AOM Domain-Specific Validations

ATZMON HEN-TOV, Pontis Ltd.
DAVID H. LORENZ, Open University of Israel
LIOR SCHACHTER, Open University of Israel
REBECCA WIRFS-BROCK, Wirfs-Brock Associates, Inc.
JOSEPH W. YODER, The Refactory, Inc.

An Adaptive Object-Model (AOM) system represents the domain as metadata. This metadata is interpreted at run time to construct an object model. One important characteristic of any robust AOM system is the ability to validate the correctness of domain entities, their properties and property values, and relationships. This paper presents a pattern for domain-specific validation of user-defined domain entities, attributes and relationships. This pattern describes how AOM validation solutions can start simply with built-in validations and how they can grow and evolve as needed.

Categories and Subject Descriptors: D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.11 [**Software Architectures**]: Patterns

General Terms: Design, Reliability

Additional Key Words and Phrases: Adaptive Object Models (AOM), Validation

ACM Reference Format:
Hen-Tov, A., Lorenz D. H., Schachter L., Wirfs-Brock, R., and Yoder, J. W., 2013. AOM Domain-Specific Validations. In *Proceedings of the 20th Conference on Pattern Languages of Programs* (PLoP'13), Monticello, Illinois, USA (October 2013), 13 pages.

1. INTRODUCTION

An Adaptive Object-Model (AOM) architecture represents user-defined domain entities, attributes, relationships and behavior as metadata [AOM, FY98, YBJ01]. In an AOM system, the domain model is constructed at run time by interpreting externally stored definitions (metadata). One important aspect of any robust AOM system is the ability to validate the consistency of domain entities, their properties, property values, and relationships. In AOM architectures this is complicated by the fact that domain experts also need support for changing the object model (or the metadata) to reflect changes in the domain.

The contribution of this paper is the presentation of a pattern for domain specific validations, which can initially be implemented simply to provide basic, built-in validation support and can later be grown in sophistication as needed. Incorporating this pattern in the early stages of developing an AOM framework and embedding it in the process of evolving the system can prevent the inconsistencies and execution flaws stemming from the relative easiness, introduced by the AOM architecture, in adapting structure and behavior in production.

The pattern presented in this paper deals with a fundamental concern of AOM systems – keeping system consistency and stability while evolving it on-site with AOM engineers. AOM engineers are knowledgeable domain experts with skills sufficient to use AOM-specific DSLs or tools to extend or customize the system. This pattern is intended for those who are building AOM systems with extensible validation mechanisms.

2. PATTERN: DOMAIN-SPECIFIC VALIDATIONS

2.1 Context

You are developing an application using the AOM architectural style to support changing the application model dynamically, but you also want to keep your application model consistent. Your AOM engineers have the know-how to define validations for the application model as they define it.

2.2 Problem

How do you provide the AOM engineers with means to express domain consistency rules as part of the declarative AOM model definition.

2.3 Forces

Usability, extensibility, performance and development effort trade-offs need to be considered:

- *Ease of use:* The AOM user needs to define their own custom validations for domain entities and their properties in addition to basic validations developers may provide. Very few domain experts can actually write Object Constraint Language (OCL) to define rules that apply to Unified Modeling Language (UML) based models. Can you provide support for users to define their own custom validations of their domain model without requiring them to use formal constraint languages?
- *Flexibility:* It is hard to anticipate future validation needs. A simplistic, non-extensible, solution will render the AOM user incapable of defining new validations. Can you provide simple yet extensible validations to start that can grow in sophistication as needed?
- *Evolving validation needs:* Validation needs vary between different domains. It is better to develop the validation system progressively rather than developing a full-blown validation framework up front as part of your AOM application. How can you grow a validation framework?
- *Model evolution:* Changes in one Entity instance may cause other, dependent, Entity instances to become invalid [HNS10]. How can you support evolving the model and its validations, while guaranteeing consistency as it evolves?

2.4 Solution

A variety of solutions are possible, depending upon the domain-specific requirements. These could range from something as simple as adding basic type validations and validator classes to creating a full-blown domain-specific rule language for complicated business rules [YJR02]. What is important is to not over-design a solution. The following outlines the solution space. Following are a list of several potential solutions that can be applied to perform AOM model validations, ordered in terms of increasing sophistication and difficulty. An extended validation framework or advanced validations can be added to basic validations, as needed.

- Basic validation
  - Implement basic type validations in `EntityType` and `PropertyType` classes.
  - Create simple `Validators` implemented as part of your AOM `Entity` and/or `Property` framework.
- Extended validation framework [Jon99]
  - Implement domain specific validations in domain specific subclasses of `Entity` and `Property` (these validations will hereafter be referred to as built-in validations).
  - Separate these validations into their own classes for easier composition.
  - Allow the AOM user to configure built-in validations when defining new entity types.
- Advanced validations
  - Allow the AOM user to add custom validation logic via hooks [AHS11].
  - Create a base rule language for building and composing the validators.
  - Incorporate a rules engine into the AOM for validation.

Declaring AOM entities and properties using `TypeSquare` constrains the legal types of property values for any given `EntityType`. These simple type constraints can be augmented by adding declarations to the `PropertyType` that describe required properties, cardinality and simple syntactic validations such as length of Property value (see mandatory attribute in Figure 1). For example, consider an `EntityType` for Employee which requires first name and last name whereas the middle name is optional. Additionally there can be validation rules that state that the length of first name and last name are constrained to each be less than thirty characters in length.

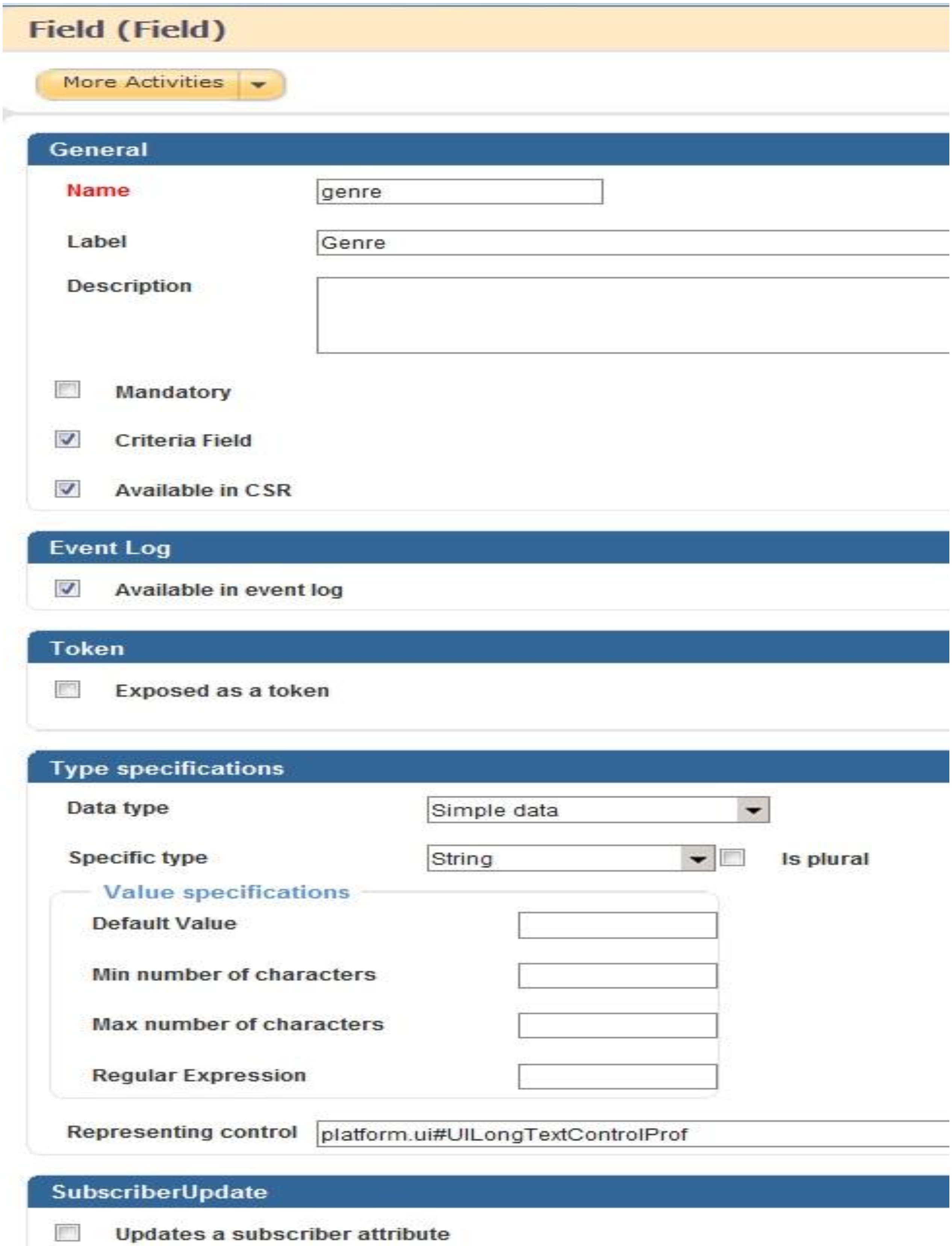

Figure 1 – User Interface for defining a field. Several prebuilt validations allow the user to define legal values for the field

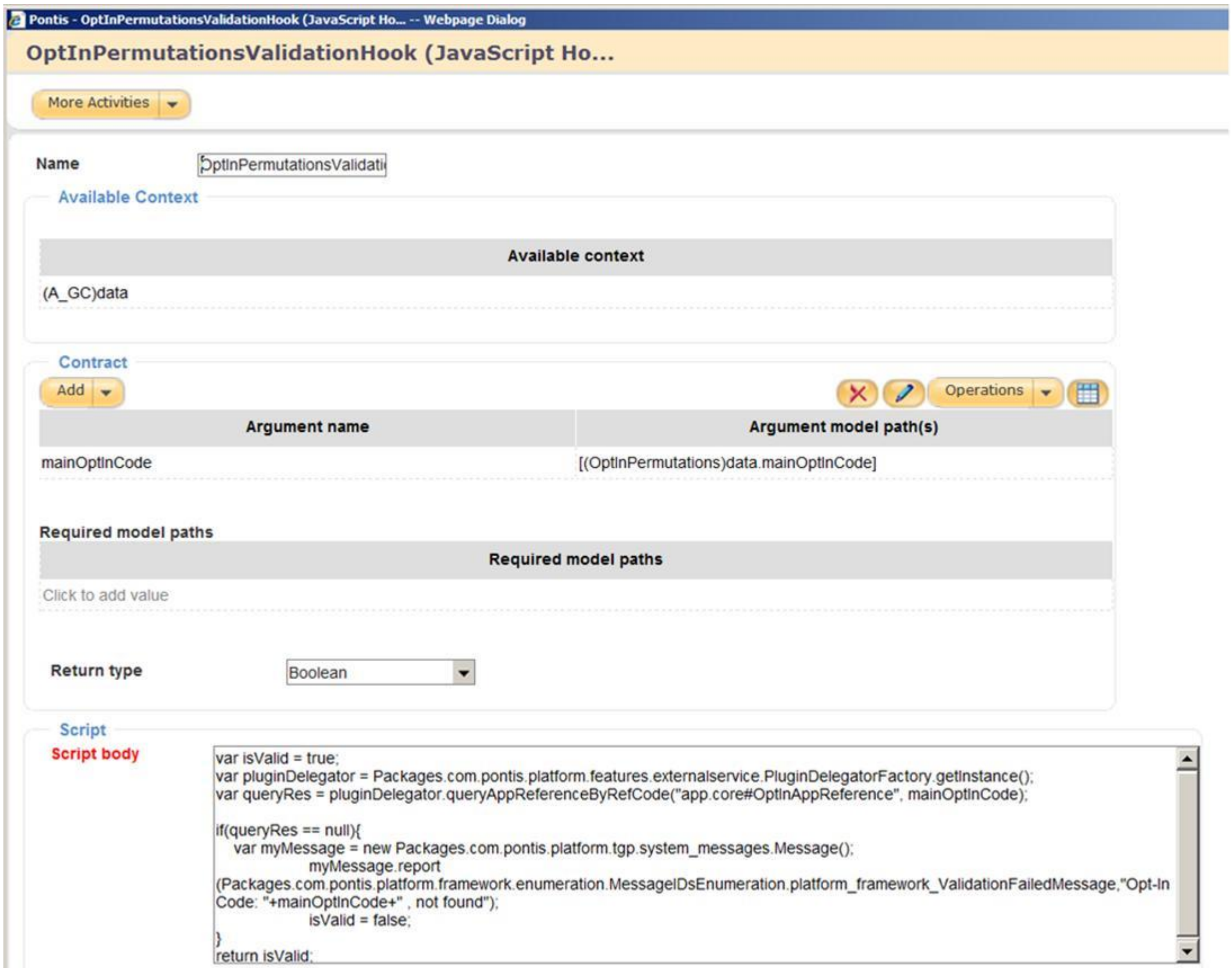

Figure 2 – An Editor in the PONTIS system for defining custom script-based validations

Ultimately, a more flexible solution separates the validations into their own classes. This allows for composition of validators and possibly the creation of a validation language that can be reused throughout the system [Fow97]. These often lead to using the interpreter pattern [GHJ95] for a little rule language or mini DSL [Fow10]. Sometimes a rule engine is used for managing these validations and the validations can go across many different entities with dependencies across the entities and their properties. Rules can usually be broken down into 1) Constraints on values, relationships, state change, 2) Functional in nature, 3) Workflow, and 4) Event based.

When a little language evolves it is common to develop an editor or Visual Language for defining the entities and rules. Figure 21 is an example of a property validator implemented using additional meta-data on the property definition. `Mandatory` check box, `Min/Max number of characters` and `Regular expression` fields are all used by the custom property validator (in this example `StringValidator`) to validate instances. Figure 2 depicts an Entity validator implemented using the EVOLUTION RESILIENT SCRIPT pattern [HLN10]. When defining new EntityType, in this case `OptinPermutation`, the validation logic is expressed in Java-Script code. It will be invoked by the AOM framework, on an `OptinPermutation instance` before persisting it to the database.

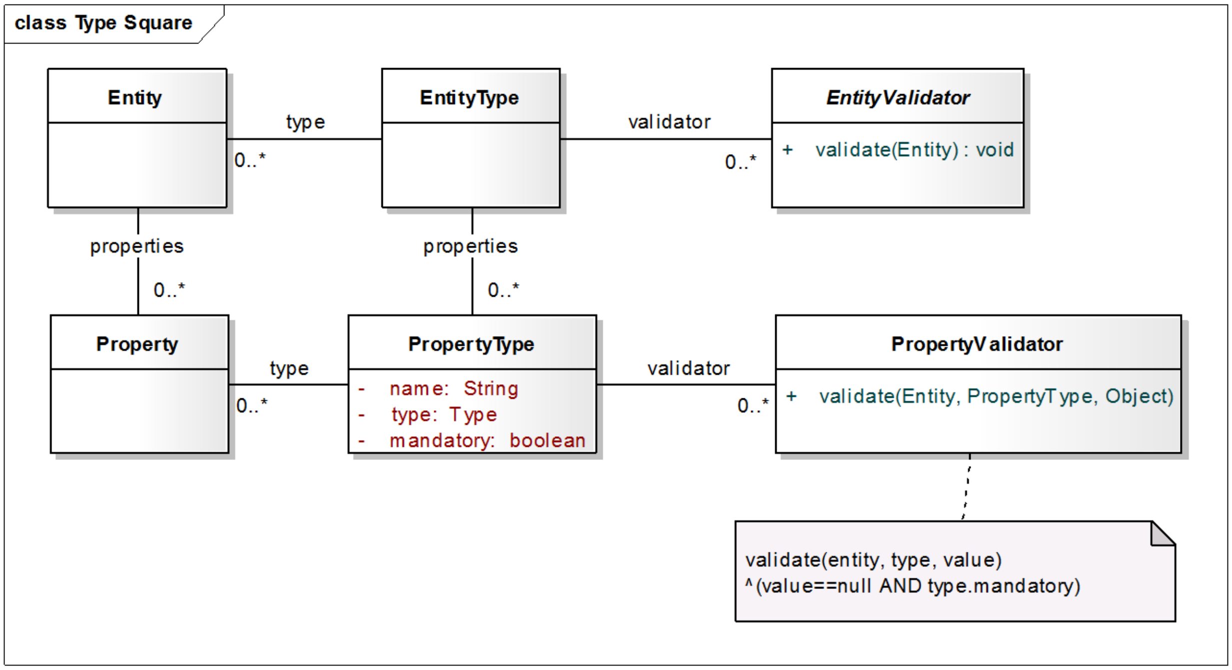

Figure 3 – Class Diagram of Type-Square with Validators

Choosing the right type of validation is one of the hard parts of developing AOM's. Design is always about tradeoffs and it is important to let the requirements and user scenarios drive the design. One size does not fit all and it is often the case where a couple of the above mentioned validation techniques might be needed in combination.

When simple type validations only need simple rules, then consider the basic validation techniques mentioned above. If the rules become more complex and are definitely evolving based upon the product or client, then consider techniques mentioned in Extended Validation Framework. If the rules really need to be flexible because you change them often, then using advanced techniques such as dynamic hooks or evolving to a domain specific language are good solutions for consideration.

2.5 Implementation

Figure 3 outlines the class diagram for two types of validators. `EntityValidator` is associated with `EntityType` and is responsible for validating an Entity instance. It performs cross-property validations and other specific business logic validations. Adding a new validator for an `Entity` is accomplished by sub-classing `EntityValidator` and attaching the validator to the domain classes, e.g. a `PersonValidator` validates that every Person entity with a driving license is above 17 years old. A `PropertyValidator` is associated by `PropertyType` and is responsible for validating a specific property value. Figure 4 exemplifies how more specific types (`StringPropertyType`, `NumericPropertyType`) can be created with additional metadata (e.g., a `regExp`, or a `minValue` and `maxValue`) by subclassing `PropertyType`. The corresponding validators (`StringPropertyValidator`, `NumericPropertyValidator`) use this additional metadata to validate the property's value.

As illustrated in Figure 5, validations are invoked upon saving an entity instance. This sequence shows how, basic, extended and advanced validations can be invoked in succession. To start a validation sequence, the method validate is called on the entity type, passing in the entity instance itself. The following outlines the interactions during the validation process:

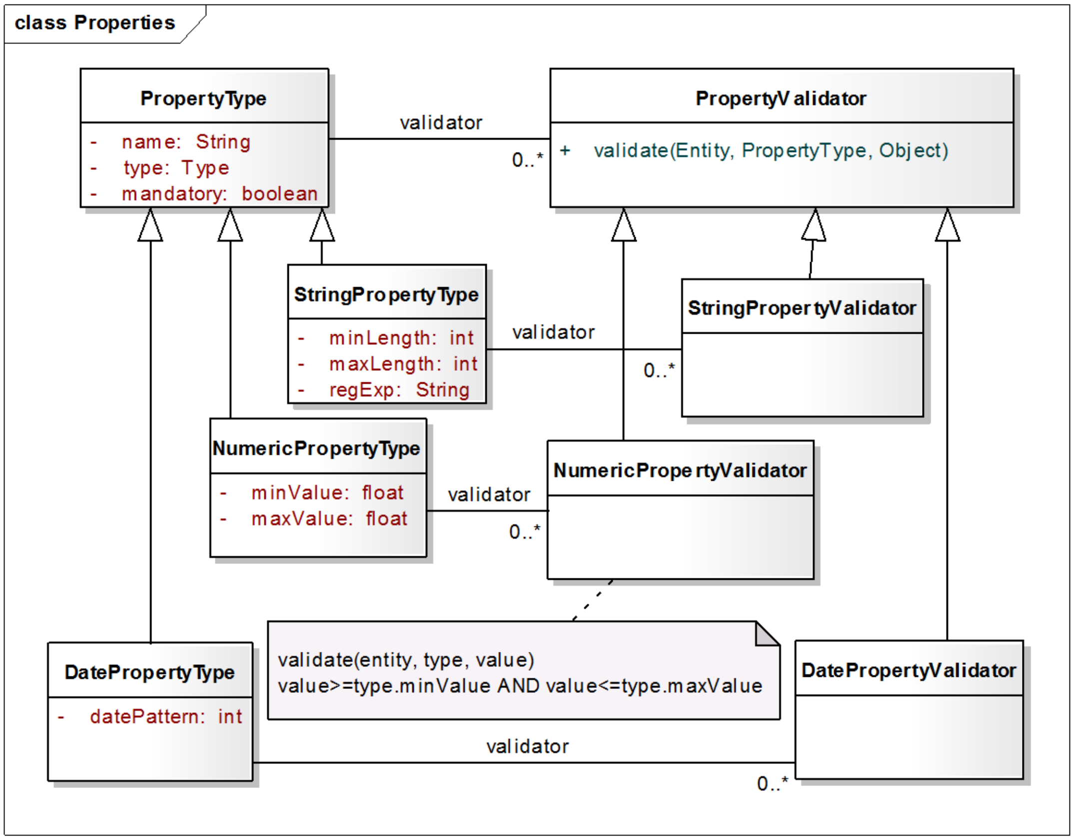

Figure 4 – Class diagram of `PropertyType` class hierarchy and validators

1. The `EntityType` iterates over its properties and invokes the `PropertyType` validator. The `PropertyType` invokes the `PropertyValidator` attached to it passing the original entity instance (the root validation context), the actual property value, and the `PropertyType` itself.
2. The `PropertyValidator` first checks if the relation is a composition type. In that case it invokes the PropertyValue's `validate()` method and so on... If a relation is a reference to another entity (rather than a composition), validation doesn't proceed to that referenced entity.
3. The `EntityType` invokes the `EntityValidator` to perform Entity level validations. Domain-specific entity level validations are implemented in subclasses of `PropertyValidator` and `EntityValidator`.
4. The `EntityType` then iterates over its custom validations, if any are present, and invokes them `ValidationHook`.
5. The `EntityType` then iterates over its validation hooks, if any are present, and invokes them.
6. All errors are gathered in a shared context and returned to the application layer to process according to the error handling policy. The application can require the user to correct the errors before the save proceeds or provide other means to manage entity inconsistencies [HNS10].

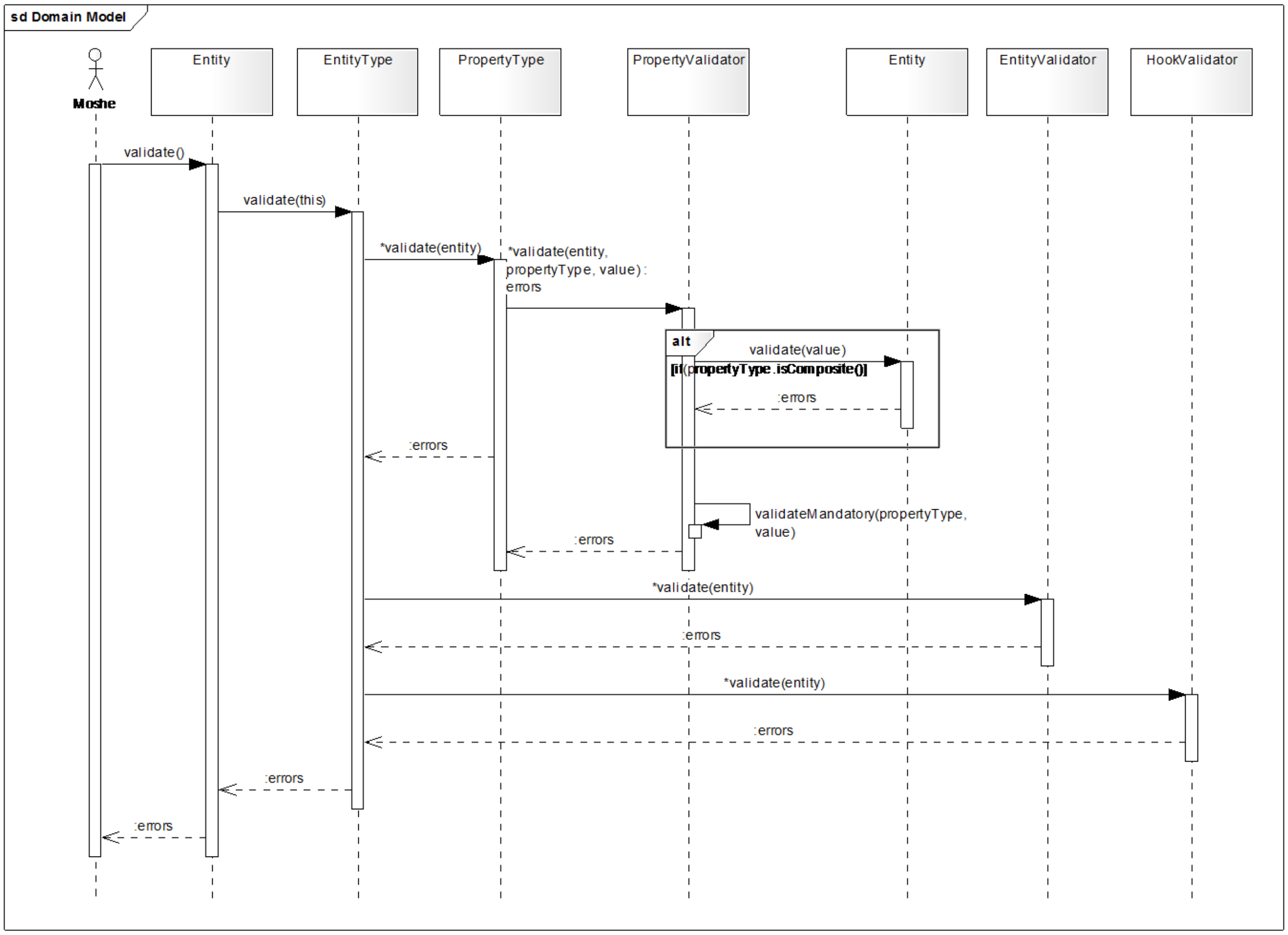

Figure 5 – Sequence diagram of Validators execution flow

Figure 5 shows one example of where validations are performed, upon saving objects. But there are several places where validation may be performed: on object creation, whenever a dynamic object changes state or relationships. There are even other well-known techniques that can be used to validate domain objects such as constraint languages such as OCL, or via XSLTs during object loading, importing or exporting.

Ultimately, it is important to provide a means for an AOM engineer to configure and extend any built-in validations when defining new entity types. One alternative is to evolve the validators into a little DSL that can be used express domain specific validations using declarative composition. A second alternative is to use a rule-engine which through composition and simple formulas supports the definition of complex, domain specific validations. When the domain is narrow enough and validators can be reused across `EntityType` definitions then a small DSL solution will probably be the preferred choice since it provides a more robust and domain-specific solution. A rule engine provides more generic solution which is appropriate when a variety of validations is needed.

2.6 Examples

For an AOM system developed for the Illinois Department of Public Health, the Refactory implemented a variation of the OBSERVATION pattern [Fow97]. Ultimately for the model to handle basic validations, the model was extended so that `ObservationTypes` were responsible for validations. The model described the validation rules. The architecture allowed for different types of observations, "measurements" and "traits" to describe their structure and relevant validation rules. The subject of each observation was defined by one particular instance of the class `ObservationType`. It was possible to extend each type and describe the set of possible valid values associated with them. Some validation values were shared between different types of observations, e.g. any observation quantifying the presence of an illness had three possible values such as YES,

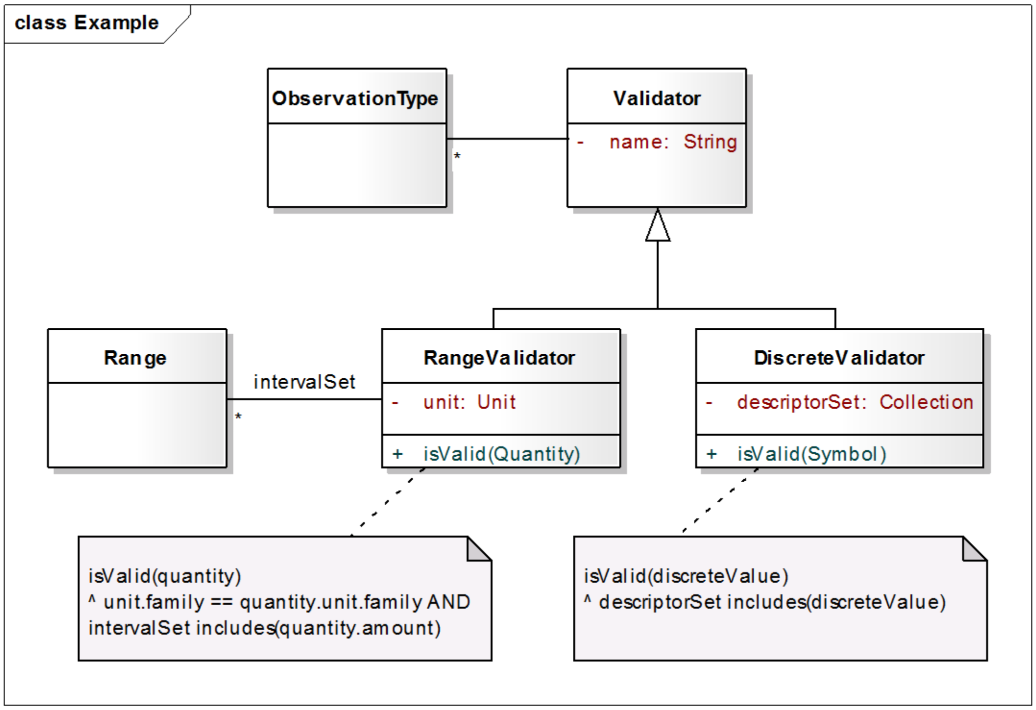

Figure 6 - Architecture for Observation Validation

```
public abstract class Validator {
	String name;

	public Validator(String name) {
		this.name = name;
	}

	public abstract boolean isValid(String value);

	public abstract boolean isValid(Double value);

	public abstract boolean isValid(Quantity value);

}
```

Figure 7– Validator

NO, and UNKNOWN. A greatly simplified representation of the `Validator` class hierarchy that was implemented is shown in Figure 6.

When a new `ObservationType` is created, it is associated with its appropriate `Validator`. After an Observation is created, it calls `isValid()` which delegates to its corresponding `ObservationType` which delegates to its `Validator` passing in the appropriate value type. Validators implement `isValid()` for built in primitive types as well as `Quantity` objects (Figure 7). `Validator` `isValid()` methods decide if the value is valid or not, returning a boolean.

There is a `DefaultValidator` that always returns true (Figure 8). This is an implementation of the `NullObject` pattern [WOO98], which was provided for those types of Observations that are always valid. For example, consider when a doctor observes you look shaky. Clearly any value entered as free-form text for the personal observation is valid, so that is why a `DefaultValidator` is associated with this property.

```java
public class DefaultValidator extends Validator {
	public DefaultValidator() {
		super("DefaultValidator");
	}
	@Override
	public boolean isValid(String value) {
		return true;
	}

	@Override
	public boolean isValid(Double value) {
		return true;
	}
	@Override
	public boolean isValid(Quantity value) {
		return true;
	}
}
```

Figure 8 – DefaultValidator

A `DiscreteValidator` validates whether the value is included in a set of known values (Figure 9).

```java
public class DiscreteValidator extends Validator {
	Set<String> legalValues;

	public DiscreteValidator(String name, Set<String> values) {
		super(name);
		legalValues = values;
	}

	public DiscreteValidator(String name, String... values) {
		this(name, new HashSet<String>(Arrays.asList(values)));
	}

	public DiscreteValidator(String name, String commaSeparatedValues) {
		this(name, parse(commaSeparatedValues));
	}

	@Override
	public boolean isValid(String value) {
		return legalValues.contains(value);
	}
```

Figure 9 – DiscreteValidator

Code for a `Range` class is also shown (Figure 10). A `RangeValidator` (Figure 11) ensures that a `Quantity` is within a range of valid values.

```java
public class Range {
	Quantity lowerLimit;
	Quantity upperLimit;

	public Range(Quantity lowerLimit, Quantity upperLimit) {
		if (lowerLimit.units != upperLimit.units)
			throw new RuntimeException("Range limits must use same units");
		this.lowerLimit = lowerLimit;
		this.upperLimit = upperLimit;
	}

	public Range(Double lowerLimit, Double upperLimit, Units units) {
		this(new Quantity(lowerLimit, units), new Quantity(upperLimit, units));
	}


	public boolean includes(Quantity value) {
		return value.units == lowerLimit.units
				&& lowerLimit.compareTo(value) <= 0
				&& value.compareTo(upperLimit) <= 0;
	}
}
```

Figure 10 – Range Class

```java
public class RangeValidator extends Validator {
	List<Range> ranges = new ArrayList<Range>();

	public RangeValidator(String name, List<Range> ranges) {
		super(name);
		this.ranges.addAll(ranges);
	}

	public RangeValidator(String name, Range... ranges) {
		this(name, Arrays.asList(ranges));
	}

	@Override
	public boolean isValid(Quantity value) {
		for (Range each : ranges)
			if (each.includes(value))
				return true;
		return false;
	}
}
```

Figure 11 – RangeValidator

2.7 Consequences

- ✓ System Robustness: applying this pattern when developing an AOM system provides a safer and more robust framework for changing application model and behavior by AOM engineers; hence minimizing regression and stability issues.
- ✓ Increased flexibility: It is possible to choose on a per-case basis whether to provide a basic validation on pre-defined types and entities, extensible validation framework which can be programmatically extended to handle domain specific concerns or a declarative, rule-based framework which support defining validations using domain terminology.

- ✓ Framework Evolution – separating the validation logic a different class hierarchy and plugging it to the framework as suggested in this pattern supports developing the framework progressively and adapting it per domain.
- ✗ Ease of use: Definition of complex domain-specific validations usually requires programmatic skills. Developing a DSL or rule engine is usually a time consuming task which requires few iterations.

2.8 Related Patterns

Several AOM patterns are related to the VALIDATION PATTERN:The DYNAMIC HOOK [AHS11] can be used to allow the AOM user to express complex validation logic in scripts. EVOLUTION RESILIENT SCRIPTS [HNS10] enhance the DYNAMIC HOOK by providing type-safety for scripting. DYNAMIC MODEL EVOLUTION [HLN10] relies on AOM validations to diagnose model inconsistencies when upgrading the core AOM application. Specifically, BREAK AND CORRECT allows the AOM team to fix the inconsistencies between Entities reported by the validation framework.

2.9 Known Uses

A medical-based AOM system developed by The Refactory for the Illinois Department of Public Health [YJ02] is an example of a system that extensively uses the observation validation framework described above. In addition to extensive use of the TYPESQUARE pattern for basic validations, reflection is also used to dynamically bind hook points. Custom behavior can be described as a dynamic method or a STRATEGY associated with new types of objects. Thus a new class can be created, and by using reflection, the new behavior can be dynamically associated with new types of diseases and invoked using stored descriptive information. Ultimately there were also validation rules that ensured constraints across multiple entities and properties.

This system also integrated follow-on workflow, implemented by an AOM micro-workflow system [MAN00] that was triggered when certain medical findings were detected during validation. For example, certain medical finding could trigger events for follow-up workflow such as medical treatment for an infant. Ultimately this system evolved and was re-implemented in the Java programming language where a rules engine (JRules) was used to define cross-entity validation rules.

Two adaptive systems for Invoicing and Import developed by The Refactory in C#/.NET use a simple rule language for describing rules for invoice calculation or data import to the system. Additionally, for rules outside the core DSL provided for the domain experts to express rules, a means to add new rules was provided by using dynamic hook points that defined known places where new behavior could be added. One dynamic hook point in the Import system allowed for adding new rules. New rules can be added by creating a DLL, which contains a subclass of `ValidationRule`. This class will be tagged with the name of the validation rule and have a `Validate()` method which is invoked during the validation process. By including the DLL in the configuration file that specifies what will dynamically loaded, new rules could be added. The following code example, shown in Figure 12, is a simplified definition for the `InvalidIdValidationRule` class, which ensures that invalid Ids are not accepted during the import of orders.

```
[ValidationRule("Invalid Id")]
public class InvalidIdValidationRule : ValidationRule{
  public InvalidIdValidationRule() : base() { }
   public override void Validate(ImportContext context)
...}
```

Figure 12 -- An example of a validation rule

Pontis Ltd. is a provider of Online Marketing solutions for Communication Service Providers. Pontis' Marketing Delivery Platform (MDP) allows for on-site customization and model evolution by non-programmers. The system is developed using ModelTalk [HLP09] based on AOM patterns. Pontis' MDP system is deployed in over 20 customer sites including Tier I Telcos. A typical customer system handles tens of millions of transactions a day exhibiting Telco-Grade performance and robustness. Pontis' MDP system aggregates data received from the Communication Service Provider's systems, such as information about a subscriber's usage patterns, and grants various benefits to subscribers based on the subscriber's data and the currently active promotions (e.g., a subscriber that sent 100 text messages receives a promotional coupon).

Pontis MDP is using AOM for customizing the generic product by non-programmers, using the system GUI. In the Pontis AOM, the validation is implemented by the various `EntityTypes`. It currently provides two types of validations:

1. `PropertyType` validation – min/max length, mandatory,
2. Script validation hooks – based on the Dynamic Hook and the Evolution Resilient Scripts patterns.

An AOM architecture is used in a channel marketing platform [Gu12] developed by e-Dialog (now part of eBay Enterprise). It delivers relevant and targeted engagement to consumer on various devices/screen through continuous optimization of recommendations. User-supplied JSON objects are validated to make sure they match the correct version of the object type. This is accomplished through a set of validating classes registered by type and version.

3. ACKWOLEDGEMENTS

We thank our shepherd Eduardo Guerra for his valuable comments and feedback during the PLoP 2013 shepherding process. We also thank our 2013 PLoP Writers Workshop Group led by Peter Sommerlad and participated in by Michael John, Youngsu Son, Jiwon Kim, Hyunchul Yang, Mihaela Cardei, Roland Bijvank, Wiebe Wiersema,Leo Pruijt, and Brahim Hamid, for their valuable feedback, suggestions, and comments.

4. REFERENCES

[AHS11] Acherkan, E.; Hen-Tov, A.; Schachter, Lior.; Lorenz, D.H.; Yoder. J.; Wirfs-Brock, R.; Dynamic Hook Points, 2nd Annual Asian PLoP Conference, Tokyo, Japan. October 2011.

[AOM] Adaptive Object-Models. http://www.adaptiveobjectmodel.com

[Fow97] Fowler, M.; Analysis patterns - reusable object models. Addison-Wesley series in object-oriented software engineering, Addison-Wesley-Longman 1997, ISBN 978-0-201-89542-1, pp. I-XXI, 1-357

[Fow10] Fowler, M.; Domain Specific Languages (1st ed.). Addison-Wesley Professional. 2010.

[FY98] Foote B, 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.

[Gu12] Lei Gu, Building an Adaptive Object Model, http://2rdscreenretargeting.blogspot.co.il/2012/07/building-adaptive-object-model.html

[GHJ95] Gamma, E., R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley. 1995.

[HLN10] Hen-Tov, A.; Lorenz, D. H.; Nikolaev, L.; Schachter, L.; Wirfs-Brock, R.; and Yoder, J. W.; Dynamic Model Evolution. In Proceedings of the 17th Conference on Pattern Languages of Programs (Reno/Tahoe, Nevada, October 17 - 21, 2010). SPLASH/OOPSLA 2010. ACM, New York, NY.

[HNS10] Hen-Tov, A.; Nikolaev, L.; Schachter, Lior.; Yoder. J.; Wirfs-Brock, R.; Adaptive Object-Model Evolution Patterns, SugarLoafPLoP 2010.

[HLP09] Hen-Tov, A.; Lorenz, D.H.; Pinhasi, A.; Schachter, L.; ModelTalk: When Everything Is a Domain-Specific Language, IEEE Software, vol. 26, no. 4, pp. 39-46, July/Aug. 2009.

[Jon99] Jones, S.; A Framework Recipe. Building Application Frameworks: Object-Oriented Foundations of Framework Design.Edited by Fayed, M., Johnson, R., Schmidt,.D. John Wiley & Sons. 1999.

[MAN00] Manolescu, D.; Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development. PhD thesis, Computer Science Technical Report UIUCDCS-R-2000-2186. University of Illinois at Urbana-Champaign, October 2000, Urbana, Illinois.

[WOO98] Woolf, B.; "Null Object", Pattern Languages of Program Design 3, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.

[YBJ01] Yoder, J.; Balaguer, F.; Johnson, R.; Architecture and Design of Adaptive Object-Models. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.

[YJ02] Yoder, J.; Johnson, R.; The Adaptive Object-Model Architectural Style. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002.

[YJR02] Yoder, J.; Johnson, R,; “Implementing Business Rules with Adaptive ObjectModels”. Business Rules Approach. Prentice Hall. 2002.