An Extension and Implementation of Fowler's Observation Analysis Pattern

1. Introduction

Object models almost always start out as monolithic (static) models that can only be adapted to new requirements after mass reorganizations of code and classes. It is common that a set of refactorings has to be made to improve the design in order to get the model ready for the modifications [Roberts97], [Tokuda99]. The developer's domain knowledge drives the transformation from a monolithic model to a flexible one. This transformation takes time, resources, and a well-known cycle of iteration is needed to transform the more basic architecture into a framework [Roberts98].

Fowler [Fowler 95] presents some Analysis Patterns that describe dynamic models can that adapt smoothly to new requirements without necessarily needing the programmer to change the way that objects interact. Analysis Patterns are quite different than Design Patterns. Analysis Patterns describe recurrent problems in term of the elements or concepts that are present in a domain. Analysis Patterns express the domain knowledge acquired based on previous experiences. Analysis Patterns will influence how the code is designed but does not directly deal with implementation details as is done is Design Patterns. We apply Analysis Patterns in the medical domain[1].

This paper present the results of our experience implementing the observation model for the Illinois Department of Public Health. Section 2 presents a brief description of the requirements we found for modeling observations and a very first solution for the problem. Section 3 and 4 describe the two main features we add to the Fowler's observation model such as: *CompositeObservation*, and *Validators*. Section 5 presents the final architecture and some evolution aspects we faced. Section 6 relates future work and summary.

2. An Introduction to the Observation Model

As we were developing several applications to support different medical programs for the state of Illinois, we became aware that they shared very similar requirements. These requirements are related to the ability to rapidly identify and quantify characteristics of patients as well as characteristics of other persons close to the patient (e.g. mother, father, doctor, etc). These characteristics of patients are called observations.

One possible way of building observations could be done by creating a hierarchy of classes describing each kind of observation. This approach properly works when the domain is well known and there is little or no change in the set of observations. Figure 1 shows what the resulting architecture might look like for some basic observations such as height, weight., eye color, hair color, and gender.

Notice here that we associate a set of observations for a Person. Measurements have quantities associated with them for the values. These quantities allow for the possibility of converting units (i.e. 1 inch to 2.54 centimeters). It is easy to describe these classes and it is also easy to extend behavior for these observations by simply programming each class with the needed behavior.

On the other hand, if new specifications for observations are needed or new types of observations are realized, this approach lacks the ability to add or change observations without writing lots of code. This is because each time a new kind of observation is needed, or a current observation needs to change, either a new class has to be built, or the existing class has to be changed to reflect the new requirements. Afterwards a new release of the observation application has to be generated and distributed. Thus, there is a trade off between the simpler solution and the fact that we are sure that the architecture will have to be modified in the near future. Fowler's Analysis Patterns [Fowler96] gave us insight on dealing with the problem we faced.

---

[1] Federico, Joe, and Ralph have worked on the analysis, design, and implementation of the observation model used in several hospital support applications developed by the Illinois Department of Public Health.

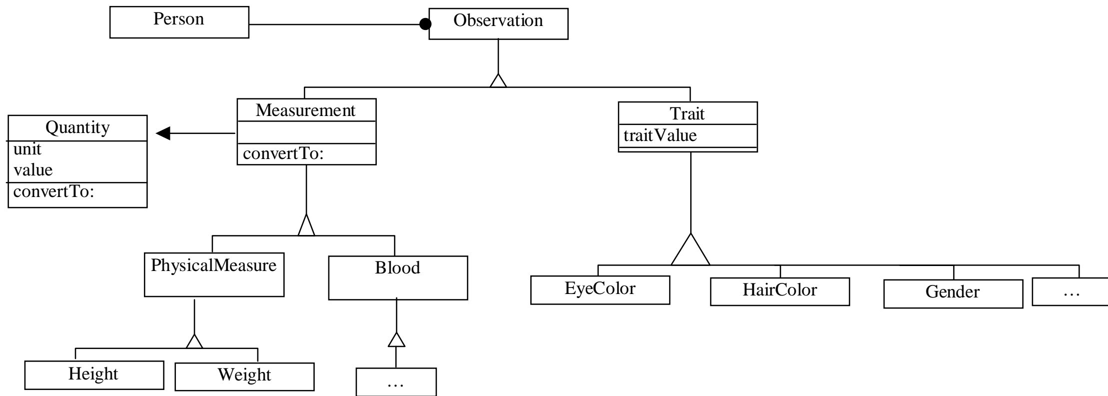

Figure 1 - Static Architecture based on Subclassing

Fowler goes into quite a bit of details describing models for capturing observations. Fowler breaks observations down into two main categories. One is for making observations that contain a finite discrete set of possible values such as eye color or gender. These are called Traits. Another type of observation is for those observations that contain sets of ranged values that can have any numeric value and possibly an appropriate unit. These types of observations are called Measurements. The Observation Model describes very well our initial knowledge of the domain, the users of the systems want to attach different kind of observations to patients such as: hair color, eye color, weight, height, feeding type, blood pressure, blood type, etc.

Figure 2 is a class diagram for implementing the first model presented in Fowler's book (page 43); we adapted Fowler's diagram to match UML and our context and domain classes. Persons are a special case of Parties which can have Observations associated with them. There are two kinds of observations: Measurement and Trait. Measurement represents those observations which are values in a continuos scale (then there is a unit associated to the value) e.g. 5 feet for height, 180 pounds for weight, etc. Trait represents discrete observations of a Party, such as blond for hair color, blue for eye color, AB for blood type, etc. The ObservationType (which is called Phenomenon by Fowler) describes the subject of the associated observation (e.g. height, weight, blood pressure, etc).

The original diagram presented by Fowler includes a class called Classification, which is linked to the Trait. In our model this class is replaced by using literal values[2], thus we do not add this class in the diagram. Fowler calls the ObservationType Phenomenon. We decide to change the name in order to express the relation with the *TypeObject* design pattern [Johnson97].

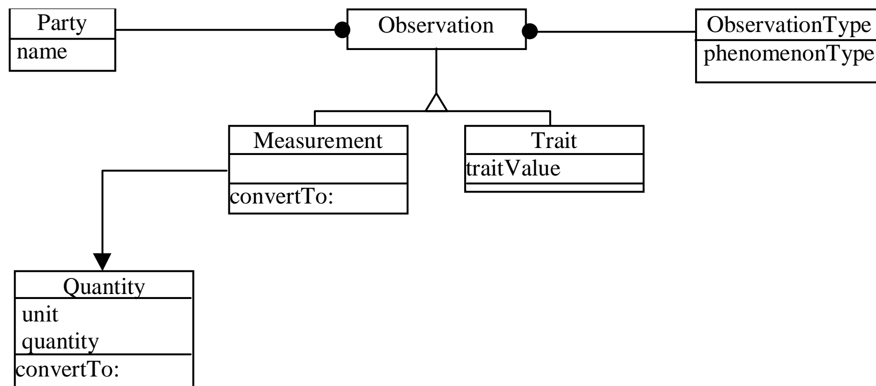

Figure 2 - Class Diagram of the Basic Observation Model

[2] Literal values are instances of the class Symbol as defined in the standard library of Smalltalk classes.

Figure 3 is a simple Instance Diagram with an example of a Party called Smith. Smith has a height observation with a value of 5 feet and an eye color observation with a value of blue. Note that the first Observation holds onto an ObservationType object which has a phenomenonType of #height, and a Quantity object which holds onto the value of 5 for quantity and feet for units.

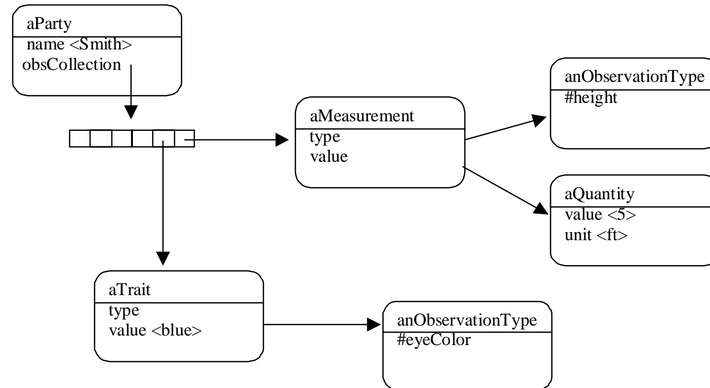

Figure 3 - Instance Diagram of the Basic Observation Model

As it is shown in Figure 3, there is an instance of ObservationType for each different kind of observation. It means that if you want to add a new kind of observation, a new instance of ObservationType has to be created and added to the model (for more details see the *TypeObject* pattern [Johnson98]). This model definitively solves the problem of creating new types of Observations without packaging new versions of our applications. This model was the starting point for our work and an excellent tool for communicating with the domain experts.

3. Composite Observations

The resulting observation model presented in Section 2 can be used to represent a large class of observations seen in many domains. However it is often the case that the observation might be more complicated than the above model can realize. For example, a “cholesterol” observation for a patient is composed by two independent measures such as HDL and LDL. Fowler's model does not consider these multi-value observations. He does extended his basic model to allow for compound units for observations in order to allow for compound units such as area (square yards) and speed (feet per second). It should also be noted that something like cholesterol is actually composed of two separate observations called HDL and LDL. Often, the HDL observation is used independently and the LDL is only considered when the HDL observation value is high.

Another example where composite observations make sense can be seen with observations for blood test results; this is something that we needed to model for our work at IDPH. A blood test observation can be composed of many individual observations of different types. Some may be measurements such as white blood cell count while some of the values may be traits such as blood types. Another example could be an overall observation of a patient's health. This might include many observations such as blood pressure, pulse, vision, reflexes and the like.

Therefore, we extended Fowler's concept by applying the composite pattern [GOF95] to the observation pattern. This allows observations to be composed from other observations. Therefore, a cholesterol observation can be composed of two atomic observations of HDL and LDL. The resulting architecture can be seen in Figure 4. This composition stills allow to capture the compound units for observations that Martin describes, and allowed us to describe more complicated observations. It also makes it easier to use observations such as HDL independently of the cholesterol observation but still be used in the composed observation of cholesterol as a whole.

(***RALPH, THIS SECTION COULD USE A GOOD TOUCH UP ***)

Figure 4 - Class Diagram of Composite Observations

An instance of CompositeObservation can be composed of any kind of observation (*Composite Pattern* [GOF95]). In this way it is possible to define a complex Observation based on basic/atomic ones. For example, an observation about the blood pressure could be described in terms of a composite observation. The "diastolic pressure" and the "systolic pressure" are the components of the "blood pressure" observation (as a composite). Figure 5 shows an instance diagram for this simple example

Figure 5 - Example of Blood Pressure

Note that the instance of CompositeObservation (with type #BloodPressure) and the instances of Measurement are associated with different instances of ObservationType (aMeasurement with #SYSTOLIC and anotherMeasurement with #DIASTOLIC). It makes clear the difference between these two different observations. This model covers our requirements, which asks us to record some information related with each atomic observation as well as the composite one.

4 Validations

At this point we faced a new problem; the relationship between the value of the observation and its type is not yet completely realized by the proposed architecture. In fact, any value or quantity could be assigned to an instance of Observation whether that assignment is valid or not (in terms of the domains rules). It means that these rules have to be recorded in the head of the programmer (in order to validate inputs from the GUI) or in the head of the final user.

One solution to this problem is to extend the architecture to add some part of the responsibility of validation to the ObservationType; afterward the model could describe by itself the validation rules (extracted from the domain). Nguyen and Dillon [Nguyen98] presented a similar idea in "An Alternative Solution to the Observation Pattern Problem."

The proposed architecture allows for different types of observations, "measurements," "traits," and "composed observations" to describe their structure and relevant validation rules. The subject of each observation is defined by one particular instance of the class ObservationType. As previously mentioned, it is possible to extend each type for describing the set of possible valid values associated with them. Some of these valid values could be shared between different types of observations, e.g. any observation quantifying the presence of any illness has three possible values such as YES, NO, UNKNOWN. The resulting architecture for Validators is shown in Figure 6.

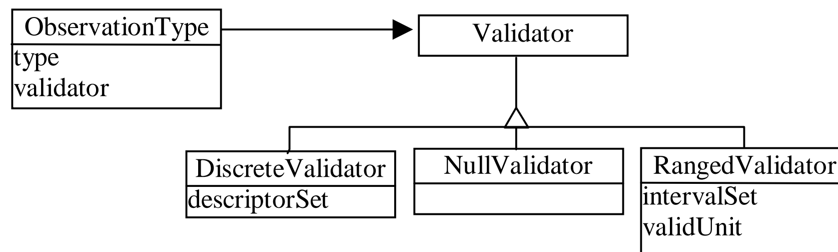

Figure 6 - Architecture for Observation Validation

Each ObservationType is associated with an object that is responsible for determining whether a value is valid or not. A class hierarchy of Validators had been developed and dynamically plugged in at run-time. However, analysis revealed that there are two basic kinds of values that could be the quantifier of an observation; first, a quantity (a single values of a continuous scale) e.g. height, weight, etc.; second, categories (single values, constants) e.g. eye color, hair color, etc. This allows for the ObservationTypes to be extended with types of Validators. This can be done by associating each instance of ObservationType with an appropriate *Strategy* [GOF95] for validating the types.

So in a sense, an observation uses a *TypeObject* to describe its type of observation, which in turn uses the *TypeObject* pattern to describe its Validator. Descriptive data (metadata) can then be used to associate and instantiate the appropriate Validators with the appropriate types of observations. This double use of the *TypeObject* pattern (which we call the *TypeSquare* pattern) is commonly seen in dynamic meta-architectures. Foote and Yoder [Foote98] describe these types of dynamic architectures in more detail.

Each subclass of Validator records a different class of valid elements. In the case of the DiscreteValidator, it knows a collection of valid constants e.g. blue, green, brown for valid eye color. In the case of the RangedValidator, it contains a collection of intervals where a value is valid and unit in which the quantity of the measure is "legally" expressed.

5. Final Architecture

The last two sections describe our main contributions to the original Observation Model presented by Fowler (*CompositeObservations* and *Validators)* along with any associated implementation details. In this section we present an object model of the final architecture along with examples.

Notice that instances of Trait are always associated with an instance of DiscreteValidator and instances of Measurement are always associated with an instance of RangedValidator. Thus, DiscreteValidator and RangedValidator are just describing the difference between the values they are expecting to store in the #observationValue variable. Therefore, rather than having two classes for representing traits and measurements, a single class representing PrimitiveObservations can model the Measurement and Trait classes. Different types of traits and measurements are realized by associating the ObservationTypes to their respective Validator class (DiscreteValidator and RangedValidator).

Figure 7 shows the resulting class diagram for the implementation of observations with Validators. Party is an abstract concept described in Fowler's book for dealing with dynamic organizations and people. Parties have Observations associated with them. The Observations can either be PrimitiveObservations or CompositeObservations. Each Observation has its ObservationType associated with it, which describes the structure of the Observation and hangs on to the validation rules through its relevant Validator.

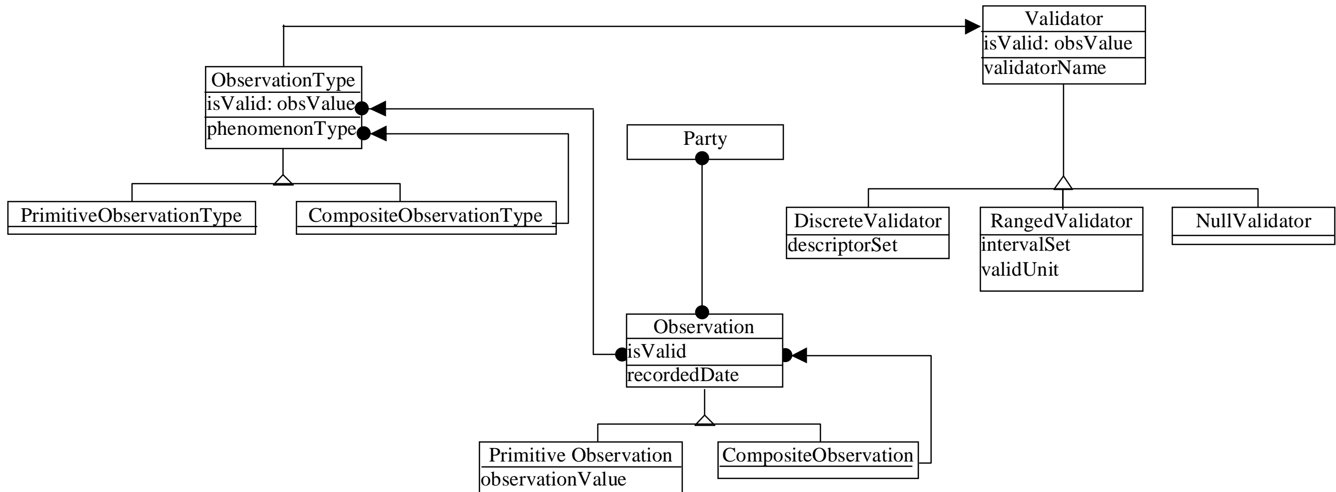

Figure 7 - Class Diagram of the Final Architecture

An instance of PrimitiveObservation represents atomic observations. Instances of CompositeObservations represent compound observations. The TypeObject takes care of the structural properties of the Observation that it is describing. CompositeObservationType is used as a descriptor to create and validate any kind of CompositeObservation. All ObservationTypes have a Validator associated with them. A CompositeObservationType is valid when all of its components is valid.

CompositeObservationTypes could be extended to allow for its value to be derived from a function of all of its components. For example, a cholesterol observation could have a value of OK if its HDL value is within a valid range and its LDL value is within a valid range. It would have a value of NOT-OK if either one of its components (HDL or LDL) values are outside of there valid range. Thus, in this example, the value for cholesterol is similar to a Trait while its components of HDL and LDL are Ranged observations.

Very similarly, CompositeObservationTypes could be *validated* by a more complex validation function on its components. Currently, the architecture applies a simple AND operation to all of its components in order to be valid. It is possible to apply any predicate logic function to any or all of an Observation's components for validating it.

(***RALPH, THE ABOVE FEW PARAGRAPHS COULD USE SOME HELP ***)

Example

There are several "medical programs" currently implemented in each hospital of the State of Illinois. One of those medical programs takes care of the general condition of each new baby. When entering in data for each new baby their parent's demographic data as well as the baby's physical characteristics data is interested into the system. These characteristics are a bunch of Measurements and Traits such as blood type, height, weight, gestational age, feeding type, hepatitis B indication, and so on. After the first 24 hours of life, a number of tests are made using the few samples of the baby's blood (blood-specimen). The result of each test becomes an observation related with the baby.

Sometimes the doctor requests more specific tests for the baby. For example, if galactacenia (GAL) tests positive, the physician will request another test called UDT. Therefore, UDT is a "sub-test" of the GAL

general test. When the doctor requires the UDT test, the result is a "composition" of both results (GAL, and UDT); the doctor can not understand the UDT-result without the GAL-result. Figure 8 shows an instance diagram of this composite observation.

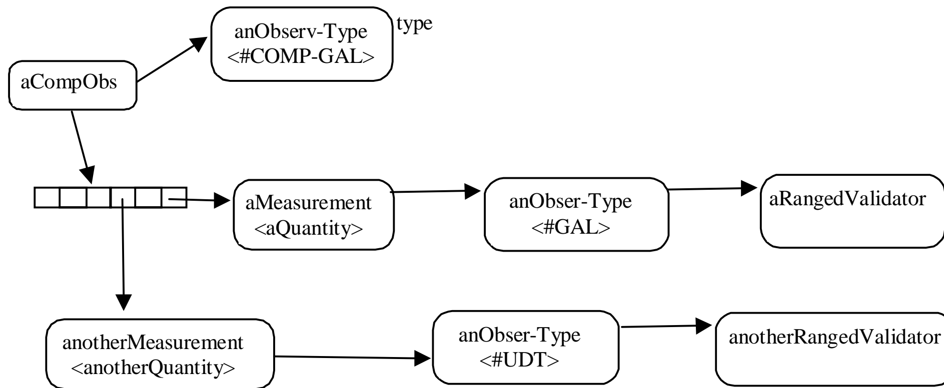

Figure 8 Instance Diagram from the Composite-GAL Result

Figure 8 is a simple example of a composite observation. Only when the UDT-result is the complement of the GAL-result, the overall test makes sense to the doctor. We found more complex and nested instances of this kind of observations when implementing results for general (and complete) test over the blood itself (pressure, type, red cells, blank cells, cholesterol level, etc.). Figure 9 shows an Instance Diagram where both a COMP-GAL result and gestational age are attached to a baby.

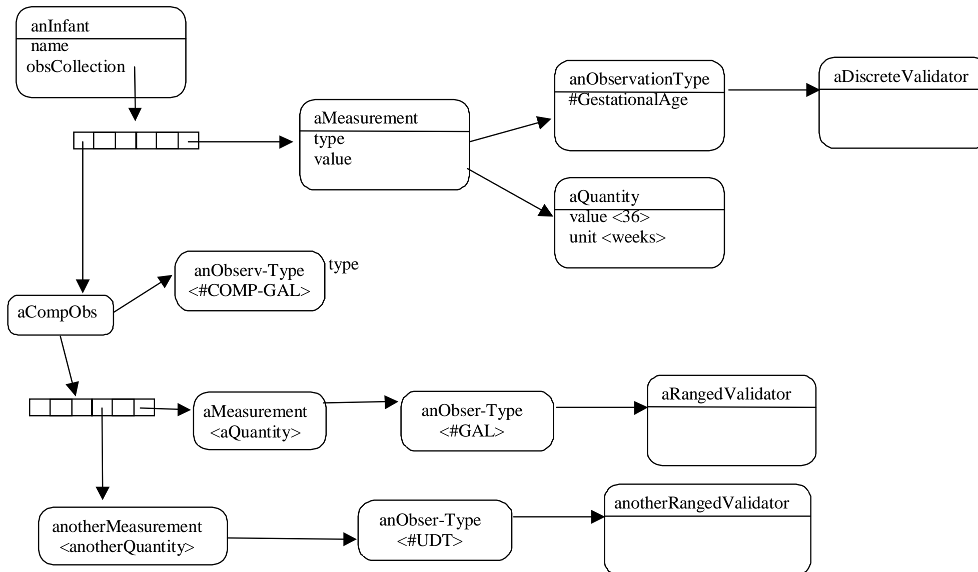

Figure 9 Instance Diagram of Two Observations Attached to a Baby

The Infant instance is actually an instance of Party. The Infant has a collection where it stores each Observation. There is a number of instances involved in getting the information for each baby. On the other hand, the model makes it possible to create new kinds of Observations as well as the validation rules on the fly. The creation and maintenance of the observation's functionality is a responsibility of the system

administrator, who can create, edit or modify the set of Observations directly from the database or using editors and browsers that we specially make for this task.

6. Summary

As we faced the problem of implementing some of Fowler's Analysis Patterns for observations, we found some important aspects that should be considered in order to get successful results. Fowler's basic model is a powerful abstraction that provides a good way for looking at observations, specifically in the medical domain. We found some nice extensions to his patterns that allowed us to deal with some of the issues that we did not see how to immediately solve with his proposed model. There are also some design patterns that greatly assist with the implementation of the analysis patterns. The most important design patterns needed for implementing observations are those that are common to most dynamic systems such as *Type-Object*, *Properties*, and *Strategy*.

Fowler's observation model is rich in variations and extensions, but only part of the whole pattern was needed in order to solve our requirements. One of these possible extensions to the original model, [Fowler96] is presented on page 50 as "Associated Observation". In particular this architecture deals with ways to record the chain of evidence behind a diagnosis. In our case we can build these types of diagnosis by creating composite observations. However, we did not have a need to model diagnosis since there were many legal issues that we did not want to get involved with.

The architecture proposed in this document handle simple observations (Measurement and Trait) and complex observations (CompositeObservation). This architecture is based on the *Composite* design patterns [GOF95]. Because the former composes the last one, in a very general way both are "associated"; but the semantic of the relationship is different from the idea expressed by the "associated observation" architecture. In fact, any class hierarchy and most of the design patterns have associative relationships. The problem is that general associations can be confusing until you give the associations meaning such as class hierarchy or composite. Martin discusses this in detail in his UML book [Fowler 97].

By extending ObservationTypes to describe the structure through the use of the composite pattern and by having the observation types contain their respective Validators, we were able to provide a means for making sure that we get desired values for observations. As we first started to apply Fowler's model, we created Traits and Measurements in order to capture the elements of the domain. We eventually evolved to where we had both kinds of Observations modeled by associating observations with a type of Observation and the rules that validates the values of it. In our final model a Trait is a PrimitiveObservation associated with a DiscreteValidator; while a measurement is a PrimitiveObservation associated with a RangedValidator. This is true because we ensure that any public method in any hierarchy is correctly implemented. Thus, the only difference between a Trait and a Measurement is the type of value they are expecting to store.

After implementing the Validators separately from ObservationTypes, we saw that there are really only a few types of Validators in our domain that are shared. These are YES/NO, POSITIVE/NEGATIVE, and WEIGHT. There may be more but most Validators are not shared. With this in mind, we could change our design to incorporate the Validators into the ObservationTypes. This would lead to five types of Observations; RangedObservationType, DiscreteObservationType, YES/NOObservationType, POS/NEGObservationType, and WeightObservationType. In fact, YES/NOObservationType and POS/NEGObservationType might be subclasses of DiscreteObservationType and WeightObservationType might be a subclass of RangedObservationType. This would remove the use of the second *TypeObject* pattern. This new model is simpler to understand and debug, as you know exactly what your validation method is for each ObservationType. However, if you get a large class structure of ObservationTypes, it might be easier to maintain and add new types based upon the original model.

Fowler talks about using observations in the financial domain. We have seen that we can also use observations on other domains such as inventory systems and licensing systems. For example, in an inventory system, it is easy to dynamically specify facts or events that could modify the valuation of certain item. Those events could be primitive as well as be composite ones.

DON'T FORGET TO TALK MORE ABOUT USING OBSERVATIONS IN OTHER DOMAINS.

7. Acknowledgements

We are grateful to the Illinois Department of Public Health, in particular XXX; the University of Illinois pattern group (To Be Named); and ....

8. References

[Fowler96] M. Fowler. *Analysis Patterns, Reusable Object Models*. Addisson Wesley. 1996

[GOF95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.

[Johnson98] R. Johnson, B. Wolf. "Type Object". *Pattern Languages of Program Design 3*. Addisson Wesley, 1998

[Nguyen98] N. Nguyen, T. Dillon. "An Alternative Solution to the Observation Pattern Problem". Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98.

[Foote98] B. Foote, J. Yoder. "Metadata and Active Object Models". Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98.

[Roberts97] D. Roberts, J.Brant, R. Johnson. "A Refactoring Tool for Smalltalk". Theory and Practice of Object Systems, Vol 3 Number 4, 1997.

[Roberts98] D. Roberts, R. Johnson. "Patterns for Evolving Frameworks". *Pattern Languages of Program Design 3*. Addisson Wesley, 1998

[Tokuda99] L. Tokuda, D. Batory. "Evolving Object-Oriented Architectures with Refactorings".