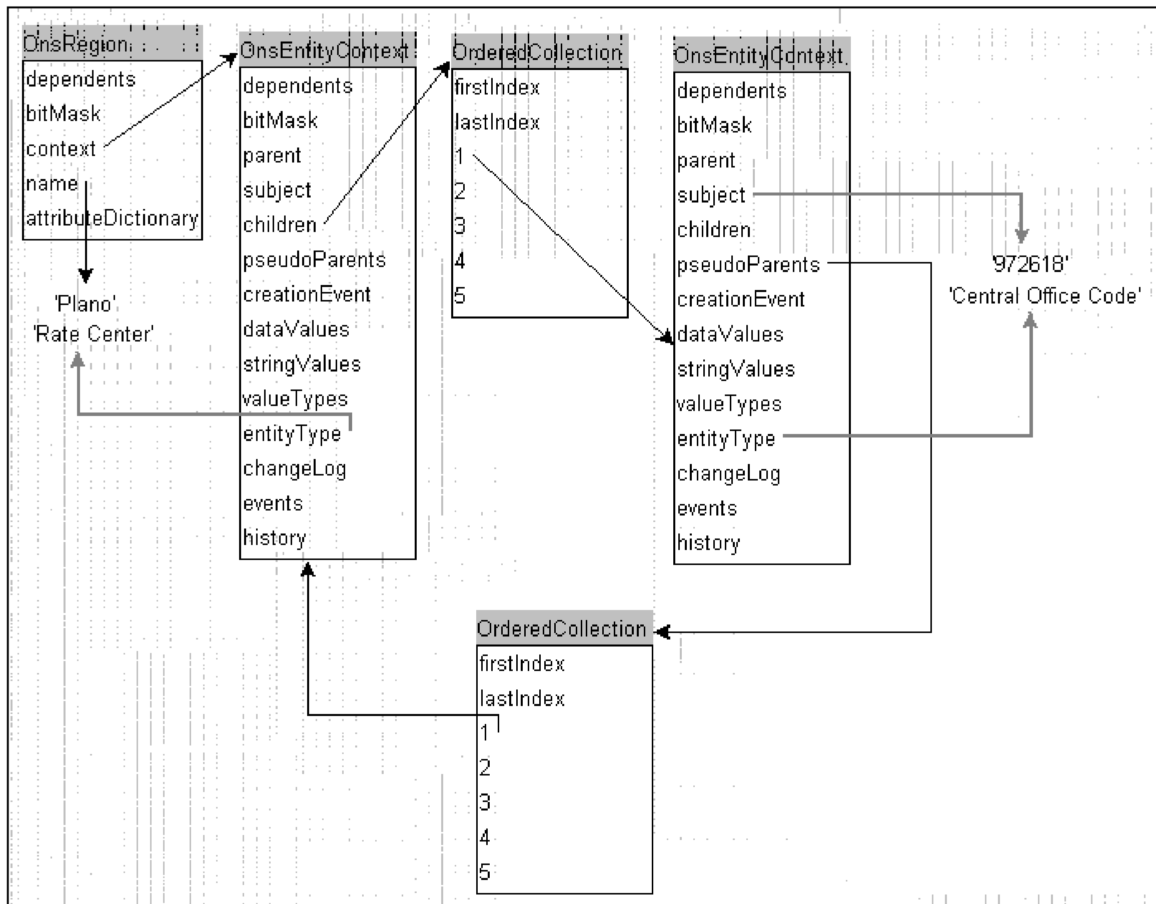

Figure 31: Instance Diagram of an Inclusion Relationship

The Instance Diagram of an Inclusion Relationship (Figure 31) shows the implementation of the relationship between Rate Center ‘Plano’ and COC ‘972618’, which is a weaker form of aggregation that Objectiva calls inclusion. Before becoming part of a rate center, a COC must already have been created by an NPA, and the back pointer from the COC to the rate center is stored as a member in the pseudoParents variable, thus allowing an entity to be a member of multiple inclusions.

NODE is defined in the root component of the Objectiva Architecture – the Model minor component of the Domain major component of the Domain Model Engine domain. We will not be discussing PROPOSITION and ERROR yet.

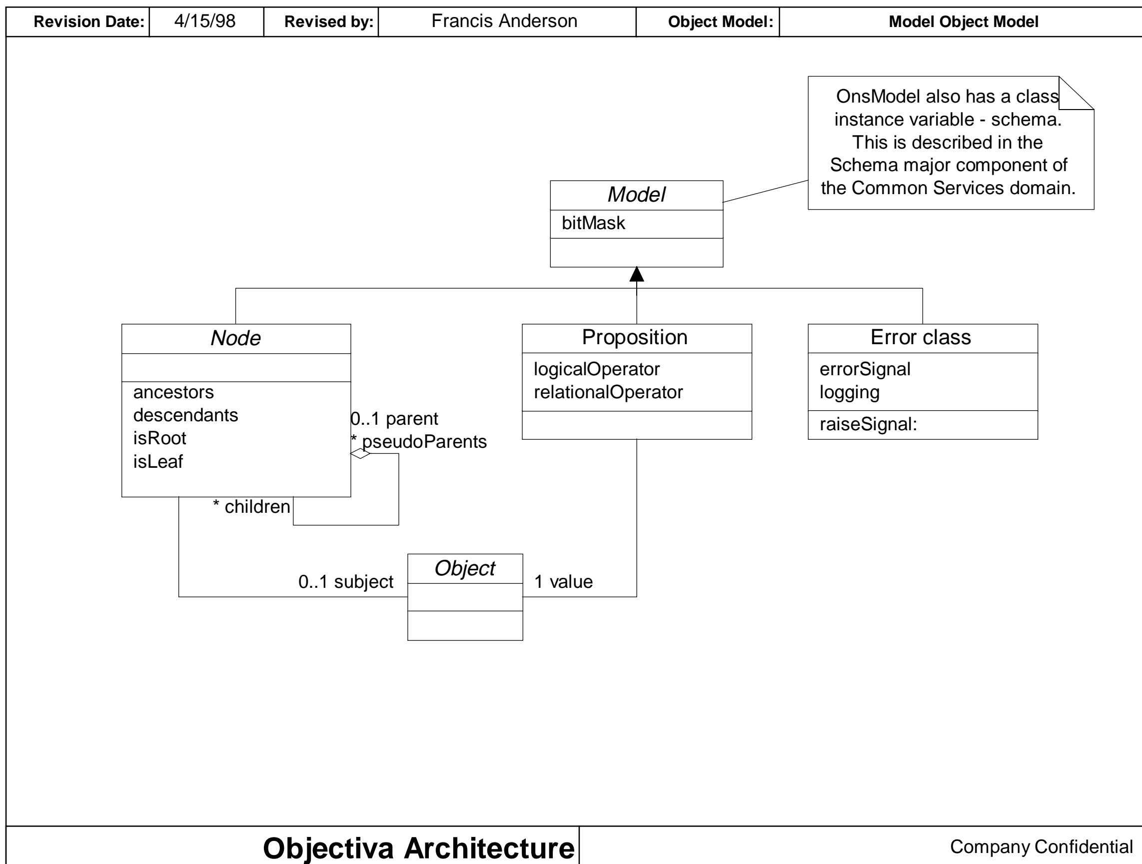

Figure 32: Model Object Model

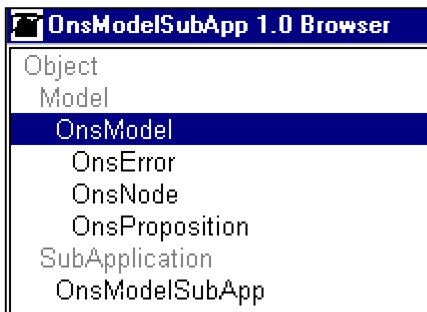

Figure 33: Model *ENVY* Subapplication

Entity Context

Having now reached the root of the operational level Entity hierarchy, let us summarize before describing the knowledge level that provides the rules mechanism that governs the operational level.

An ENTITY has a context. The ENTITY CONTEXT has the following responsibilities:

- As a NODE, it is responsible for storing the relationship values of its subject entity (COC '972618'). These are expressed in terms of its parent (NPA '972'), its pseudoParents (rate center 'Plano') and its children (empty) – see Figures 30 and 31.

- As a DATA VALUES HOLDER, an ENTITY CONTEXT is responsible for storing the attribute values of its subject entity (e.g. country 'United States'). A data value is a complex object that stores a value in the context of a data type. A continuous data value is only owned by one entity context ("has by value"), and we do not want to make a complex object persistent when a simple value is being represented. On being made persistent, a continuous data value is flattened into its string representation, and stored in the stringValues variable. If a measurement has been taken, the unit of the quantity is stored in the valueTypes variable.

- The rules governing the relationship and attribute values of the ENTITY CONTEXT are obtained from its entityType, which we will discuss in the Knowledge Level section.

- Finally, ENTITY CONTEXT keeps track of its change of state over time via its changeLog, events and history variables. We will discuss these capabilities of ENTITY CONTEXT when we demonstrate how the Event major component of the DME supports the Currency and Ledger business objects.

Knowledge Level

In the Operational Level above, we concentrated on instances of Region (United States, Texas, Area Code '972, etc.). In the Knowledge Level, we define the rules governing the types of Region (Country, State, Area Code, etc.). Particularly, we describe the relationships governing the types of Region (e.g. a country may have states), and the attributes that describe the types of Region (e.g. we want to capture a country's currency).

Entity Type

In the United States, the Telecom industry depends on the North American Numbering Plan (NANP), previously administered by BellCORE, now administered by Lockheed Martin. Telecom providers receive updates to the NANP via the Local Exchange Routing Guide (LERG), which is a set of flat files that describe the regions that make up the NANP, and the carriers that are responsible for local service in those regions.

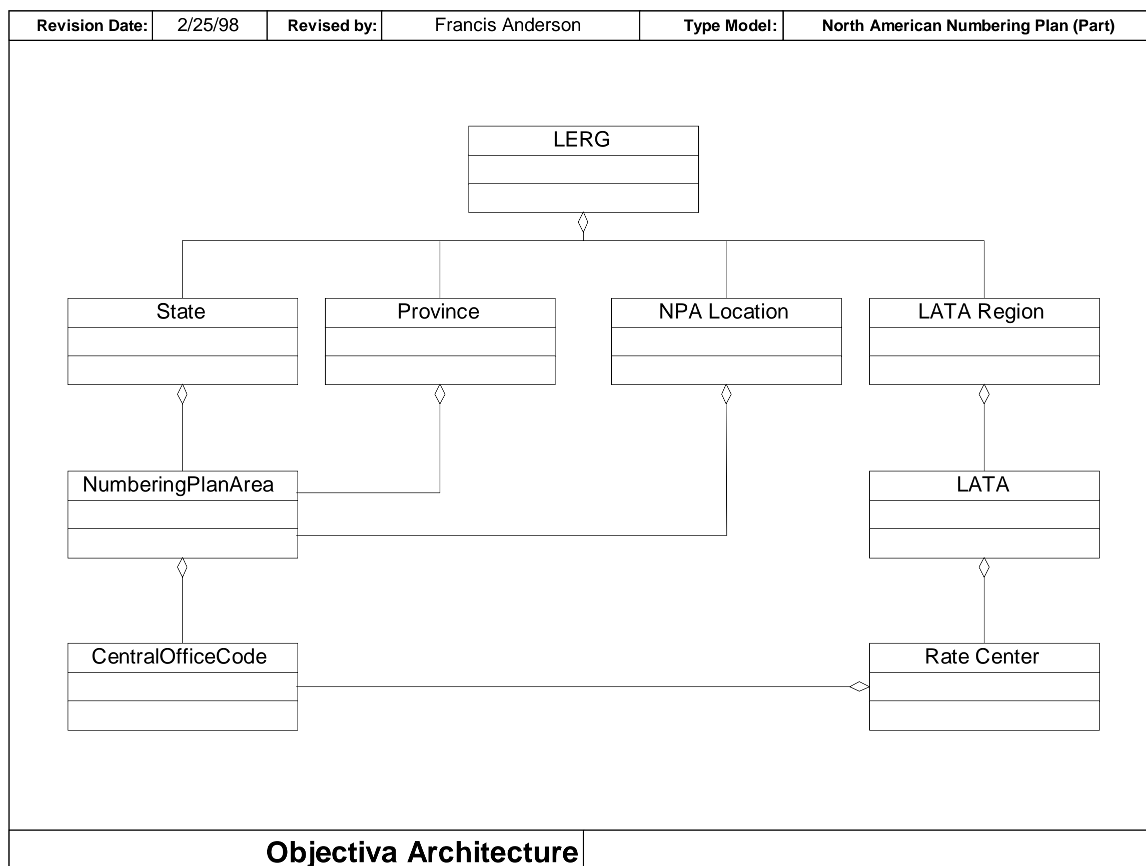

Figure 34: Partial Structure of the LERG.

The Partial Structure of the LERG (Figure 34) shows some of the types of Region contained in the LERG, and some of the relationships between them. A Numbering Plan Area (NPA) falls within either a State, Province, or NPA Location. An NPA is composed of Central Office Codes, which are included in Rate Centers, which are part of

Local Access Transport Areas (LATA). Note that this diagram will soon be invalidated with the full implementation of Number Portability.

Object-oriented systems often support variability with inheritance. For example, one way to describe how these types of regions vary would be to them subclasses of REGION. However, this would lead to new subclasses for every application, so Objectiva uses a different technique. The core of this technique is provided by the Entity and Data components of the DME. Entity and Data both use the Type Object [JohnsonWoolf97] pattern to define new types of business objects. Entity Types are described in terms of Attributes using the Property [Foote97] pattern. Attributes are also an example of the Observation [Fowler97] pattern. This dense combination of patterns at the core of Objectiva is part of what makes it so powerful, but is also part of what makes it hard to learn.

It should also be noted that these requirements are purely for a billing application. A trouble call dispatching application would require mapping capabilities, which would be additional responsibilities of Region, but are not described in this document.

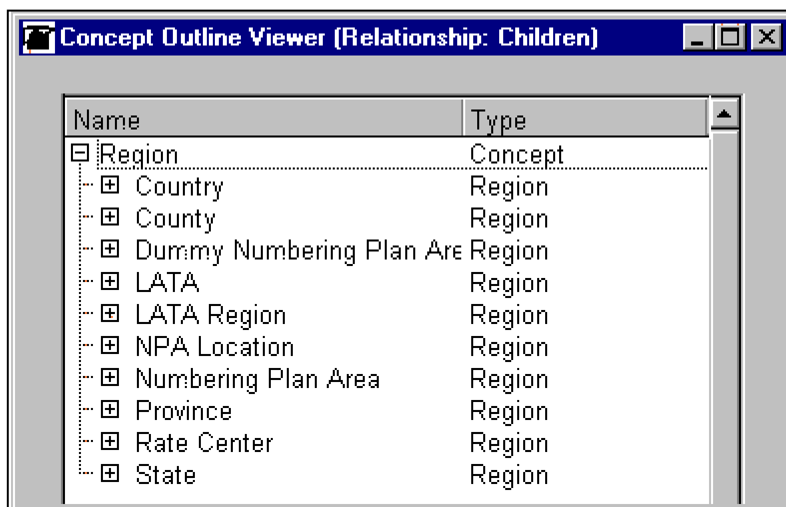

Figure 35: Region Entity Types

The Region Entity Types (Figure 35) shows examples of the types of Region that may be implemented by Objectiva. This shows the Nested Type Object pattern, since "Region" is an instance of CONCEPT. So, the United States (an ENTITY) is a Country (an ENTITY TYPE), which is a Region (a CONCEPT).

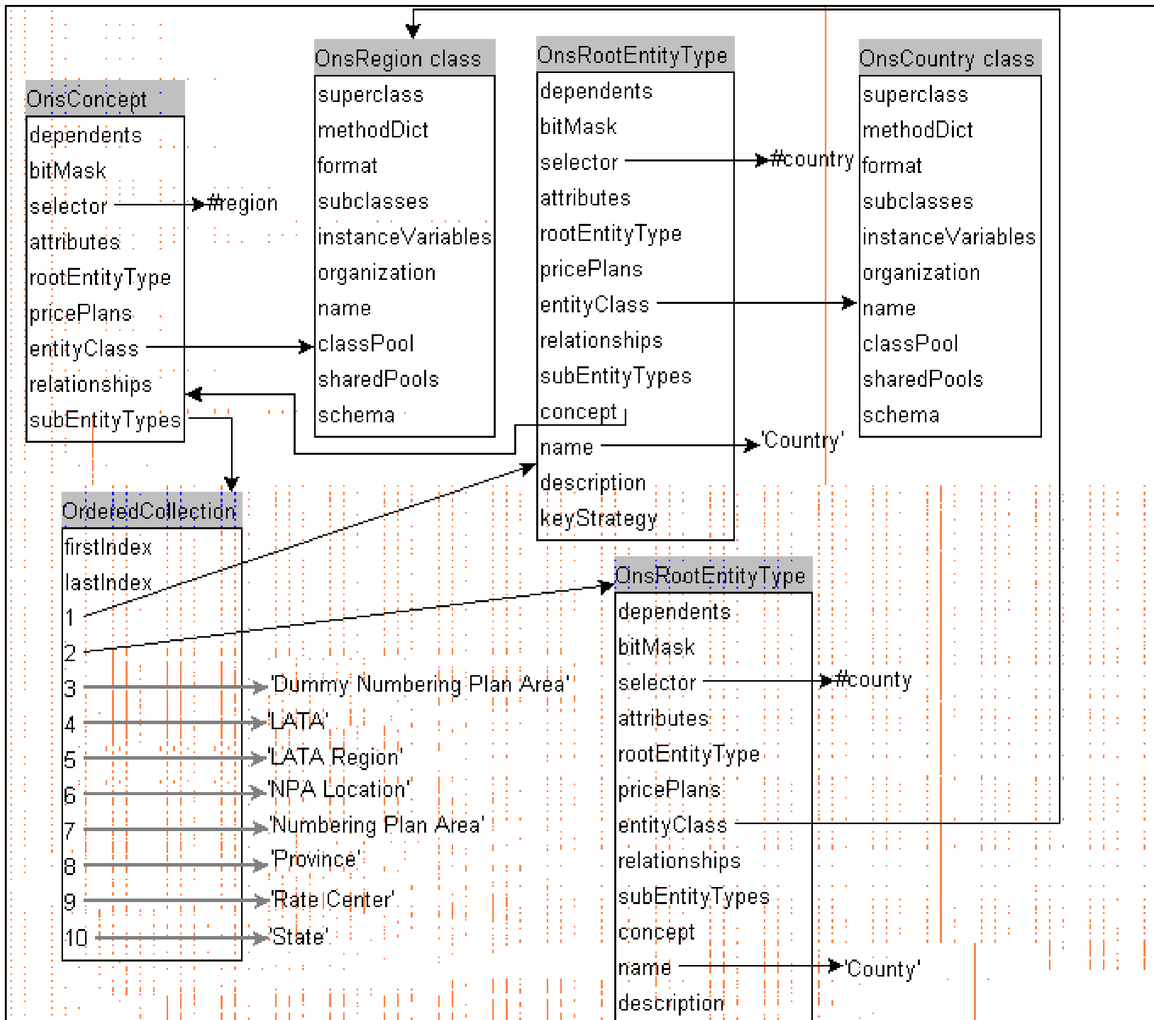

Figure 36: Instance Diagram of the Region Concept

The Instance Diagram of the Region Concept (Figure 36) demonstrates the Power Type pattern [Odell95], in which an instance of CONCEPT (Region) corresponds to a subclass of ENTITY (REGION). A ROOT ENTITY TYPE (Country) within a CONCEPT may override the default entityClass.

We create new entities by sending the message #createEntityNamed: to a Root Entity Type, rather than sending #new directly to a class. ENTITY TYPE is thus a factory for ENTITY.

```
ENTITY TYPE>>createEntityNamed: aString

        ^self entityClass newNamed: aString
                withType: self
```

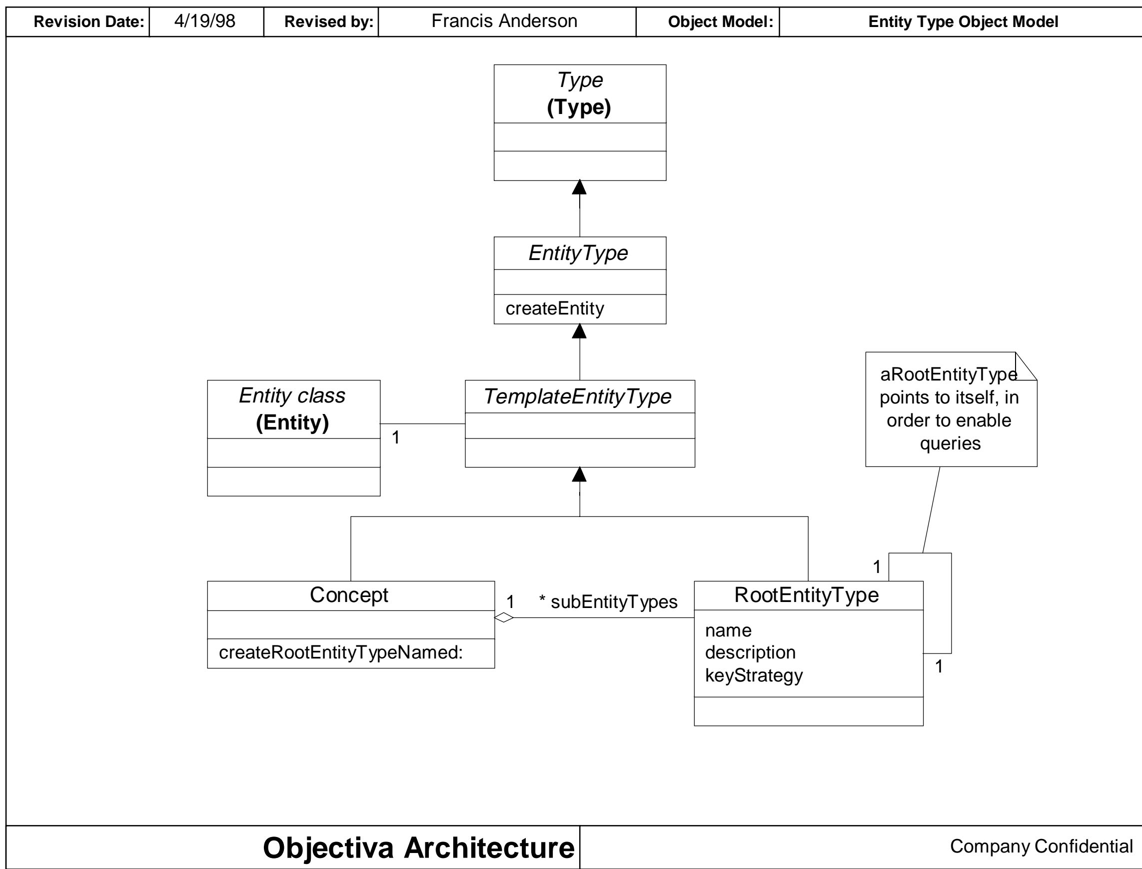

Figure 37: Entity Type Object Model

The Entity Type Object Model (Figure 37) brings us into the Type System of Objectiva, and is an implementation of the Active Object Model [Johnson97] pattern. This pattern is appropriate in those systems that have to support a large number of rapidly changing business rules, which, according to Tom Peters in Thriving on Chaos, is happening more and more frequently. We have chosen the traditional Entity Attribute Relationship model as the vehicle for capturing the rules, which provide the implementation of the Knowledge Level data [Fowler97].

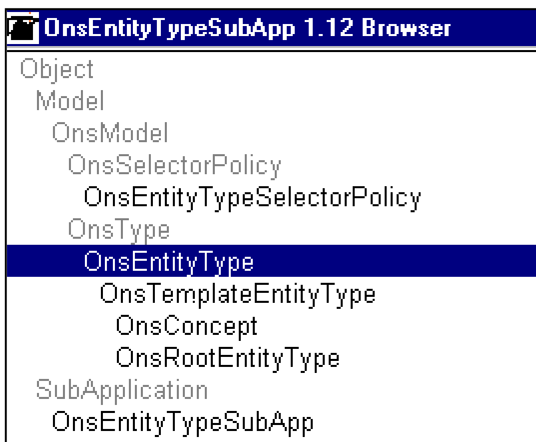

Figure 38: Entity Type *ENVY* Subapplication

Since this model is somewhat abstract, let us recap from the previously stated examples:

- Region is an instance of CONCEPT, with default entityClass REGION .
- Country, State, Numbering Plan Area, etc. are instances of ROOT ENTITY TYPE that are sub (entity) types of Region.
- When we ask a Region type to create a new Region, by default, we will get an instance of REGION, except if we ask Country, in which case we will get an instance of COUNTRY.

We will now look at how we assign ATTRIBUTES to an ENTITY TYPE.

Attribute

An ENTITY TYPE (Country) has attributes, which describe the data values to be recorded for an ENTITY ('United States') of this type. An attribute defines a mapping between an entity type ('Country') and a data type (String). The traditional solution to the definition of attributes is to implement them as instance variables. There is absolutely nothing wrong with this solution, in some circumstances. Everyone understands it. There is no complex framework to learn. But if the business rules change, and we wish to add or remove an attribute of Country, we need to change both the code (behavior) and the schema of the persistence mechanism (structure).

We would also have numerous subclasses of REGION, the need for which we are trying to eliminate. Is this such an effort? Not as much in Smalltalk as in other languages, but we would like to keep code and schema changes to an absolute minimum, since those who can make them are a particularly limited resource, and often sit right on the critical path of implementing a business rule change. Instead, we wish to be able change the business rules through the maintenance of object instances. The problem is that these instances must express rules governing different types of data.

There are a number of patterns that describe problems and solutions in this area, including Property [Foote97] and Observation [Fowler97]. Objectiva adopts the Entity, Attribute, Relationship (EAR) model as the basis of its solution, and uses traditional data modeling terminology, with its standard meanings.

As stated above, we define an attribute as a mapping between an entity type and a data type; this is in contrast to a relationship, which is defined as a mapping between entity types, and is discussed later. Instance variables of an object do not usually make this kind of distinction, the expression "has by value" is sometimes used to attributes, as opposed to the expression "has by reference", which describes relationships.

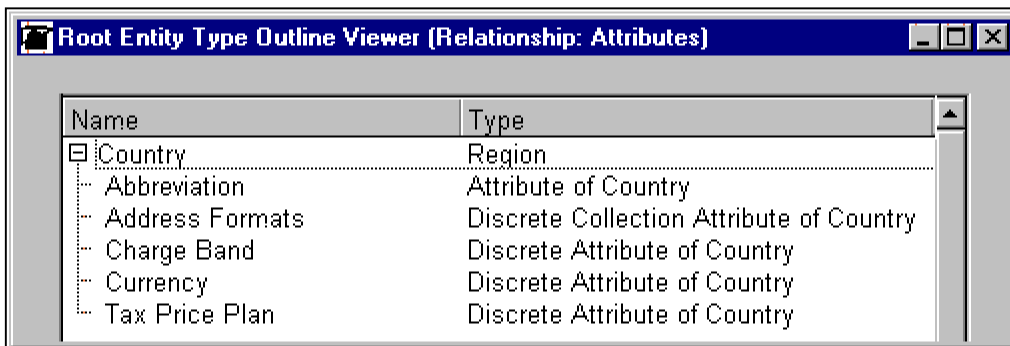

Figure 39: Country Attributes

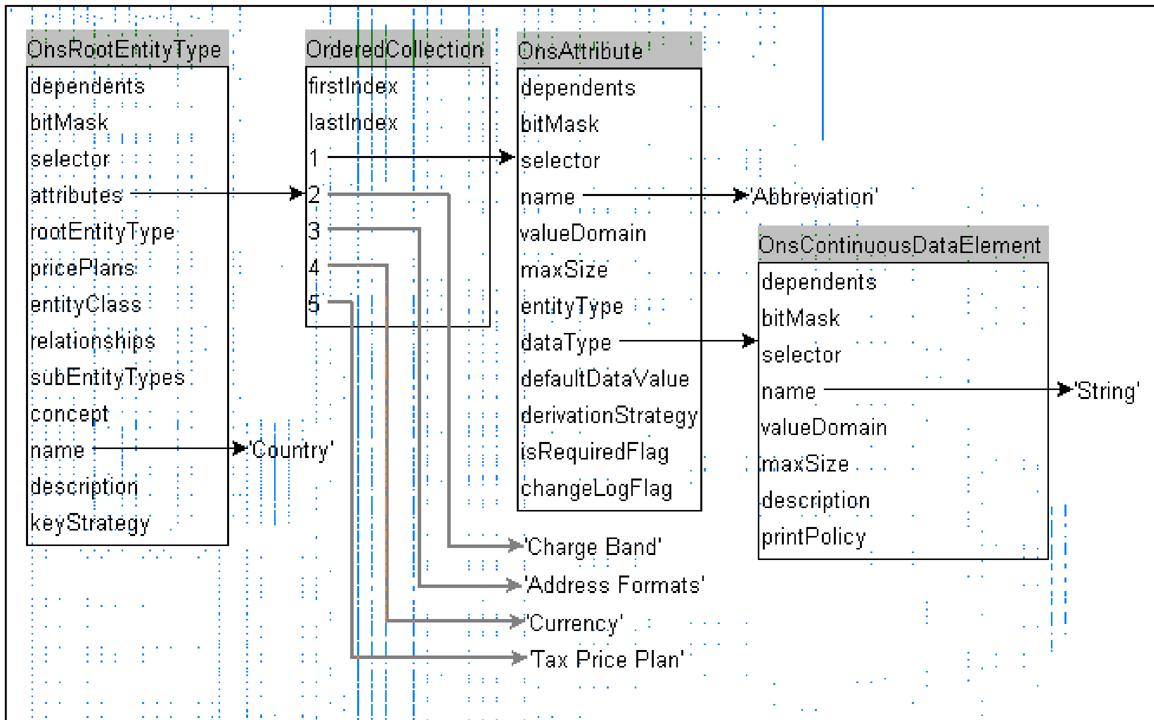

Figure 40: The Attributes of Country

We started to discuss continuous data (e.g. abbreviation) in the section on Data Value above. A continuous data value only has meaning in the context of a single entity. In contrast, a discrete data value (e.g. charge band 1), may be shared by a number of entities, (e.g. all the countries in Europe), since there is only a limited set of values {'0', '1', '2', '3'} that it may take. In this case, the values of the discrete data type (charge band) are strings, i.e. simple data types. Objectiva also allows the values of complex data types (e.g. currency) to be available to a discrete data type.

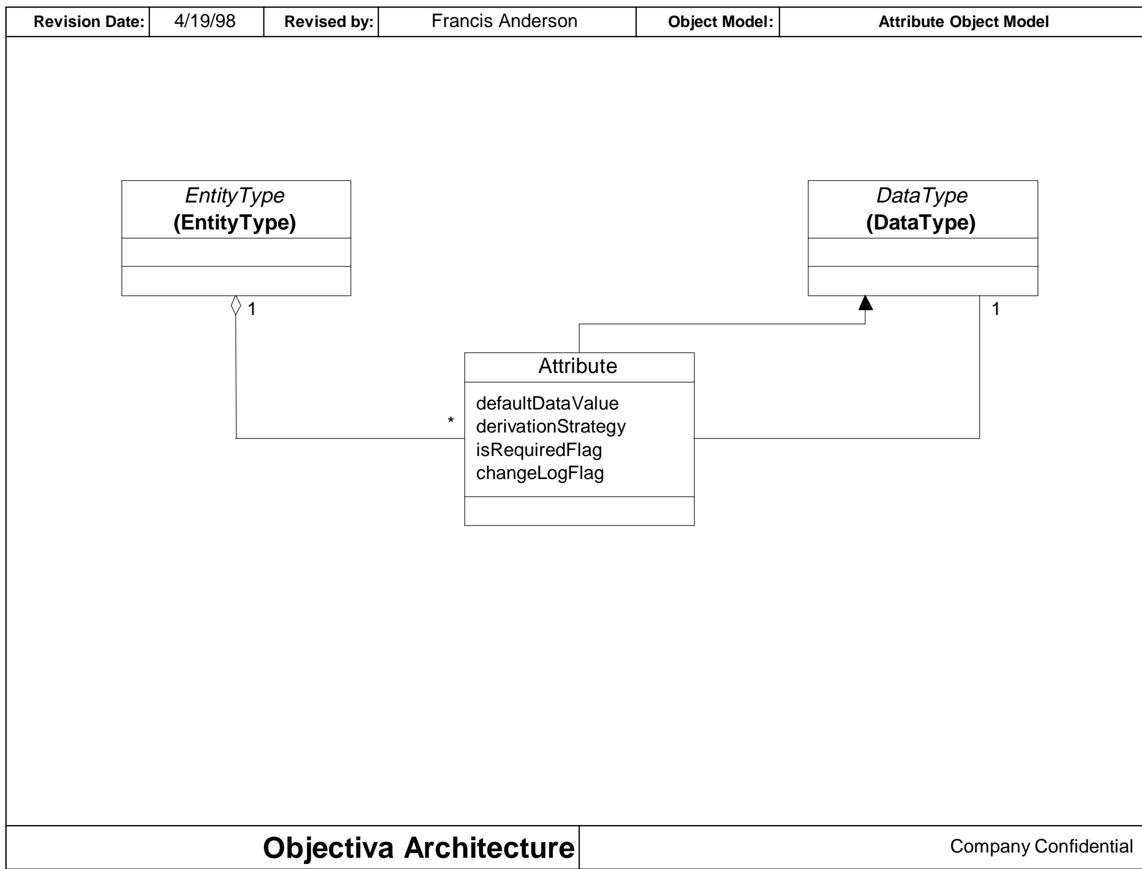

Figure 41: Attribute Object Model

In the Attribute Object Model (Figure 41), ATTRIBUTE could be represented as a UML Association Class. This is an example of a data modeling pattern that is very common, its most well known implementation being the relationships between Order, Order Line and Product, which is an example used in many a modeling class.

At the conceptual level, an entity type (order) may be described by many data types (product), and a data type (product) may describe may entity types (orders). The type of relationship between these concepts is called a many-to-many association. In looking at the association, we discover that there are attributes that we want to record about the association itself. In the case of the order, we want to record the quantity of each product ordered. In the case of the attribute, we may want to change the name (e.g. originating and terminating charge band), specify a default value, etc.

Thus the association itself becomes a class, (ATTRIBUTE and ORDER LINE), and the many-to-many relationship has been resolved into two one-to-many relationships. But these two relationships are of a very different nature. An ENTITY (ORDER) is composed of ATTRIBUTES (ORDER LINES). This should be represented in UML is a filled in diamond but my Visio template does not support this. Whereas, the type of ATTRIBUTE is specified by DATA TYPE, and the type of ORDER LINE is specified by PRODUCT. Thus, in both cases, we have replaced an association relationship by the combination of a composition and a classification relationship. The classification relationship is another name for the

TypeObject pattern. So we have another example of nested TypeObject, which, since ATTRIBUTE is also a DATA TYPE, is actually recursive.

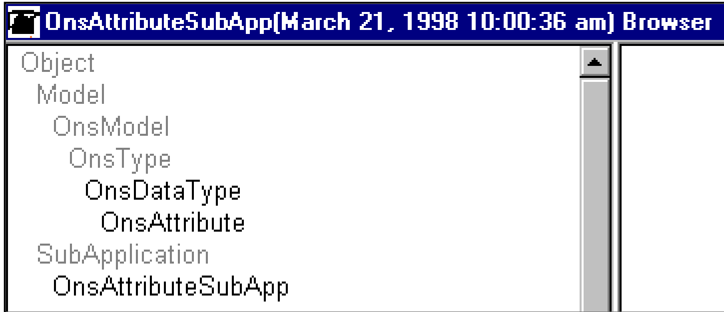

Figure 42: Attribute *ENVY* Subapplication

Before discussing the different data types supported by Objectiva, we will take a look at how the Entity and Data major components of the DME fit together.

Entity and Data Major Components

The Package Diagram of a Major Component places the object model of its Minor Components in context. So far, we have been following class level links between object models, which are implemented using a "foreign key" type of approach. For example, the Attribute Object Model (Figure 41) does not define ENTITY TYPE and DATA TYPE, but references them from the minor components of the same name, as indicated by the bold face label in parentheses beneath the class name. The Attribute and Entity Type minor components are both part of the Entity major component. Data Type, however, is part of the Data major component. Since we must reference Data in order to fully describe Entity, Entity is dependent upon Data.

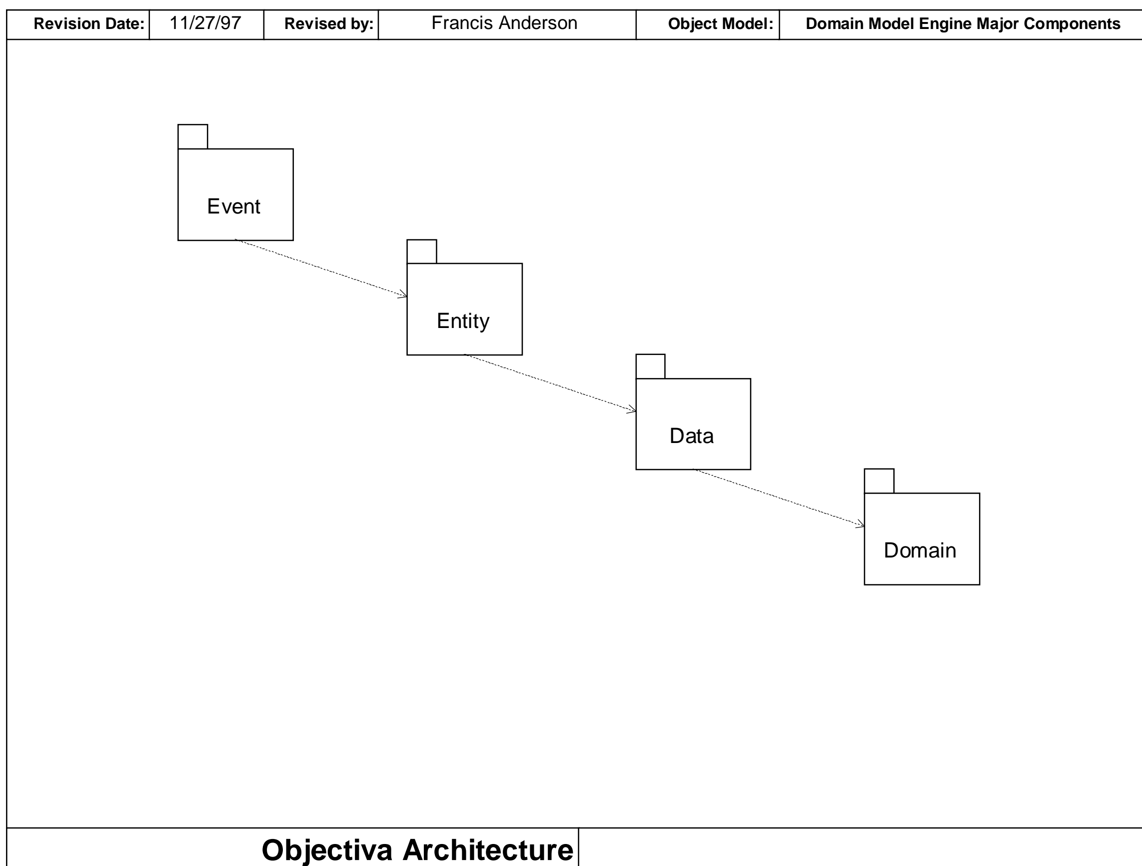

Figure 43: Major Components of the Domain Model Engine

The Major Components of the Domain Model Engine (Figure 43) also depicts the path we followed when tracing the inheritance in the Operational Level from REGION to ENTITY and ENTITY CONTEXT (in Entity), to DATA VALUES HOLDER (in Data), to NODE (in Domain).

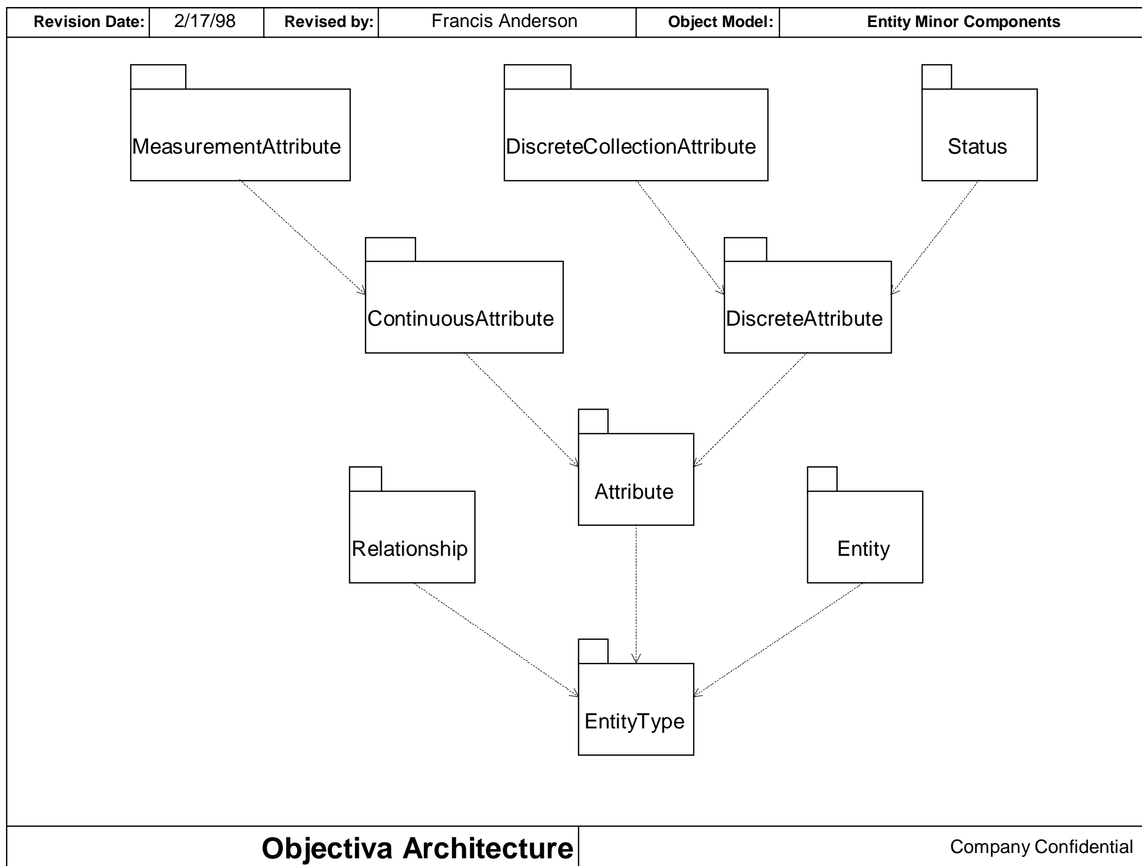

Figure 44: Minor Components of Entity

The Minor Components of Entity (Figure 44) shows the dependency of Attribute upon Entity Type. This is an existence dependency: without an entity type, an attribute could not exist. Existence dependency is a property of the composition relationship: a component (child) is existence dependent upon its composite (parent). If the composite (anEntityType) is deleted, its components (attributes) are too; if the composite is copied, its components are too (anAttribute is the mapping between one Entity Type and one Data Type).

So the Minor Components of Entity (Figure 44) tells us that we cannot really understand Attribute unless we understand Entity Type. Also, it tells us that we have a number of different types of Attribute, with the continuous / discrete discrimination playing a very important role. This discrimination first occurs in the Data major component of the DME upon which Entity is dependent.

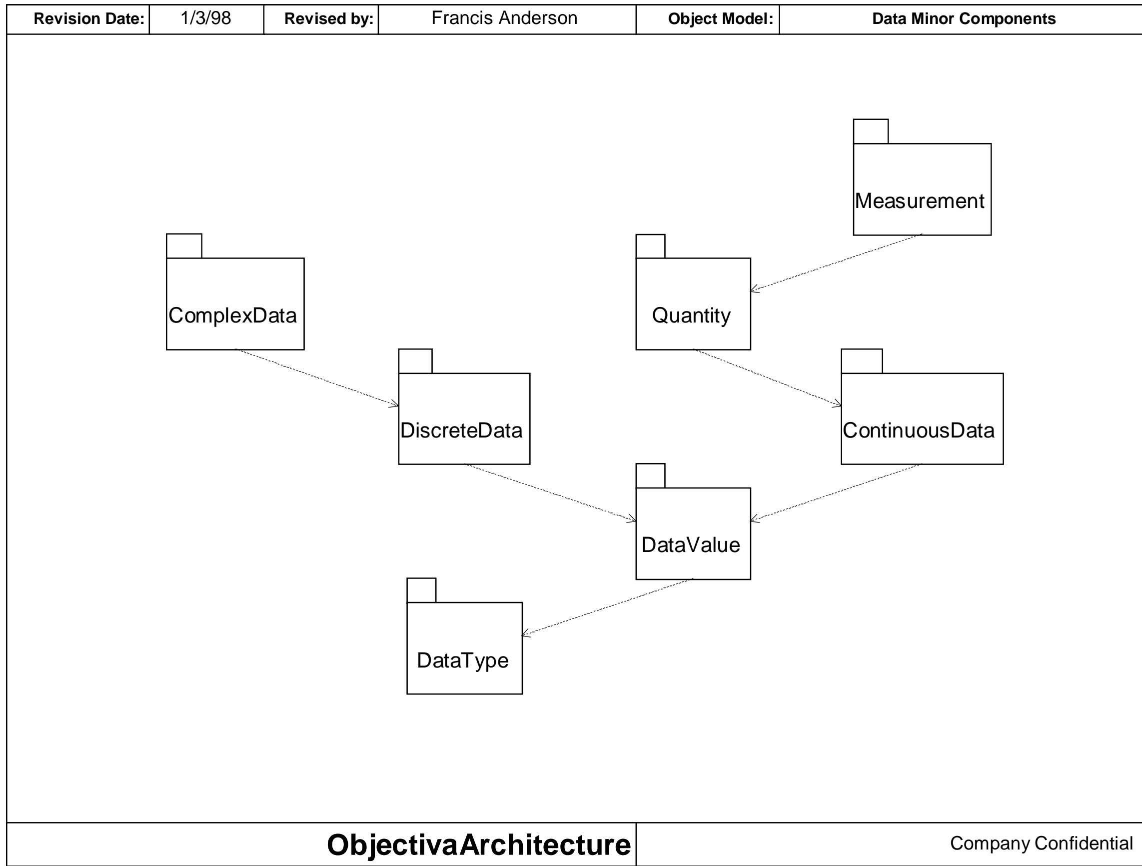

Figure 45: Minor Components of Data

The Minor Components of Data (Figure 45) shows us that data comes in different "dimensions":

- Discrete Data has a limited domain of available values, which may be strings or complex objects.
- Continuous Data has an unrestricted domain of values, which are representable as strings or are the quantity of some unit, which may be recorded as a measurement.

Continuous Data

Abbreviation 'USA' is a continuous data value, since the range of possible values that it could take is effectively infinite, limited only by the maximum length that we choose to allow for the value, and the value can be represented as a String. Basically, what this means is that a continuous attribute is represented as an input field on a user interface, see the United States Entity Editor (Figure 16).

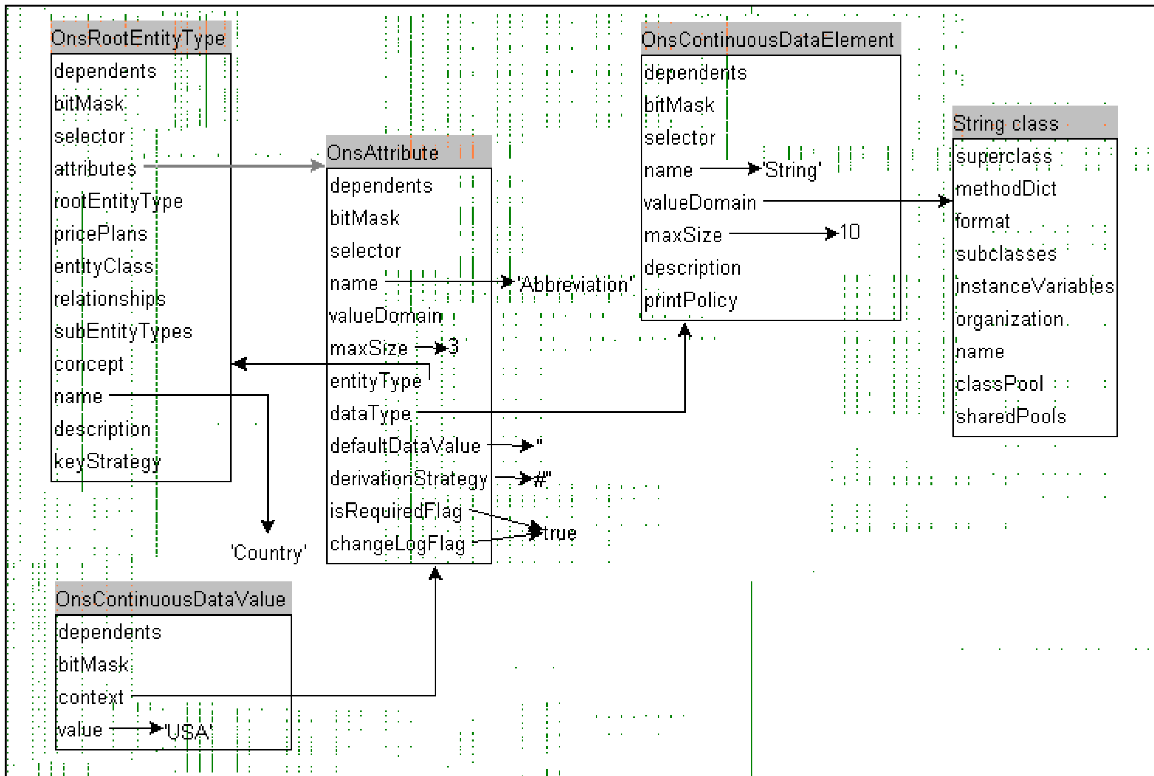

Figure 46: Instance Diagram of Continuous Data Value

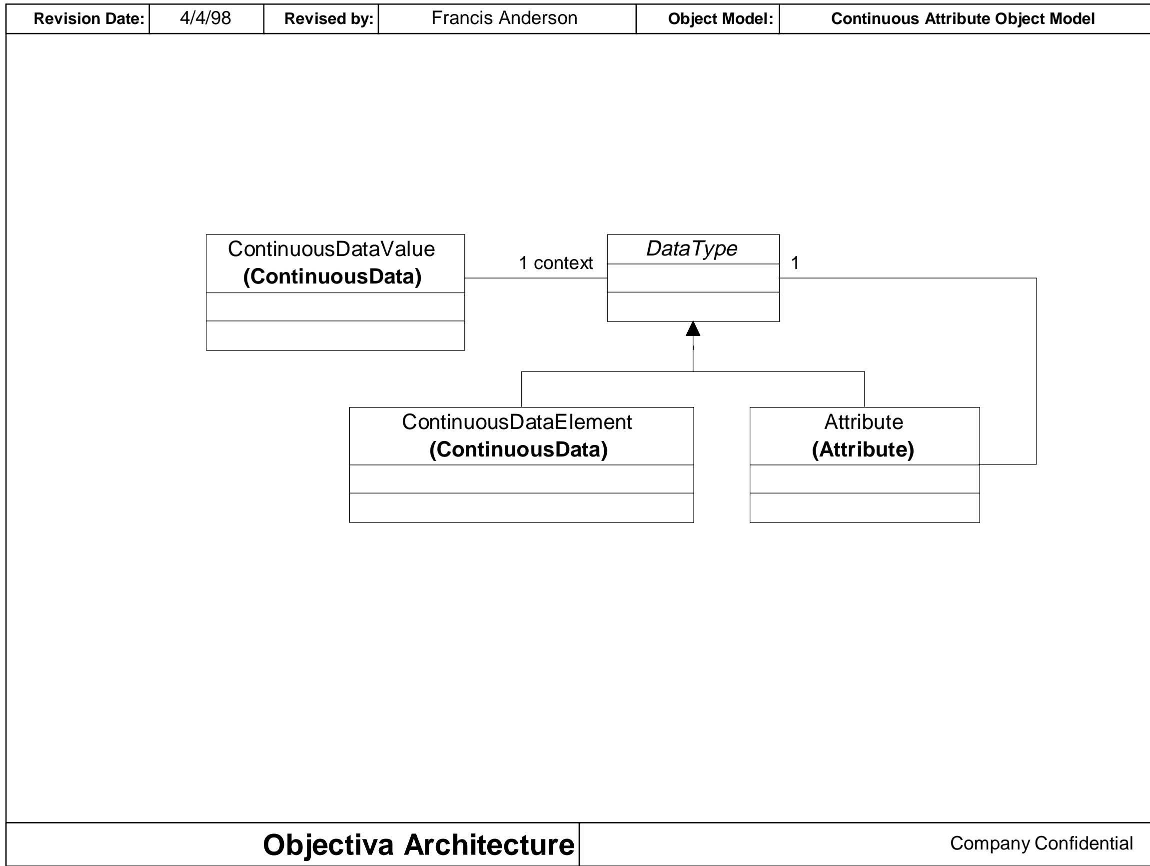

Figure 47: Continuous Attribute Object Model

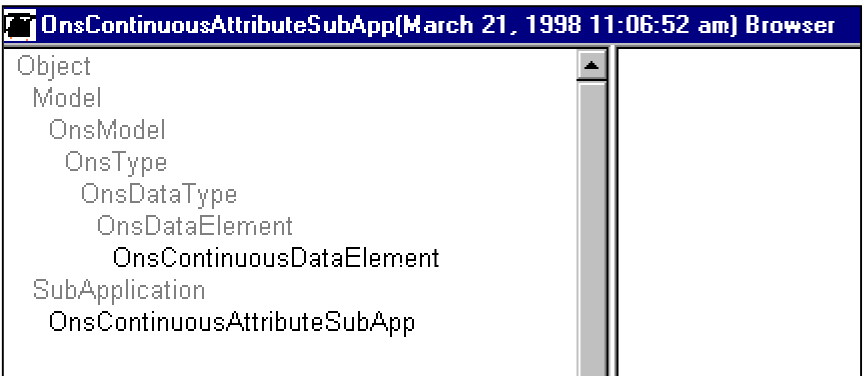

Figure 48: Continuous Attribute *ENVY* Subapplication

Dimension Outline Viewer (Relationship: Children)

Name	Type
Continuous Data Element	Dimension
Date	Date
Duration	OnsDuration
North American Phone Numbe	OnsNorthAmericanPhoneNumber
Number	Number
String	String
String Collection	OnsStringCollection
Time	Time

Figure 49: Continuous Data Elements

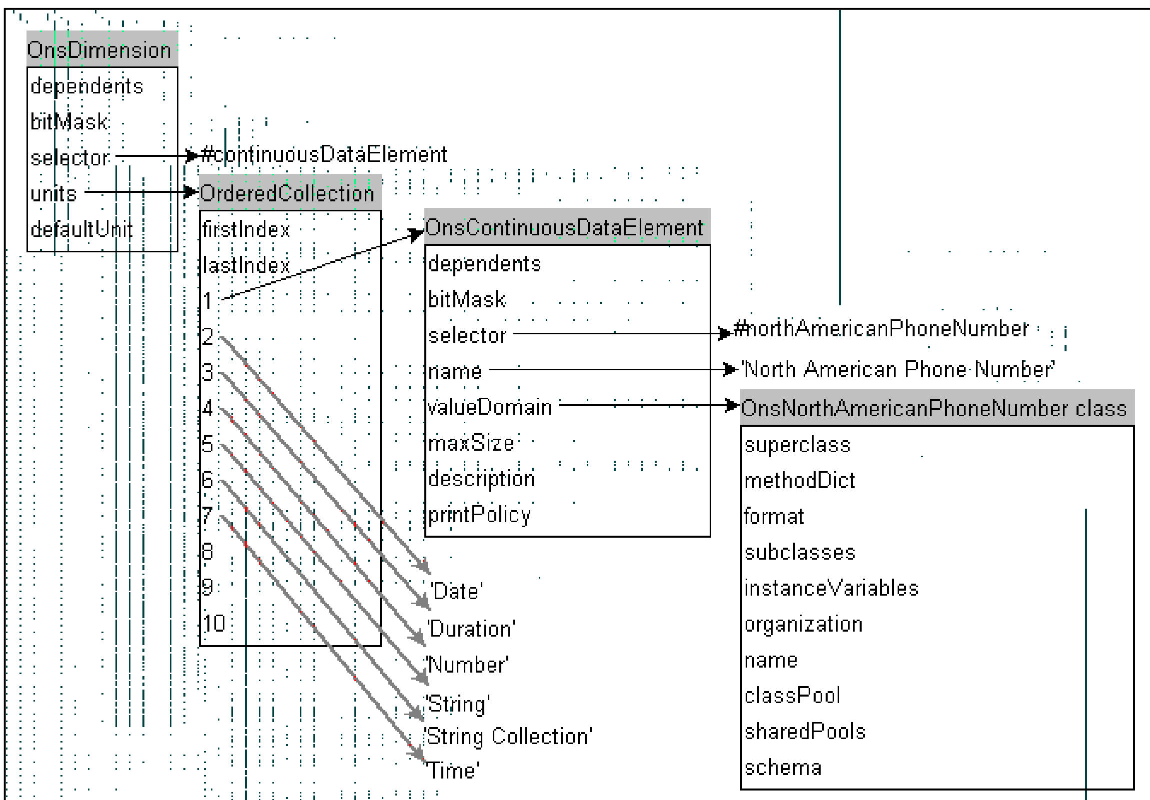

Figure 50: Continuous Data Elements

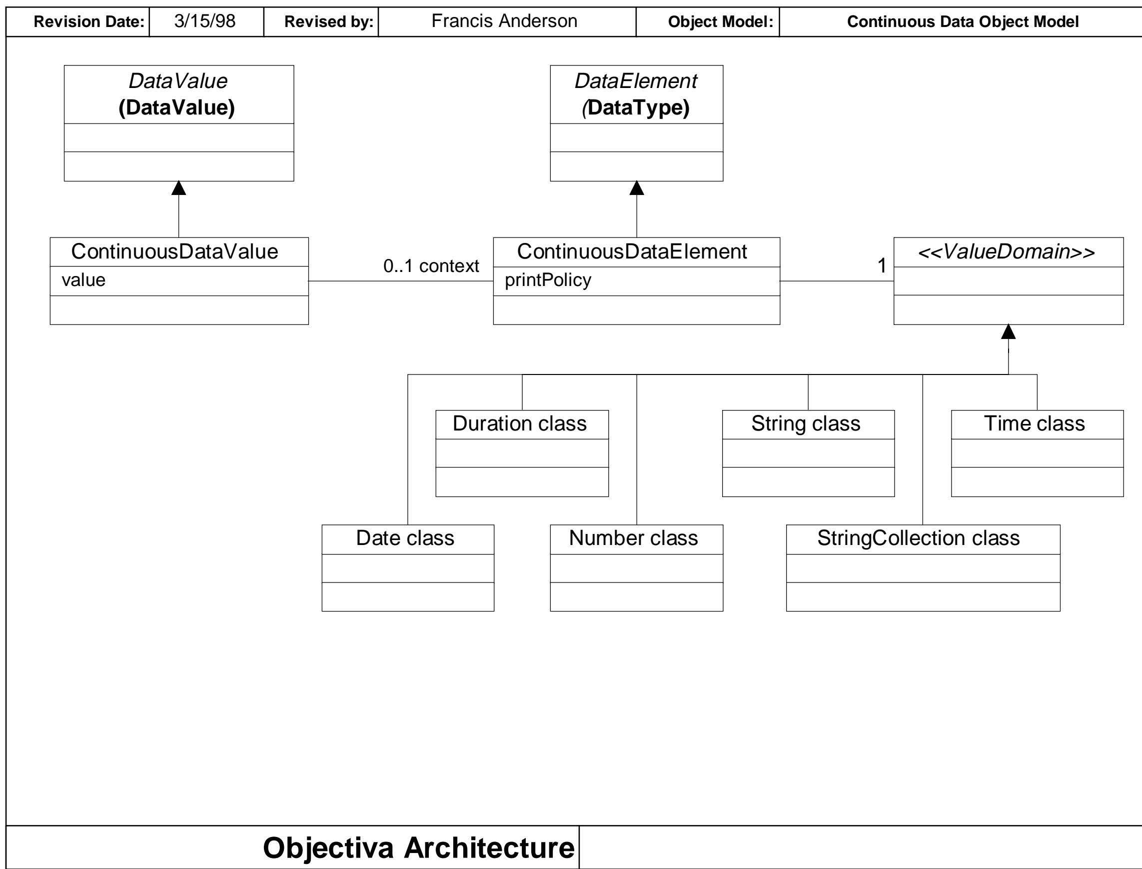

Figure 51: Continuous Data Object Model

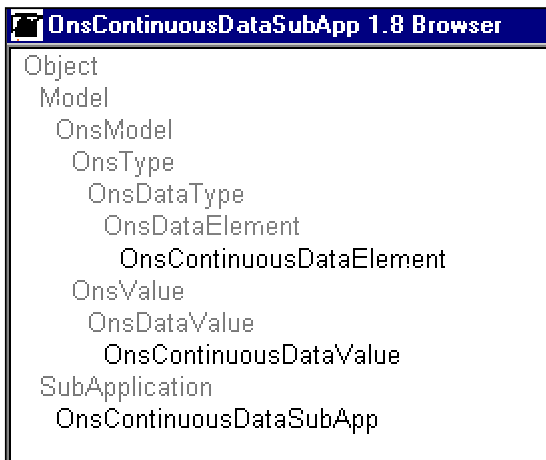

Figure 52: Continuous Data *ENVY* Subapplication

Discrete Data

Charge Band '0' is a DISCRETE DATA VALUE, since we have constrained its available values to '0', '1', '2', or '3'. Basically, what this means is that a discrete attribute is represented as a combo box on a user interface, see the United States Entity Editor (Figure 13).

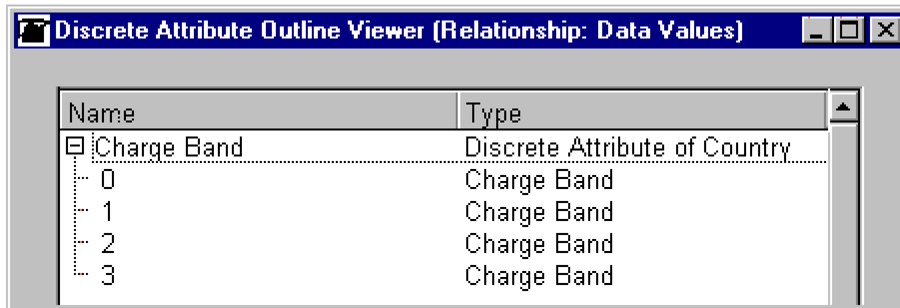

Figure 53: Charge Band Data Values

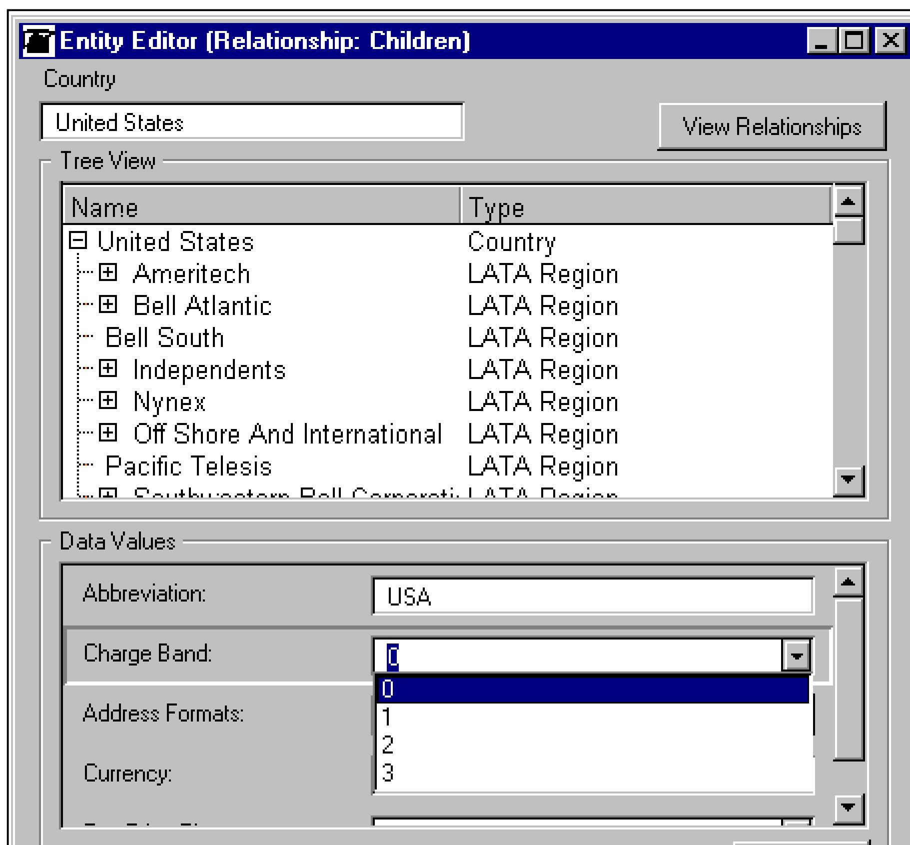

Figure 54: Charge Band Combo Box

This is an example of a Category Observation [Fowler97]. In this case, a DISCRETE DATA VALUE instance may be shared by a number of objects, and the meaning of Charge Band 0 is distinct from the meaning of Charge Band 1. Due to its sharing, Discrete Data Value is an example of the Flyweight pattern.

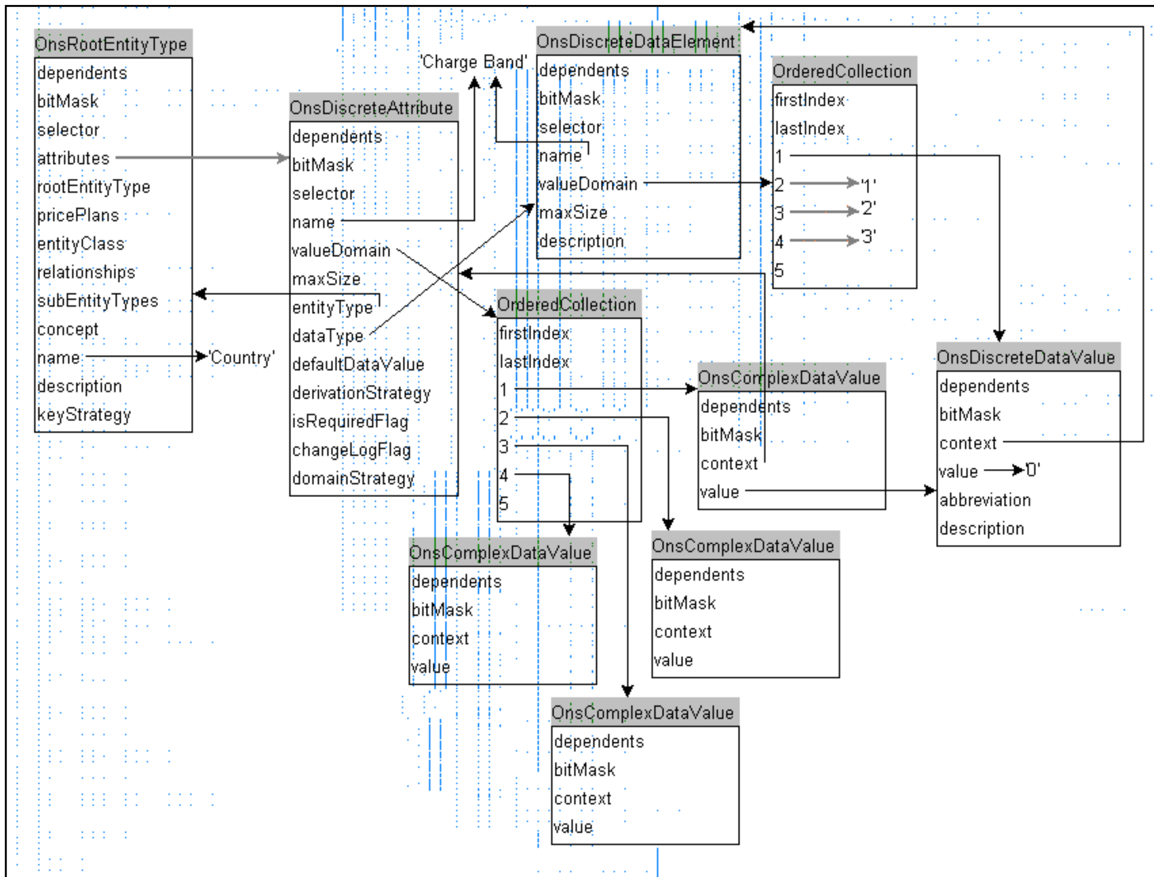

Figure 55: Instance Diagram of Discrete Data

The Instance Diagram of a Discrete Data (Figure 55) shows us that, like CONTINUOUS DATA VALUE, DISCRETE DATA VALUE places a String ('0') in the context of a DATA TYPE (Charge Band). In this case, however, the subclass of DATA TYPE is DISCRETE DATA ELEMENT. A discrete Data Type (Charge Band) stores its available values {0, 1, 2, 3} in the valueDomain collection.

We may choose to describe a number of different types of Region, (e.g. Country, State) in terms of a Charge Band, or we may choose to describe a phone call in terms of its originating and terminating Charge Band. We will describe how this is done later on, but for now we will say that we are placing a Discrete Data Type (Charge Band) within the context of an Entity Type (Country). This context is propagated down to the values of the data type, but the meaning of Charge Band 0 has not changed. We may, however, have a different domain of values available to State than Country, and within the context of a phone call, the differentiation between originating and terminating is critical. Thus, an ENTITY CONTEXT does not directly hold on to a DISCRETE DATA VALUE; instead, it holds on to a COMPLEX DATA VALUE..

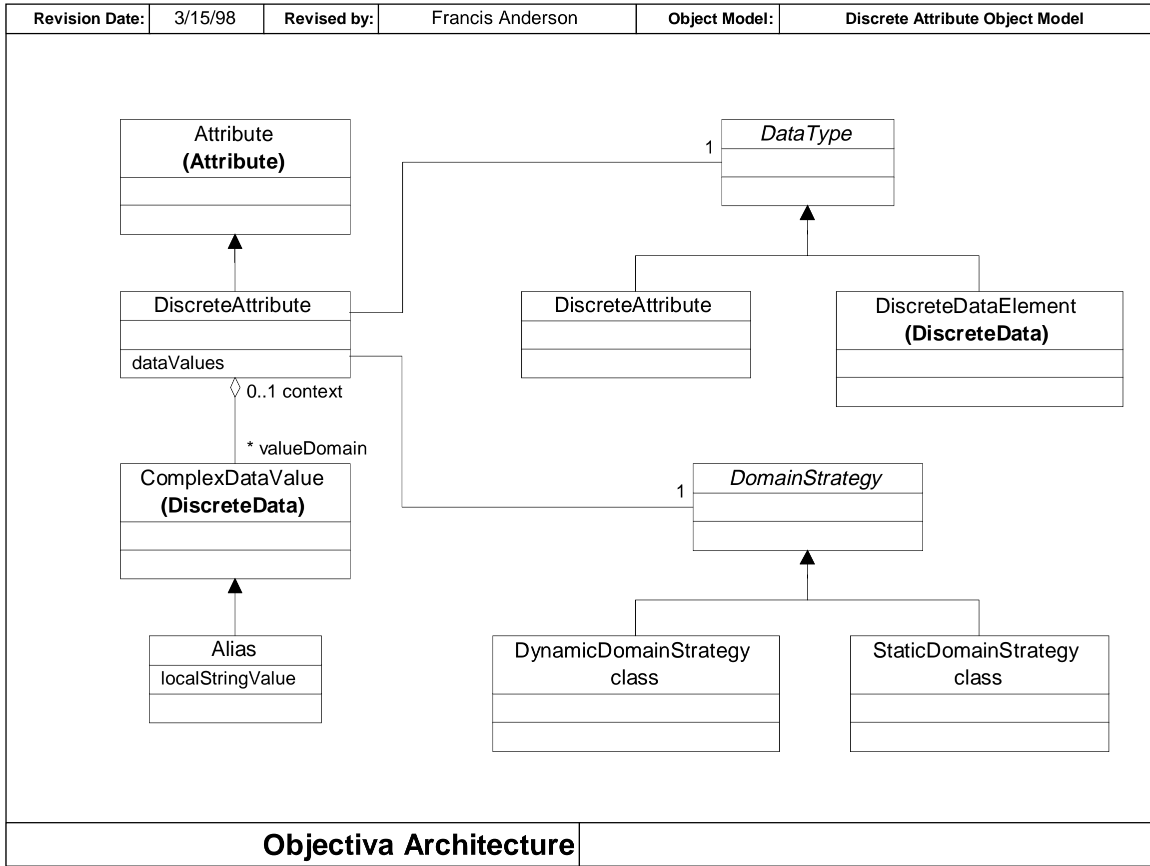

Figure 56: Discrete Attribute Object Model

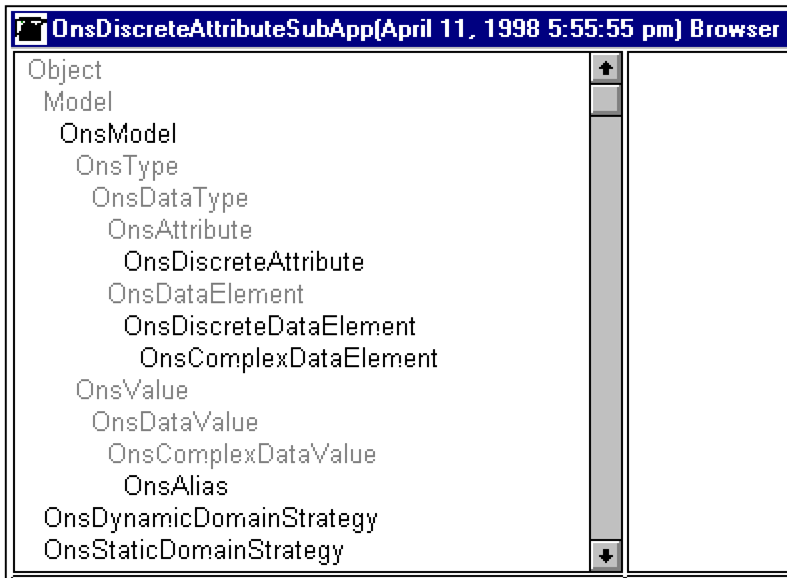

Figure 57: Discrete Attribute ENVY Subapplication

Dimension Outline Viewer (Relationship: Children)

Name	Type
Discrete Data Element	Dimension
Architectural Order	Discrete Data Element
Authorization Status	Discrete Data Element
Charge Band	Discrete Data Element
Day Of Week	Discrete Data Element
Directionality	Discrete Data Element
Equipment Status	Discrete Data Element
Logical Operator	Discrete Data Element
Phone Type	Discrete Data Element
Relational Operator	Discrete Data Element
Relationship	Discrete Data Element
SIM Card Type	Discrete Data Element
Skill Level	Discrete Data Element
Time Period Type	Discrete Data Element

Figure 58: Discrete Data Elements

Discrete Data Element Outline Viewer (Relationship: Data Values)

Name	Type
Equipment Status	Discrete Data Element
Active	Equipment Status
Packaged	Equipment Status
Pending Activation	Equipment Status
Pending Provisioning	Equipment Status
Provisioned	Equipment Status
Received	Equipment Status
Shipped	Equipment Status
Sold	Equipment Status

Figure 59: Discrete Data Values

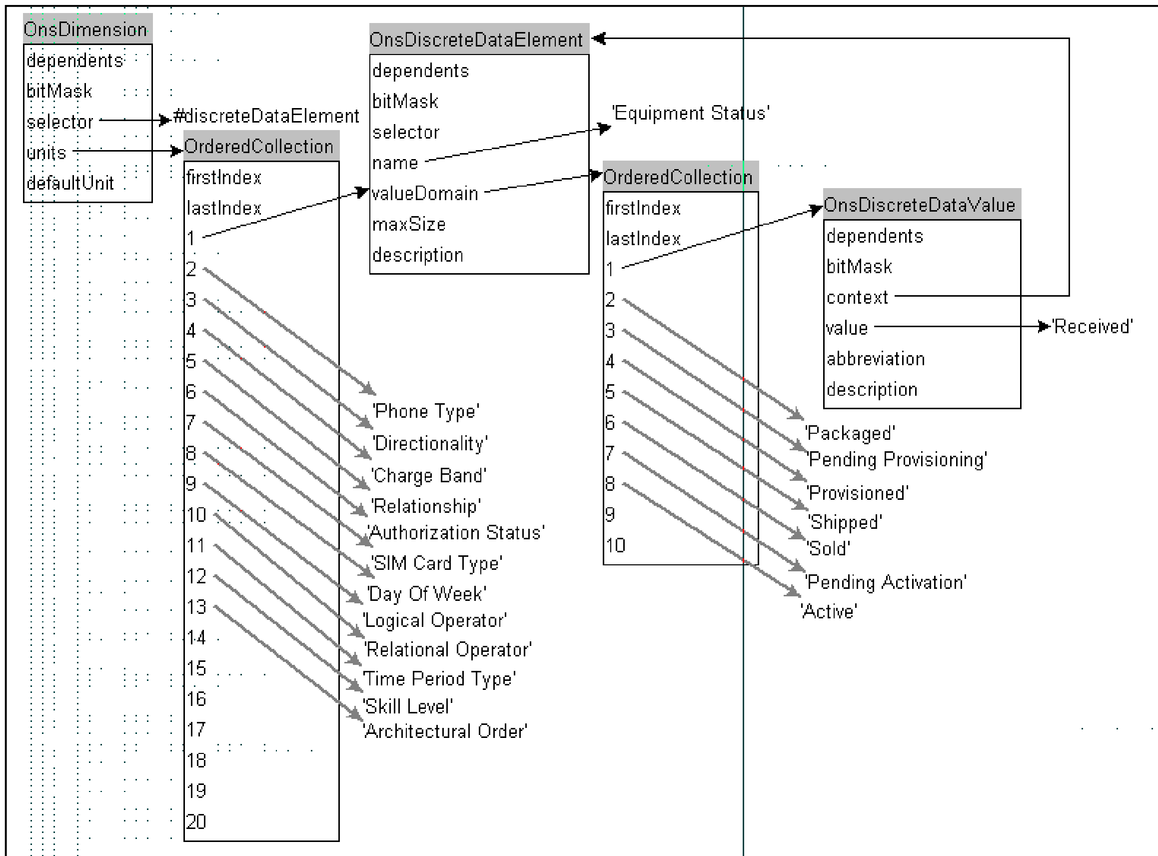

Figure 60: Instance Diagram of Discrete Data Elements

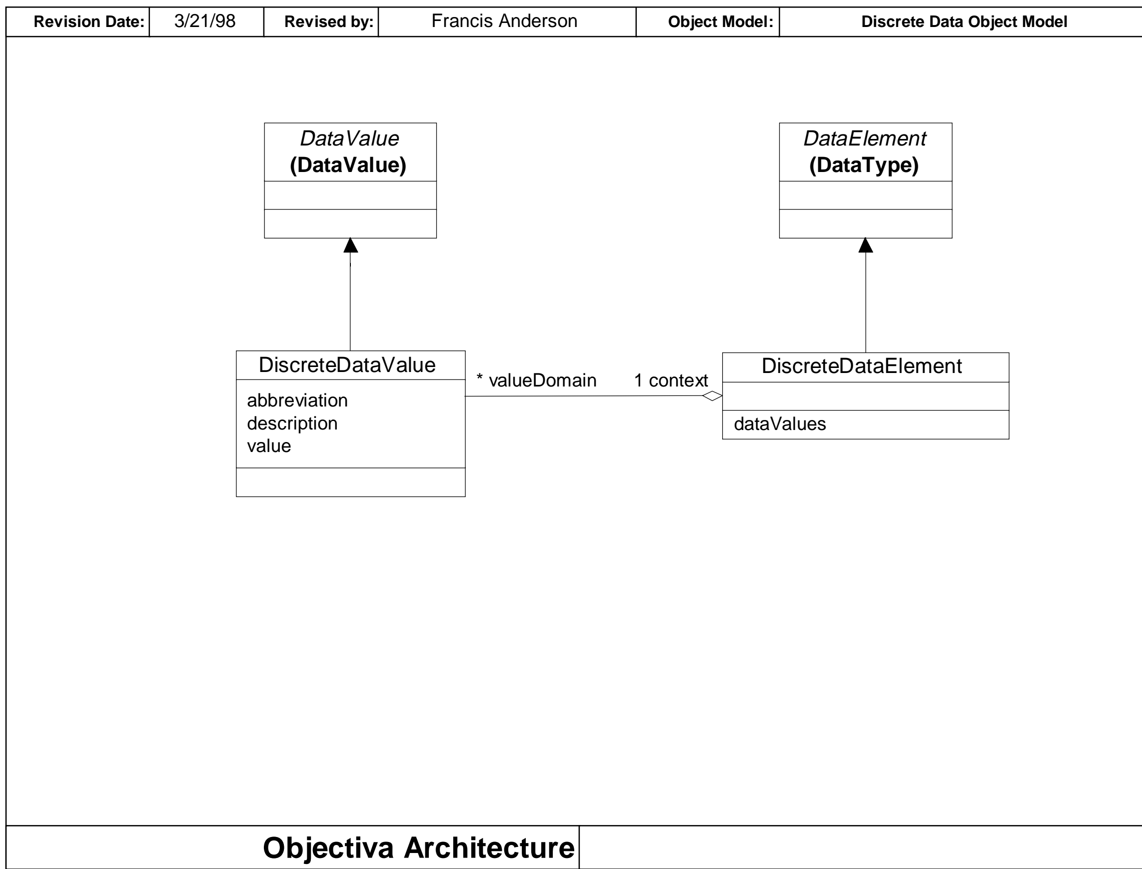

Figure 61: Discrete Data Object Model

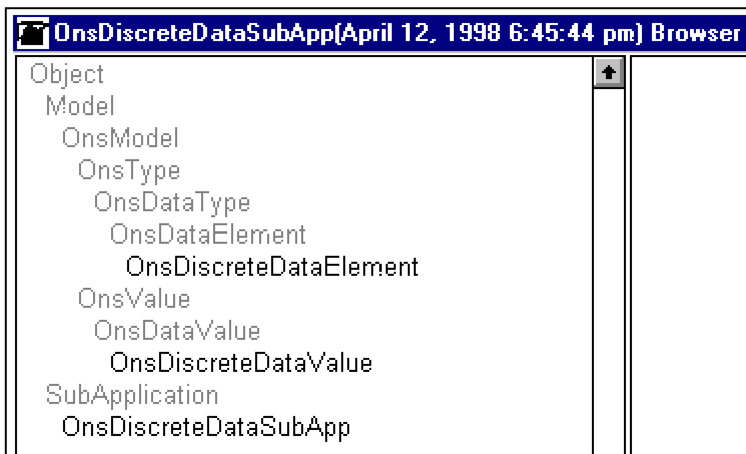

Figure 62: Discrete Data *ENVY* Subapplication

Complex Data

Currency is a Complex Data Element since the values that it may take are complex objects (instances of CURRENCY), rather than just simple strings. Complex data allows us to handle the problem introduced by the statement "One man's entity is another man's attribute". Basically, what this means is that a discrete attribute with a complex data type is represented as a combo box on a user interface but the entries in the drop down are complex objects, not just strings, see the United States Entity Editor (Figure 13).

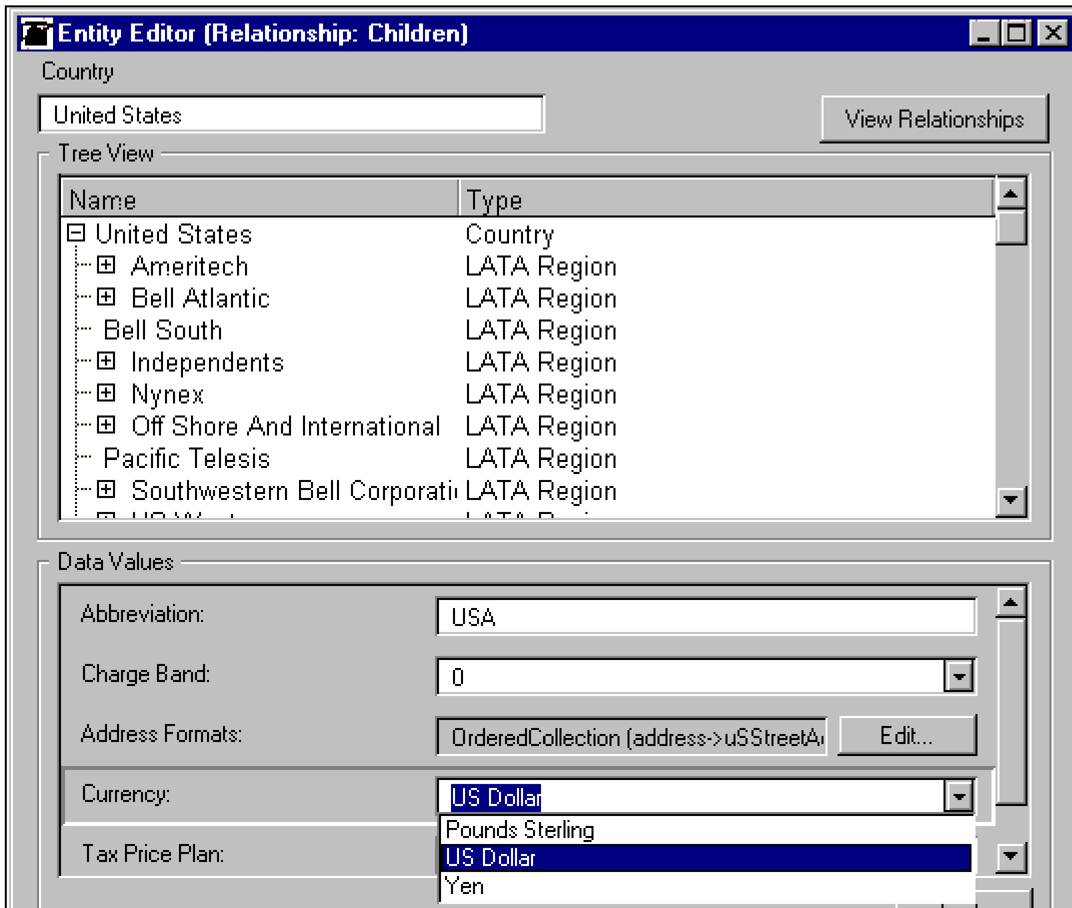

Figure 63: Currency Selection

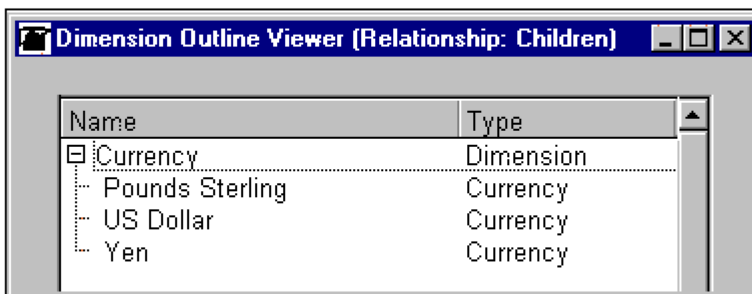

Figure 64: Currency Dimension

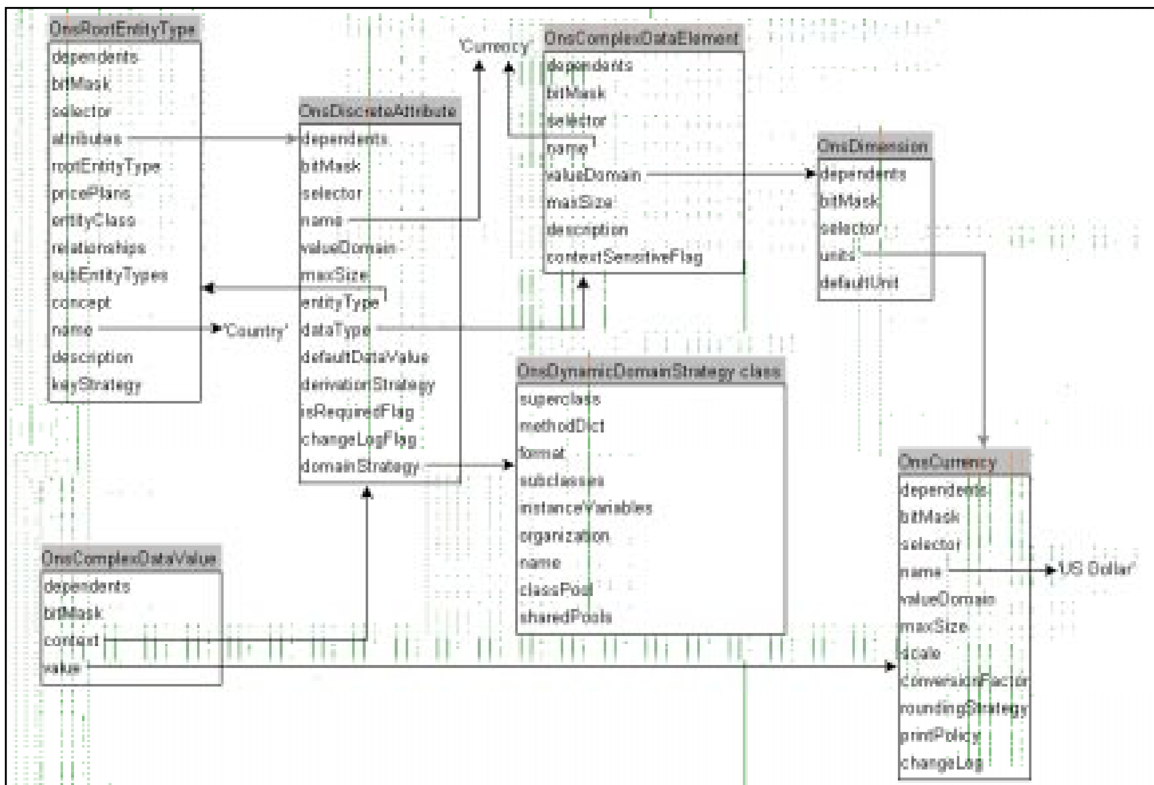

Figure 65: Instance Diagram of a Complex Data Value

The Instance Diagram of a Complex Data Value (Figure 65) demonstrates that the value of a COMPLEX DATA VALUE may be any kind of complex object, in this case an instance of CURRENCY. Although there is only a limited set of currencies, they are instances of a complex class (CURRENCY), rather than a simple class (e.g. String). The same is true for tax price plan.

Dimension Outline Viewer (Relationship: Children)

Name	Type
Complex Data Element	Dimension
Address	Complex Data Element
Base Price Plan	Complex Data Element
Charge Data Element	Complex Data Element
Country	Complex Data Element
County	Complex Data Element
Currency	Complex Data Element
Discrete Data Element	Complex Data Element
Duration	Complex Data Element
Event	Complex Data Element
Measurement Data Element	Complex Data Element
Numbering Plan Area	Complex Data Element
Phone Model	Complex Data Element
Rate Center	Complex Data Element
Root Time Period Definition	Complex Data Element
SIM Card Model	Complex Data Element
State	Complex Data Element
Tax Price Plan	Complex Data Element

Figure 66: Complex Data Elements

Complex Data Element Outline Viewer (Relationship: Children)

Name	Type
Numbering Plan Area	Complex Data Element
207	Numbering Plan Area
214	Numbering Plan Area
800	Numbering Plan Area
817	Numbering Plan Area
888	Numbering Plan Area
972	Numbering Plan Area

Figure 67: Complex Data Values

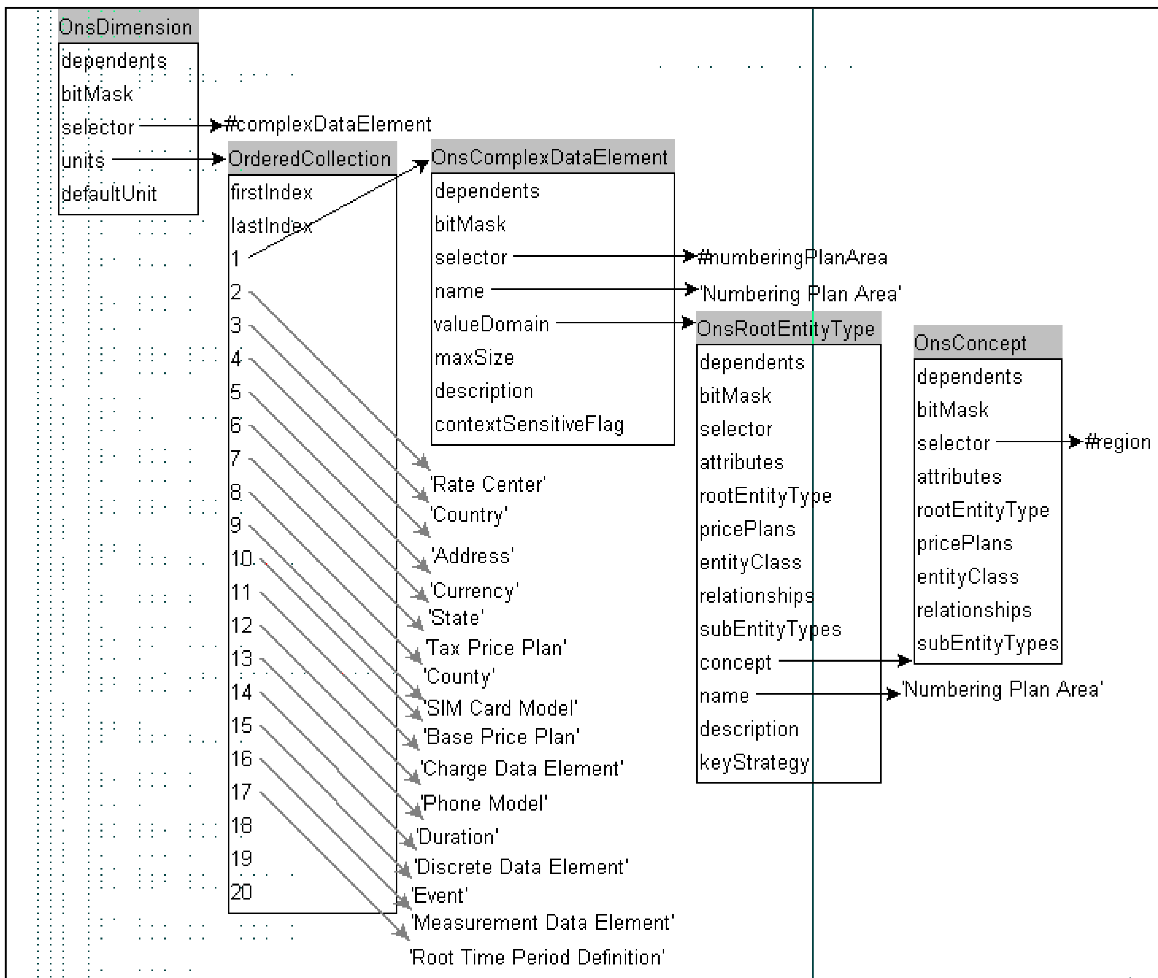

Figure 68: Instance Diagram of Complex Data Elements

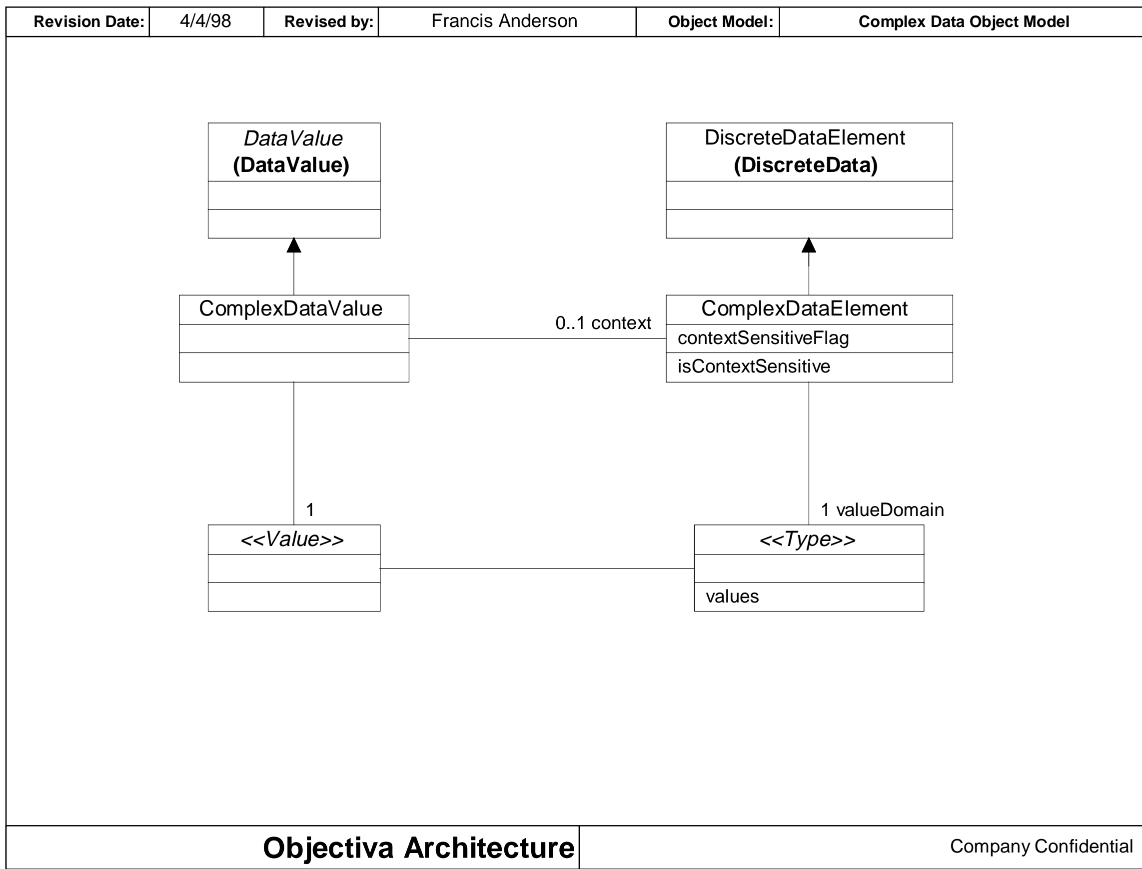

Figure 69: Complex Data Object Model

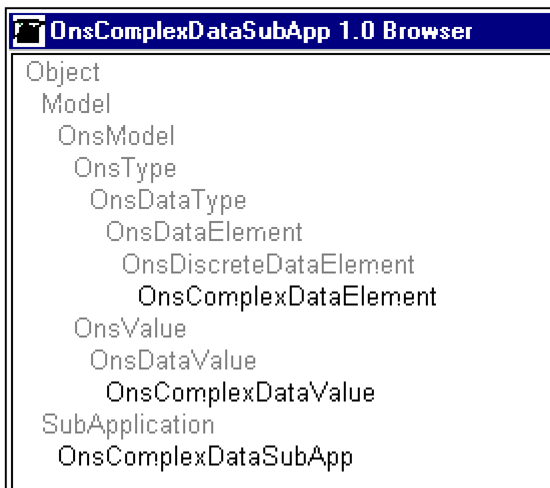

Figure 70: Complex Data *ENVY* Subapplication

Discrete Collection Attribute

Address formats {'US Postal Address' 'US Post Office Box'} is a discrete collection data value, since, although we have constrained the available values, multiple values may be selected. In this case, the available values are also complex objects. Selection via a user interface is performed via an "assign and remove" metaphor.

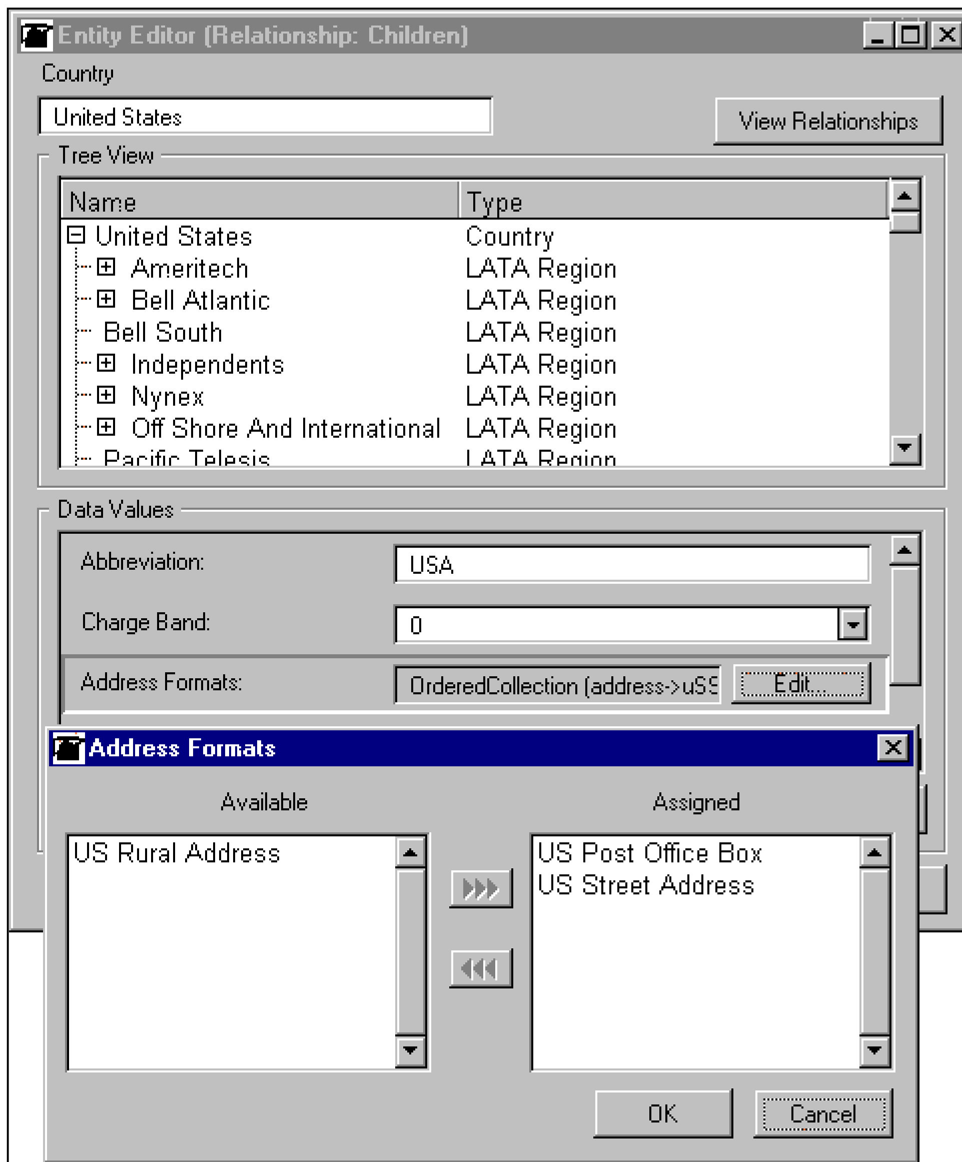

Figure 71: Address Format Selection

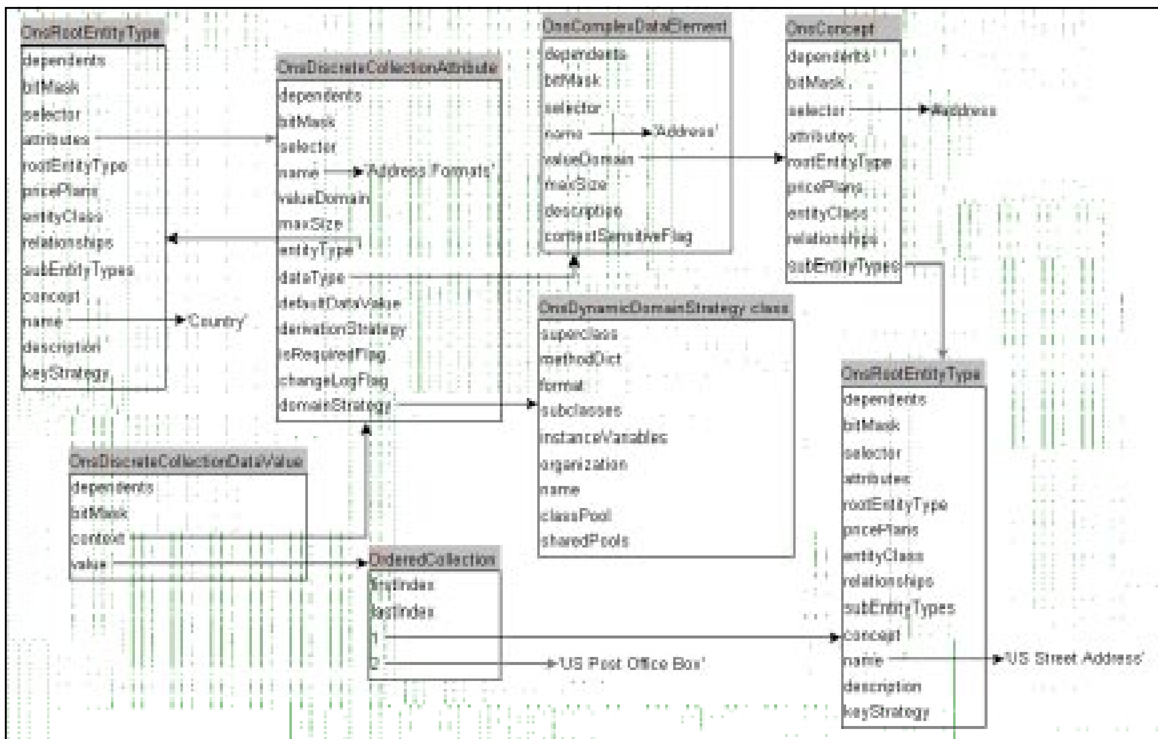

Figure 72: Instance Diagram of a Discrete Collection Data Value

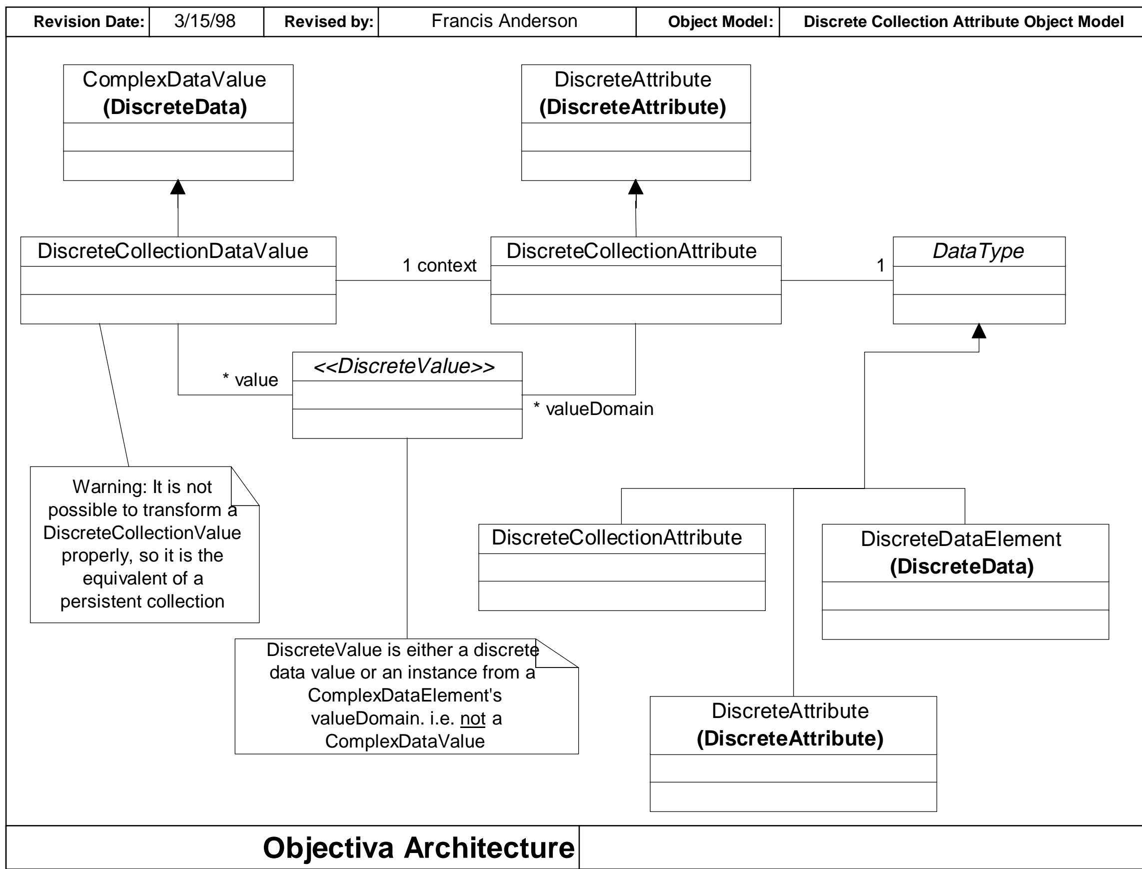

Figure 73: Discrete Collection Attribute Object Model

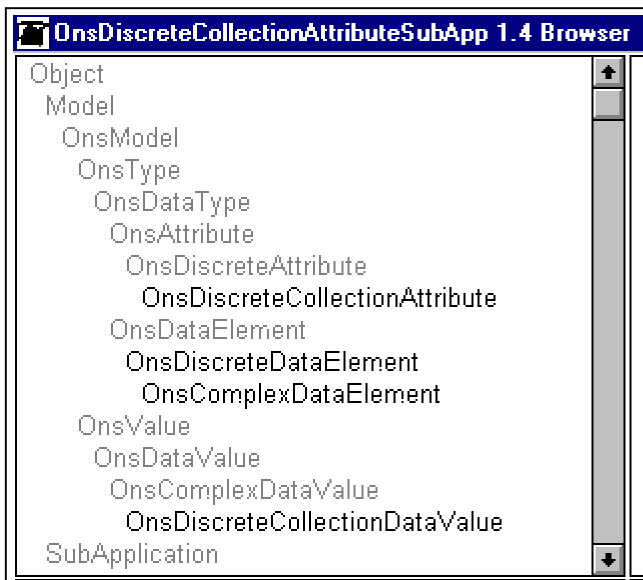

Figure 74: Discrete Collection Attribute *ENVY* Subapplication

Data Type

The problem with nesting the TypeObject pattern is that it is very hard to know when to stop. We have not yet discussed how Objectiva handles quantities, we will do this when describing the Currency business object. But, suffice it to say, Quantity requires yet another application of TypeObject, because we measure quantities (e.g. 5 feet) in terms of a unit (feet), which are convertible to other units (e.g. inches). Convertible units are in the same dimension (e.g. distance).

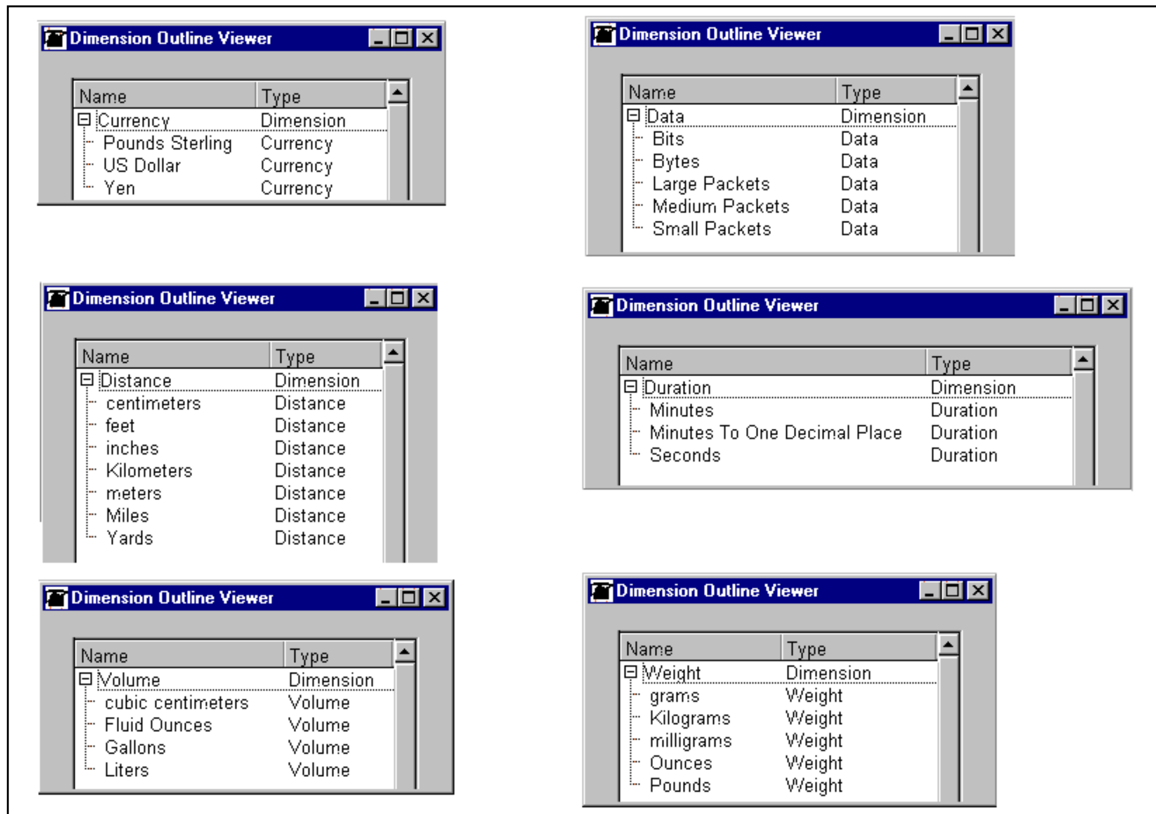

Figure 75: Unit of Quantity Dimensions

Now, particularly with object databases, reachability is an issue. This means that all objects must be reachable from a few well-known objects that act as the roots of the graph of objects. It is desirable to reduce the number of well-known, or bound, objects.

By their very definition, data elements are defined in a context-free manner, but how many data elements will there be in an Enterprise, and how volatile will they be. Similarly, concepts and countries are context-free.

Dimensions, on the other hand should be relatively stable, and make a nice foundation for the Objectiva Type System. So we apply the power type concept one last time, and introduce an additional set of dimensions that correspond to the subclasses of DATA ELEMENT. We call these "class dimensions", since we put the knowledge that a class is also a dimension in the instance creation (new) method of the class, which, in addition to creating a new instance, adds it to the units for the corresponding dimension.

CONTINUOUS DATA ELEMENT class>>newNamed: aString ofType: aClass

```
^self addInstance:
        (self new initializeNamed: aString
                ofType: aClass)
```

DATA ELEMENT class>>addInstance: anObject

```
^OnsDimension addUnit: anObject
        toDimension: self name stripPrefix
```

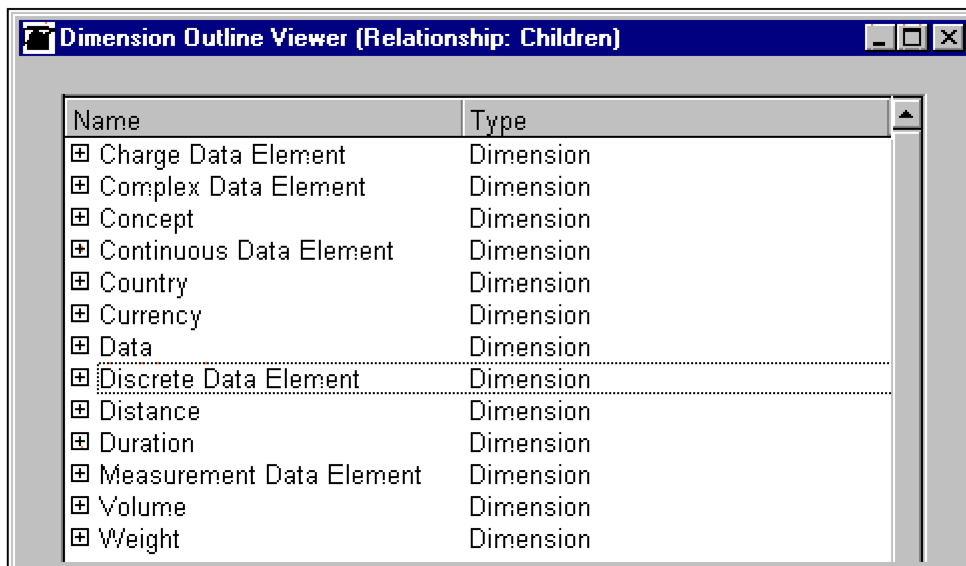

Figure 76: Objectiva Dimensions

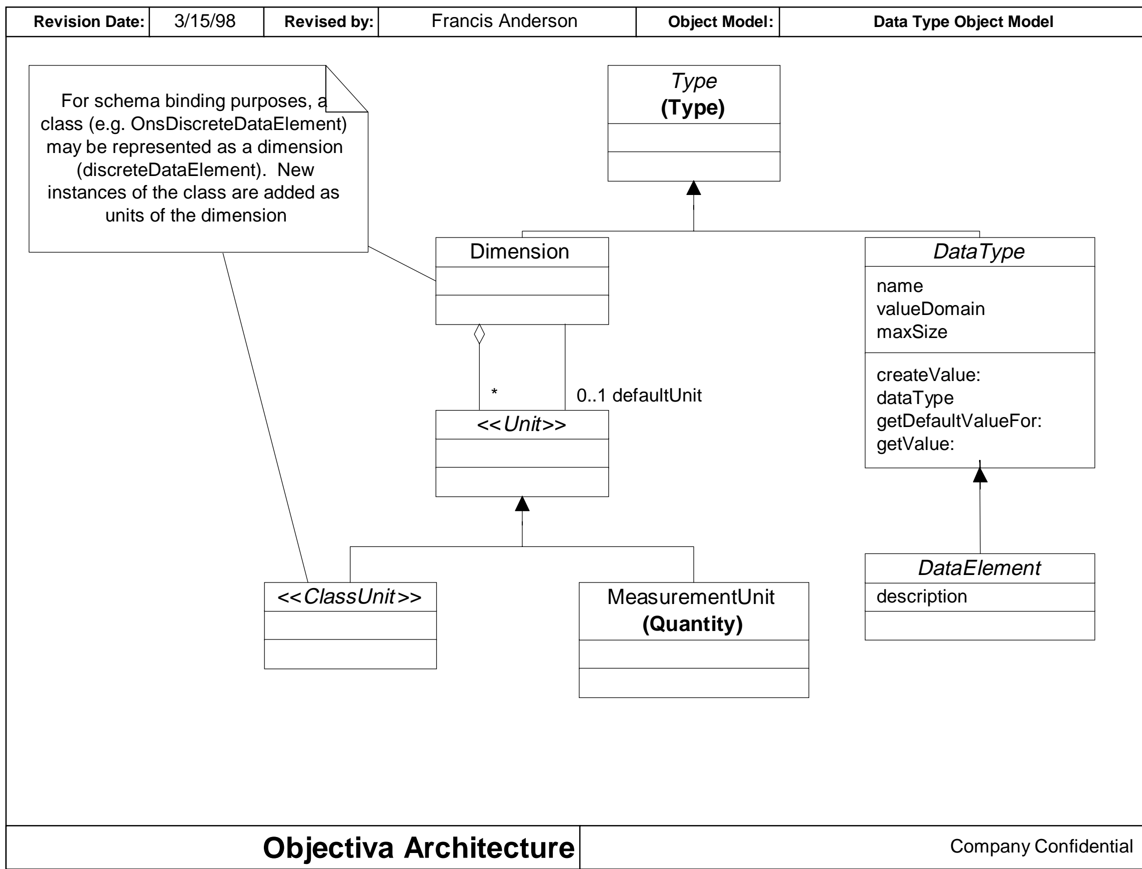

Figure 77: Data Type Object Model

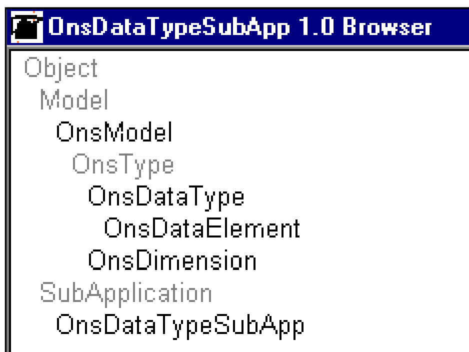

Figure 78: Data Type *ENVY* Subapplication

Framework Development

The Attribute Object Model (Figure 39) tells us that we may have an improvement opportunity. We know from the Entity Package Diagram (Figure 41) that we are going to have a number of different subclasses of ATTRIBUTE. The Attribute Object Model tells us that each subclass of ATTRIBUTE will be reflected somehow in the DATA ELEMENT and DATA VALUE hierarchies. This is the kind of combinatorial subclass proliferation that patterns minimize. In this case, a Strategy for value domain would appear to be in order so Data Value, Attribute and Data Element would share a Continuous Value Domain, rather than each having a continuous sub class. As yet, this work has not been performed. The implementation of Strategy in this context would be an example of refactoring. Let us see how this process proceeds, and whether it is appropriate in this case.

The first question is "Do the hierarchies qualify as parallel?"

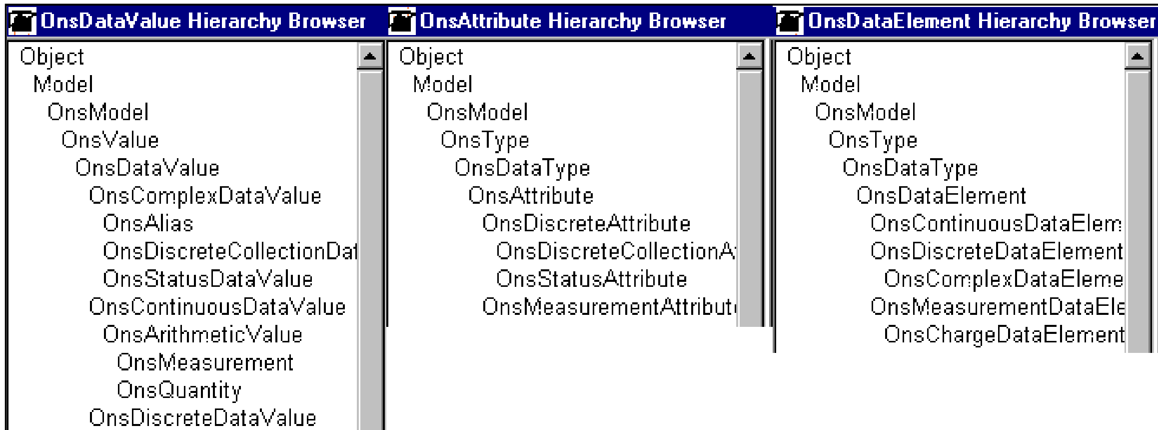

Figure 79: Hierarchy Comparison: DATA VALUE; ATTRIBUTE: DATA ELEMENT

Relationship

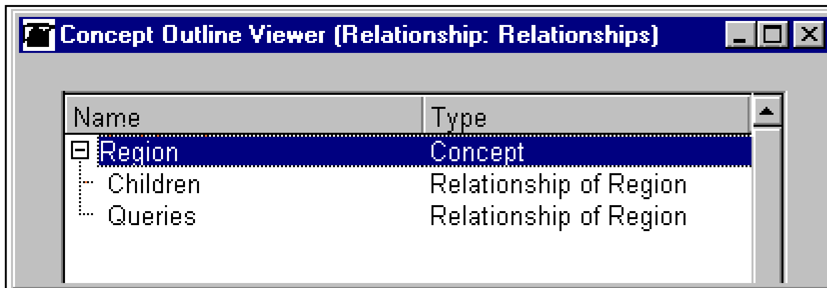

Figure 80: Region Concept Relationships

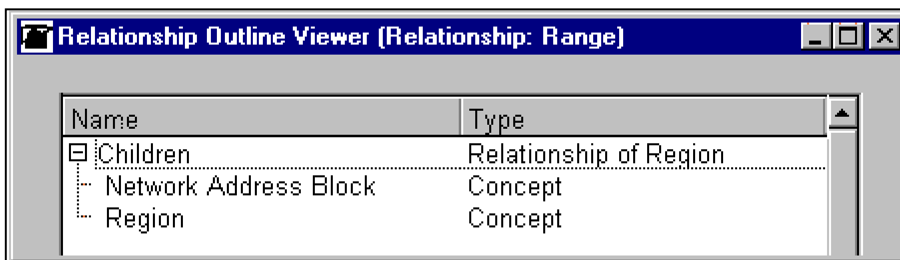

Figure 81: Range of Children Relationship

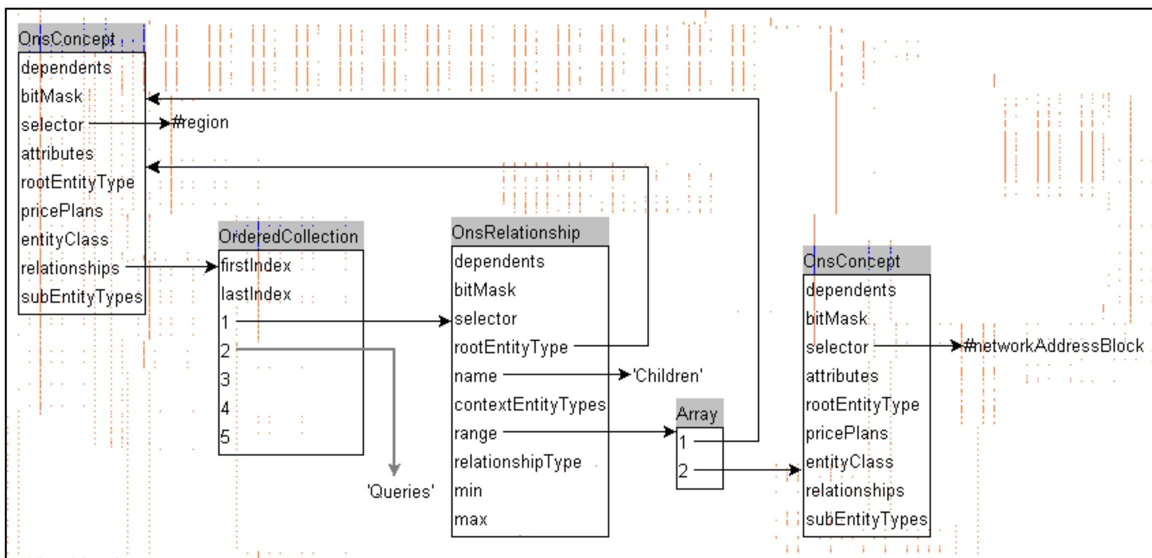

Figure 82: Instance Diagram of Concept Relationship

Root Entity Type Outline Viewer (Relationship: Children)

Name	Type
⊟ Country	Region
⊟ Children	Relationship of Country
⊞ LATA Region	In Children of Country
⊞ NPA Location	In Children of Country
⊞ Province	In Children of Country
⊞ State	In Children of Country
⊟ Price Plans	Relationship of Country
⊞ Tax Price Plan	In Price Plans of Country

Figure 83: Country Relationships

The Country Relationships (Figure 81) states the rules that a Country can have children of LATA Region, NPA Location, Province or State, and may also have a Tax Price Plan.

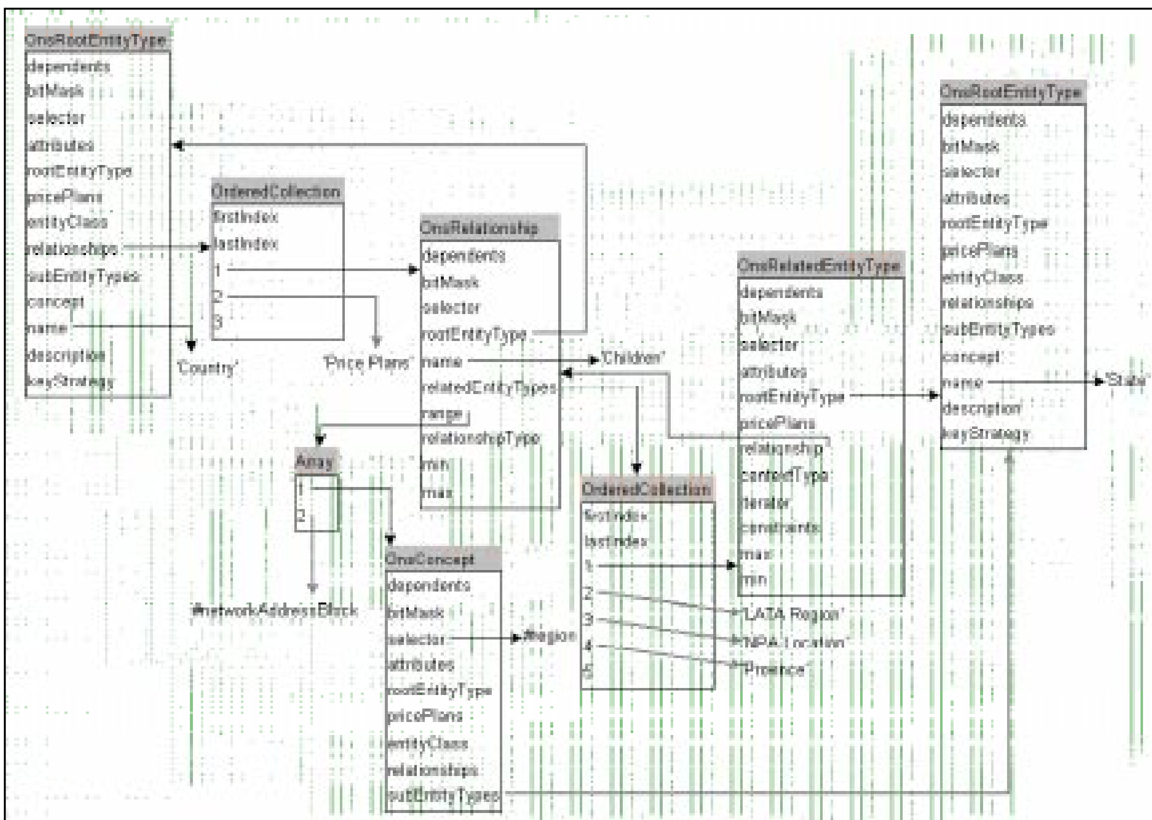

Figure 84: Instance Diagram of Root Entity Type Relationship

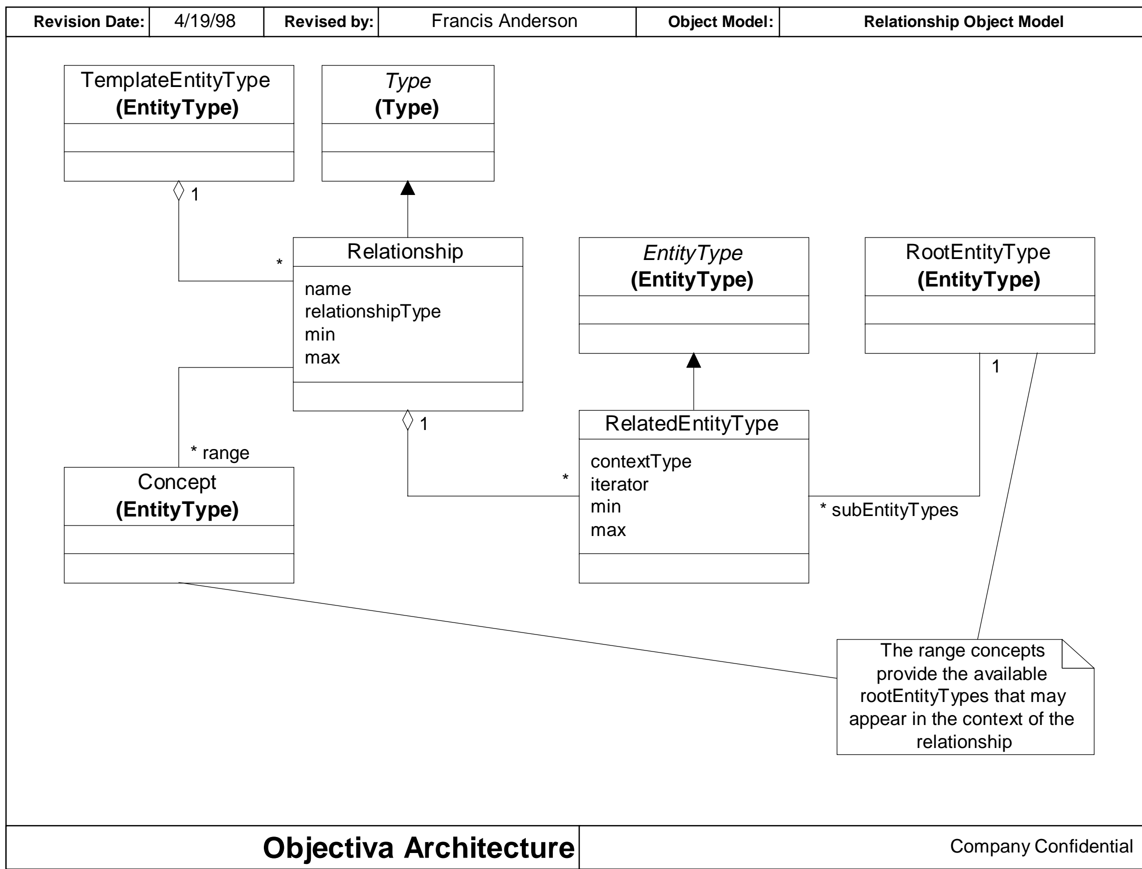

Figure 85: Relationship Object Model

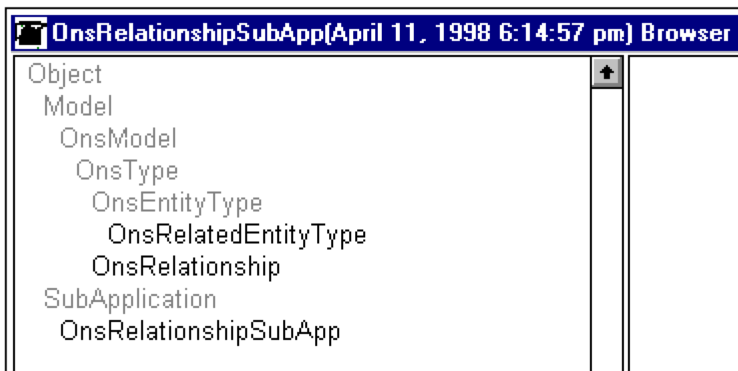

Figure 86: Relationship *ENVY* Subapplication

Use Cases

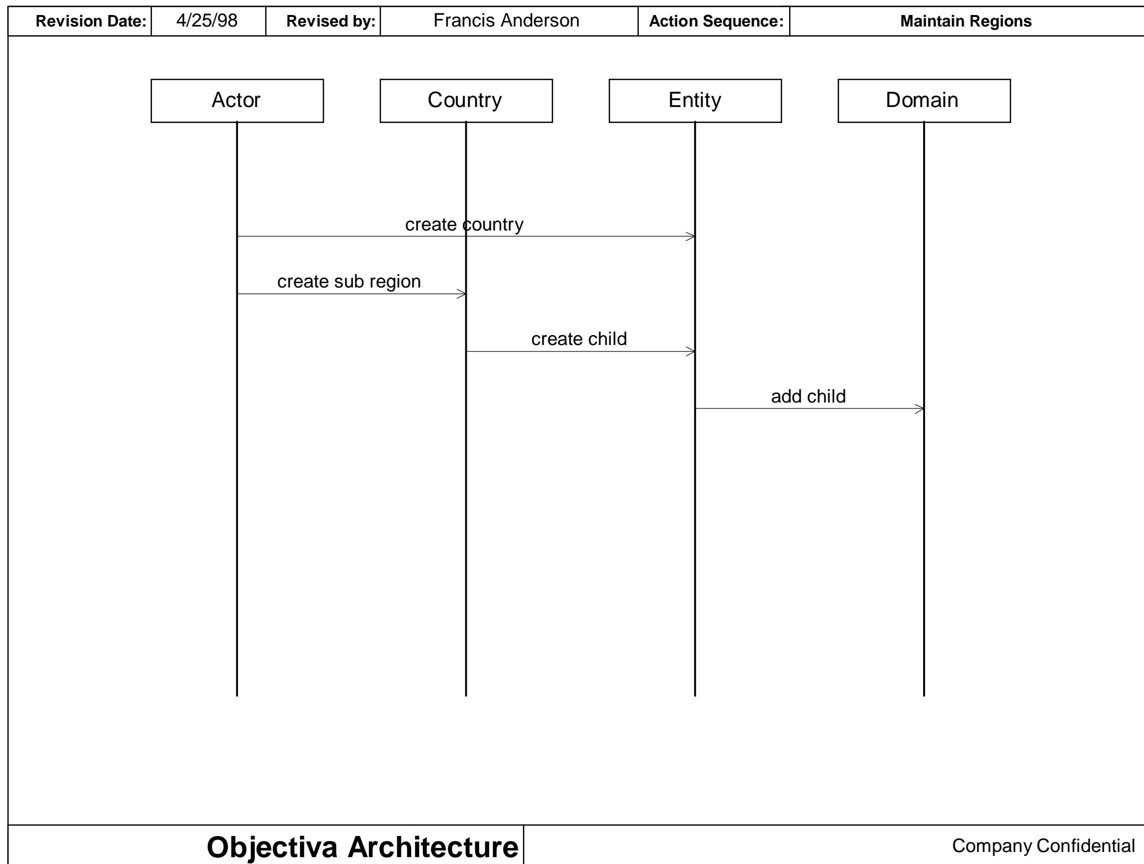

Figure 87: Maintain Regions Action Sequence

```
MAINTAIN REGIONS class>>timeLineDefinition
    "
    (self runCondition: #Default) result
    self regressionTest
    "
    ^self actions: #(       createCountry
                            createSubRegion       )
        result: #country
        businessObject: #Country
        roles: #(regionAdministrator)
```

MAINTAIN REGIONS class>>createCountry

```
"
((self getAction: #createCountry) runCondition: #Default) result
(self getAction: #createCountry) regressionTest
"
^(self sends: #getOrCreateEntityNamed:
        to: #rootEntityType
        with: #(countryName)
        result: #country
        assert:
                [ :s |  | country |
                country := s result.
                country entityType == s rootEntityType
                and: [ country name = s countryName
                and: [ ((OnsDimension getUnitsInDimension: #country)
                        includes: country)
                and: [ (country addressFormats isKindOf: OrderedCollection)
                and: [ (country currency isKindOf: OnsCurrency)
                and: [ (country taxPricePlan isKindOf: OnsPricePlan)
                and: [ country taxPricePlan typeSelector == #taxPricePlan ]]]]]]])
                rootEntityType: #selectedRegion
                        inConcept: 'Region';
                variable: #selectedRegion
                        value: 'Country';
                variable: #countryName
                        value: 'Test Country';
                variable: #abbreviation
                        value: 'TST';
                variable: #chargeBand
                        value: '3';
                variable: #addressFormats
                        value: #('US Street Address');
                yourself
```

MAINTAIN REGIONS class>> createSubRegion

```
"
((self getAction: #createSubRegion) runCondition: #Default) result
(self getAction: #createSubRegion) regressionTest
"
^(self sends: #getOrCreateSubRegionNamed:ofType:
	to: #region
	with: #(	regionName
			regionType	)
	result: #subRegion
	assert:
		[ :s |  | region subRegion |
		subRegion := s result.
		region := s region.
		subRegion entityType == s regionType
			and: [ subRegion parent == region
			and: [ (region children includes: subRegion)
			and: [ subRegion name = s regionName ]]]])
		precondition: #region
			actionSequence: self
			action: #createCountry;
		variable: #regionName
			value: 'Test Child Region';
		variable: #childrenTypes
			from: #region;
		selection: #regionType
			from: #childrenTypes
			on: #selectedRegionType;
		variable: #selectedRegionType
			value: #first;
		variable: #abbreviation
			value: 'TS';
		yourself
```