Title: Teaching Reflective Architectures

Authors:
Dr. Sara Stoecklin, Florida State University, stoeckli@cs.fsu.edu, 850-425-5266
Judy Mullins, University of Missouri-Kansas City, mullinsj@umkc.edu, 816-235-1194

Abstract

Recently metalevel or reflective architectures have received a noted amount of interest in the object-oriented community. This poster presents a method to integrate the teaching of these architectures to computer science courses or industry related training. The method uses Analysis Patterns defined by Fowler [Fow97] and a common software function of validating input from a user. In this poster we use the input validation to illustrate to students how reflective architectures work.

Description of Poster

This poster contains an example of a problem assignment for students to learn to use reflective architectures. It includes metadata information requirements, UML description, code, and narratives describing the process for defining and using the architectures.

Educational Use

Educators and trainers should be able to take the literature from this poster session and build a framework for teaching reflective architectures to students in advanced programming or software engineering courses. This should prove beneficial in both teaching reflective architectures and showing how reflection is used in application development.

Introduction

Recently, metalevel or reflective architectures have received a noted amount of interest in the object-oriented community. Practitioners in industry are successfully using these architectures on real-world problems. Application builders who have built systems using these architectures have experienced reduced development time, ease of development and low maintenance costs [Yoe97]. The essential feature of these architectures is the ability for them to adapt to changes that occur in the requirements of the system, thus some name these adaptive architectures. These architectures use polymorphism and the reflection pattern to allow dynamic linkage to needed methods and classes.

This poster presents a method to integrate the teaching of these architectures in computer science courses or industry related training courses. The method uses Analysis Patterns defined by Fowler [Fow97] and a common software function of validating input from a user. In this poster we use input validation to illustrate to students how reflective architectures work. Below is a section that defines the analysis patterns used in the presentation of the input validations. Subsequently there is a description of the input validation function. Finally, a clear description of the scenario for teaching the use of reflective architectures in this function is described.
ANALYSIS PATTERNS

The most popular current patterns, design patterns, deal with those patterns useful in object-oriented design and object oriented programming [Gam95], [Cop95], [Gra98a], [Gra98b]. Design patterns represent reusable software structures needed for implementation. In contrast, Fowler’s Analysis Patterns represent reusable abstractions needed in the elicitation and representation of the software requirements, thus analysis patterns. Analysis patterns represent conceptual domain structures denoting the model of a business system domain rather than the design of computer programs. While analysis patterns may not deal directly with the details of implementation, they do influence how code is designed.

During these analysis activities particular analysis abstractions emerge. These abstractions, called analysis patterns, represent reusable patterns for subsequent analysis efforts in various domains. As a simple example, the first analysis pattern described by Fowler is the Party pattern. A Party, according to Fowler, is an abstraction to define persons or organizations. He models a Party class with sub-classes of Person and Organization. The Party could have different roles. For example, a person could be an employer or an employee, a doctor, a mother, etc. An organization could be a business entity, a shelter, a hospital, etc. Figure 1 depicts the class representation of the Party pattern.

![Party Pattern](image)

Another analysis pattern described by Fowler is the observation pattern used to model information about the real world generally representing attributes observed about a party. Observations play an important role in information systems, since these observations are usually stored in databases and re-hashed to form statistical analysis of data. Examples of observations are a person’s eye color, hair color, weight or height. Other examples include a test score of Johnny Doe on December 4, 1999 or Jane Doe’s telephone number. The observation pattern is an abstraction that describes the quantification about a given attribute type, called a phenomenon, related to a Party. As an example, a test score type of attribute or, as labeled by Fowler, a test score phenomenon, has a quantification of 64,
which is related to a person, Johnny Doe, i.e. Johnny Doe made a score 64 on a particular test.

From the work of Yoder [Yod97], we realize that not only is an observation in the pattern but an observation type is needed to describe the subject of the associated observation. Figure 2 below depicts the relationship between a party and a portion of the observation pattern, including the observation and observation type. The observation type allows modeling of the observation phenomenon as a type. This pattern allows creation of a phenomenon such as test score, without creating a different class for each phenomenon needed within a system. One instance of the party class, such as Johnny Doe, may have multiple instances of observations, and each of those observations is defined as a specific observation type. As an example, John Doe has an observation of blue of observation type called eye color, as well as an observation of 64 of observation type test score.

Figure 2: Party associated with an Observation

INPUT VALIDATION

As users input information into the computer, a robust software system validates that the data entered meets requirements specified. Integers, entered as strings in text fields, require validations by first assuring they are composed of the characters 0-9, second by assuring that the value is indeed storable as an integer and that the value of the entered data falls within the range defined by a business rule. A shoe size may need to be validated as being within a range of values such as 0 to 17. A salary within an organization, represented as a float, may have a business rule that requires the salary to be more than $1.00 and less than $100,000.00 per year. Structured systems have these input validation functions written in each program that allows entry of the field. In object-oriented systems, one might call a class that contains the validation method for the entered input. That routine os used by each program that allows entry of the field. Using Fowler’s analysis patterns, validation for this salary observation is done using the Validator pattern.

The Validator pattern is used in collaboration with the Observation pattern to validate observations. This architecture allows different types of observation to be associated with their relevant application and business. Therefore, the observation types extended with a type of validator is associated with the instance of the observation type using a validator strategy [Gam95]. The Validator pattern is an abstraction that models the procedures for validation of different types of observations using three different validators. The Validator class, shown in Figure 3 below, depicts the three validators, namely the discrete validator, the range validator and the null validator.
When using the pattern, all observations are validated. Therefore there exists a need for a null validator for those fields which require no validation. The null validator is implemented using the Null Object pattern[Gra98a].

The Discrete validator validates values, such as those items found in a typical code value table. A common example of discrete validation is eye color in which valid values include a set with members of blue, green, hazel, brown, and black. Another common example is a two-member set representing sex as the discrete values of male or female. A user entering data in a field with a discrete validator would view a table of valid values and select one of those values. This is possible if the number of values in the code table is not too large. Large code tables often require the user to key in a few characters before a table of valid values is displayed. Observations in which the valid values is extremely large may not allow the code values to be displayed, but instead require the user to key in the entire value before validation. This is the scenario addressed by the discrete validation described in this paper.

The Range validator is used to validate an observation expected to be within a valid range of values according to a particular business rule. The salary, which must fall within the range of 1.00 to 100,000.00 per year, is an example of such an observation. A range validation routine requires a minimum and maximum value to define the valid range. The Validator Pattern classes are shown in Figure 4 each with a method isValid. While the Discrete, Range and Null validators are defined here, many other validation routines could be included.

![Validator Pattern](image)

---

My experience in teaching patterns these involved the following items:

1. Preparing a short lecture to introduce the ideas of the Party, Observation and Validator Analysis Patterns.
2. Assigning a program that had input of variables and required the validation of those variables using the Validator pattern. As an example, students would be asked to enter fields (with or without a GUI) such as name, sex, date of birth, shoe size, and salary.
3. Sending XML files that contained the metadata about the fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
<th>Validator</th>
<th>Min</th>
<th>Max</th>
<th>Discrete</th>
<th>Error Msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>Null</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sex</td>
<td>char</td>
<td>Discrete</td>
<td>M,F</td>
<td>F,M</td>
<td>Sex must be F or M</td>
<td></td>
</tr>
</tbody>
</table>
The students build a class called `MetadataClass` with this format and then build the needed instances for each of the datatypes as needed, obtaining the information from the metadata.

The typical scenario for programmers building this program is as follows:

b. Allow entry of the field
c. Look up the name of the field in the metadata provided in the XML.
d. Build an instance containing the metadata information.
e. Use reflection to make an instance of the needed validation class
f. Call the method `isValid()`, sending the field.

4. The students will have to write the method in the range validator to validate the salary as a float. They will also have to write the routine to validate date, including such items as valid month, day and year beyond today's date.

The poster session will contain the UML description of the patterns used to build the dynamic linkage between the components and the metadata, Java code implementation as shown in Figure 4. It also gives detailed scenarios for both the pattern developer and application builders using the patterns. It also presents the code for the validator with the needed integer function for educators and trainers to use in their classes.

![Figure 4: Reusable GUI Component and the Validator Pattern](image)


This work is supported by the U.S. Army Research Office grant number DAAd19-01-1-0502.