The Dynamic Relations Pattern

SINVAL VIEIRA MENDES NETO, Universidade Federal da Paraiba
RODRIGO ALMEIDA VILAR, Universidade Federal da Paraiba
AYLA DANTAS, Universidade Federal da Paraiba

This paper presents a design pattern to support the implementation of relationships between entities represented by metamodels in dynamically adaptable systems. The relationships that the Dynamic Relations pattern covers are simple association relationships, with one to one, one to many and many to many cardinality.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns*

General Terms:

Additional Key Words and Phrases: Adaptable systems, Relationships, Metadata

ACM Reference Format:

Vieira, S. M. N., Vilar, R. A. and Dantas, A. 2014. The Dynamic Relations Pattern. jn 2, 3, Article 1 (May 2010), 8 pages.

1. INTENT

This design pattern is intended to aid in the development of adaptable systems, specially regarding the implementation of relationships among their entities in a dynamic manner. We consider that a system is adaptable when it can be changed by end users, even if these end users have limited programming skills [Ferreira 2010]. The Dynamic Relations pattern allows developers to create and change relationships between entities at runtime. The relationship types covered by this pattern are one-to-one, one-to-many and many-to-many associations.

2. CONTEXT

When we model an object-oriented system, we should design its entities and the relationships among them. There are several types of relationships: association, generalization, flow and several kinds of dependencies, including realization and use [Rumbaugh et al. 2004].

The Unified Modeling Language (UML) defines **association** as a semantic relation among two or more classifiers which involves connections between their instances. Such connections can associate objects of different classes (the most common case), or can also link objects of the same class.

In order to establish a relationship and change it, developers need to clearly understand which classes are involved and the role of each one in the relationship. However, system users can request changes at any time, which can demand a lot of effort. Sometimes the changes require the creation of new entities and new relationships in the system. Sometimes they require changes in the cardinality of certain relationships between system entities.

Authors' email: sinval.vieira@dcx.ufpb.br, rodrigovilar@dcx.ufpb.br, ayla@dcx.ufpb.br

For example, in a customer relationship management system, we can have an entity called Person with attributes such as first name, last name, email and phone number. Nevertheless, after some time, the client may request some system changes in order to better deal with internal information regarding the phone number, such as provider, country and local area numbers and main number. In order to do so, the developer may need to create a new `PhoneNumber` entity and a new one-to-one "`Person has PhoneNumber`" relationship.

Later on, the client may need to manage several phone numbers for the same person (e.g. personal and professional numbers). Therefore, the cardinality of the relationship between Person and Phone would change from one-to-one to one-to-many. Such change in the `Person` class would need to be propagated to all system layers, such as Database and GUI.

In common object-oriented systems, developers implement entities and relationships using classes and references. As a result, the previous examples of changes would lead developers to write new source code, compile and deploy it again. Using another approach — the Adaptive object model (AOM) architectural style [Yoder et al. 2001] — and specially exploring it in different system layers, would enable system functionality changes to be performed immediately, at runtime. AOM uses flexible artifacts, such as databases and configuration files, to maintain system metadata information. This way, AOM systems can interpret metadata at runtime in order to produce business logic, GUI and persistence functionality. Every time metadata changes, the system can interpret it again and change at runtime.

AOM uses metadata to model system entities and relationships. Such relationships, however, can be implemented in several ways. Yoder [Yoder et al. 2001] suggests the use of the Property design pattern [Foote and Yoder 1998] twice. This pattern should be used to represent entities primitive attributes, like a string property to represent the `Person.firstname` attribute. It can also be used to represent entity properties which are also other entities, such as the `Person.phone` attribute. In this latter case we would have a representation of a relationship, but not in an explicit way. Important information such as navigability (unidirectional or bidirectional) or cardinality would be missing. According to Yoder [Yoder et al. 2001], system designers that use AOM feel the necessity of explicitly expressing the relationship, what makes the use of the Property pattern twice not to be a good option for them.

An alternative proposed by Ferreira [Ferreira 2010] is to merge the Property and Accountability [Fowler 1997] patterns in order to implement entity relationships in AOM systems. He proposes the use of two classes: `AccountabilityType`, which represents relation types, such as `Parenthood`; and `Accountability`, which links entities that relate to each other through a relationship (e.g. Sue and her son, John, are two entities from the parenthood relationship). In Figure 1, Ferreira shows an example of the implementation of the Accountability pattern through a diagram illustrating classes and objects regarding the parenthood relationship example.

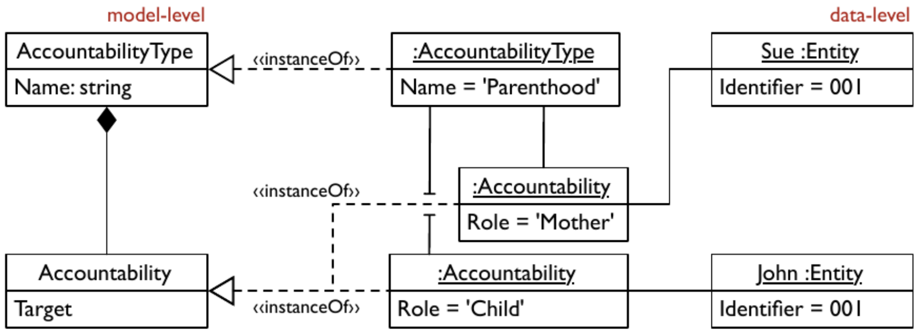

Fig. 1. Example of the use of the Accountability pattern to implement AOM relationships [Ferreira 2010]

Fowler defined the Accountability pattern to handle complex hierarchical relationships in a flexible and dynamic fashion [Fowler 1997]. For example, this pattern can organize the same elements of an enterprise structure into different hierarchies. For instance, we can visualize an administrative hierarchy with presidency, headquarters, departments and operational cells; or we can see a geographic hierarchy of the enterprise by continent, country, state and city. Accountability can also handle relationships that connect more than two objects (ternary, quaternary and so on). Conversely, most object-oriented relationships are simpler, since they do not represent hierarchies and can be modeled as references between two objects (source and target objects).

Considering the Accountability flexible and complex nature, the approach proposed by Ferreira to represent relationships needs two objects to link Sue and John as illustrated by Figure 1. Moreover, it is not clear what is the relationship cardinality and navigability. Besides, it may not be easy to understand how the `[Sue is] Mother` and `[John is] Child` accountability objects relate to each other. In a scenario where other people also relate to each other through the `Parenthood` relationship, this approach can be confusing and it may not be simple to identify Sue and John among other mothers and children.

In summary, we believe the current relationship approaches proposed to AOM may not be adequate to some situations, specially because they do not hold important data regarding the relationship in a cohesive way.

3. PROBLEM

How can we implement relationships between entities in dynamically adaptable metadata-based systems, managing relationship properties such as cardinality, navigability and roles, with a simple and cohesive design?

4. FORCES

In order to solve this problem, the Dynamic Relations pattern balances the following forces:

—It is important to enable changes on relationship structures at runtime and one way to do that is through metamodels;

—AOM proposes some solutions to dynamic relationships, managing metadata at runtime;

—Developers need more details to implement dynamic relationships in AOM;

—The implementation of relationships with metamodels should be simple and cohesive, to ease comprehension, but must contain all data needed in a relationship metamodel to enable some simple verifications;

—Relationship metamodels need to specify details about cardinality, navigability and participants types and roles, cardinality and navigability, in order to enable the automatic validation of relationships at runtime.

5. SOLUTION

The *Dynamic Relations* pattern proposes a way to implement relationships in dynamically adaptable systems based on metamodels. It allows the creation of new relationships between entities of an application and changes in these relationships to be performed at runtime. Besides, as it is a design pattern, we believe that it can make concrete some ideas proposed by AOM for the creation of totally adaptable systems, including the relationships between entities of such systems.

This pattern is intended to use less objects than the ones proposed by solutions based on the *Accountability* pattern. Moreover, it also proposes the use of additional information regarding relationships, such as their cardinality and the types of the entities involved, allowing this way a validation mechanism to be used when creating new relationships between entities.

In order to do that, the pattern suggests the use of a class called `RelationType` to model the restrictions of relationships between types of entities (classes) of a system. Besides, it also proposes the use of another class, called `Relation`, to represent the instances of relationships between two entities (objects). For each relationship in the conceptual model of the system, an instance of a `RelationType` should be created, storing the relationship name, the types of source and target entities and the relationship cardinality. There should also be some information indicating if the relationship is unidirectional or bidirectional. In this latter case, it would be necessary to indicate the reverse name of the relationship (from target to source).

From a `RelationType` $RT(A, B)$, between entity types A and B, several instances of `Relation` (r_k) can be created between entities of type $A : (a_i)$ and entities of type $B : (b_j)$. Such relations could have the format $r_k(RT(A, B), a_i, b_j)$. The r_k Relations should also know their `RelationType` RT, because they should follow the restrictions stated by RT. For instance, the `RelationType` "A person has several phone numbers" defines bidirectional relationships between instances of the `Person` entity (source) and instances of the `PhoneNumber` entity (target). A person should have zero to N phone numbers and a phone number can be of only one person.

When a `Relation` r_1 is created, between one `Person` p_1 (Joseph) and a `PhoneNumber` $e_1(Personal, Verizon, +55, 83, 87879090)$, it is necessary to verify if there is a `RelationType` related to r_1, if p_1 is a `Person`, if e_1 is a `PhoneNumber`, and if it does not exist another `Person` associated with the same phone number considering this `RelationType`.

These rules should be checked every time an object from class `Relation` is created. Every `Relation` should have a reference to a `RelationType`, which defines the rules it should follow. For instance, an existing relationship can be changed from 1-N to N-M. In order to do this change, we should simply change the cardinality of the relationship type source (sourceCardinality) from 1 to * (many).

Following the *Dynamic Relations* pattern, the changes in the requirements of a `Relation` can be implemented modifying the attributes of the `RelationType` that defines this `Relation`. For example, changes in cardinality, navigability or source/target entities can be done at runtime, in the metamodel, without the need of recompiling the system source code.

It is important to notice, however, that the scope of the *Dynamic Relations* pattern only covers association relationships. Therefore, the pattern does not model relationships such as inheritance.

6. STRUCTURE

The *Dynamic Relations* design pattern introduces four basic elements (also considering the Type Square pattern elements, which are used in AOM and are also related to this work):

—**EntityType**: This class represents the type of the entity, and it is where the entities metadata are described. An entity structure is represented by `EntityType` and the entities themselves are represented by the `Entity` class.

—**Entity**: The class that represents the "instance" of the entity and it is where, in fact, the entity data are stored. This class is directly related to its type, which is represented by the `EntityType` class. By this distinction, `Entity` (data) and `EntityType` (metadata), it is possible to create and to change entities at runtime.

—**RelationType**: This class is responsible for storing the information that describes the relationship between entity types. The attribute "sourceType" references the `EntityType` of the source entity in the relationship. The attribute "targetType" references the `EntityType` of the target entity in the relationship. The attributes "sourceCardinality" and "targetCardinality" represent the cardinalities of each entity in the relationship.

—**Relation**: This class is used to instantiate relationships. A `Relation` has references to the Entities in the relationship. These, in turn, are described by attributes "source" and "target". A `Relation` also has a reference to the RelationType that represents the relationship type. When a `Relation` is created, observing the "sourceType" and "targetType" that are defined in the RelationType, it is possible to verify the types of the entities (source and target) that are allowed for the relationship.

These elements and their relations are illustrated by Figure 2 using a UML classes diagram. Specific elements of the *Dynamic Relations* pattern (`Relation` and `RelationType`) are directly connected to two classes provided by the Type Square pattern [Yoder et al. 2001]: `EntityType` and `Entity`. These classes can be used in the development of dynamically adaptable systems based on metamodels to represent classes and objects of a traditional object-oriented system.

Therefore, the Dynamic Relations pattern and Type Square pattern applied together have a base structure composed by six classes, illustrated by Figure 2. The `PropertyType` and `Property` classes, also shown in Figure 2, represent the attributes of the entities and their types, which may be simple types (e.g. `String`, `int`). For properties of entities whose types are references to any other `EntityType`, we suggest that these properties should be represented using a `Relation` according to the *Dynamic Relations* pattern.

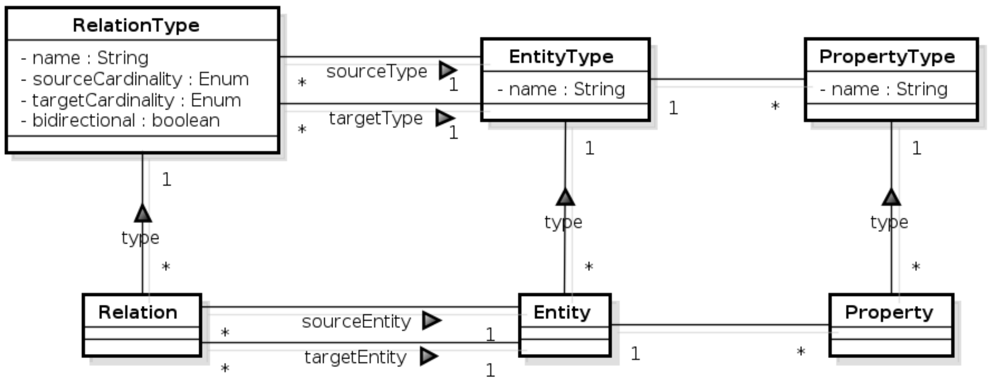

Fig. 2. Classes of the Dynamic Relations pattern together with Type Square used in AOM.

7. DYNAMICS

The dynamics of the *Dynamic Relations* pattern is illustrated by Figure 3. When creating a dynamically adaptable system, we should define its entities and their types, as well as these entities simple properties. Furthermore, we should also define the relationships between entities, and perform some verifications regarding constraints of the relationship. For instance, we should evaluate the types of the entities that are involved and also check if the cardinality defined for the relationship is being respected.

8. CONSEQUENCES

The pattern discussed in this paper presents the following advantages:

—It makes explicit the relationship between entities and their characteristics (navigability, cardinality);

—The representation of relationships is made by a simple class (`RelationType`) that allows the verification of constraints;

—It shows in a concrete way how dynamically adaptable systems should be implemented.

Similarly, the following drawbacks can be observed in the pattern:

—The `Relation` and `RelationType` classes store more information than classes of alternative implementations of relationships in metamodels, such as the Accountability pattern.

—It is less flexible than Accountability and can not represent hierarchies neither ternary relationship, because it works only with simple binary associations.

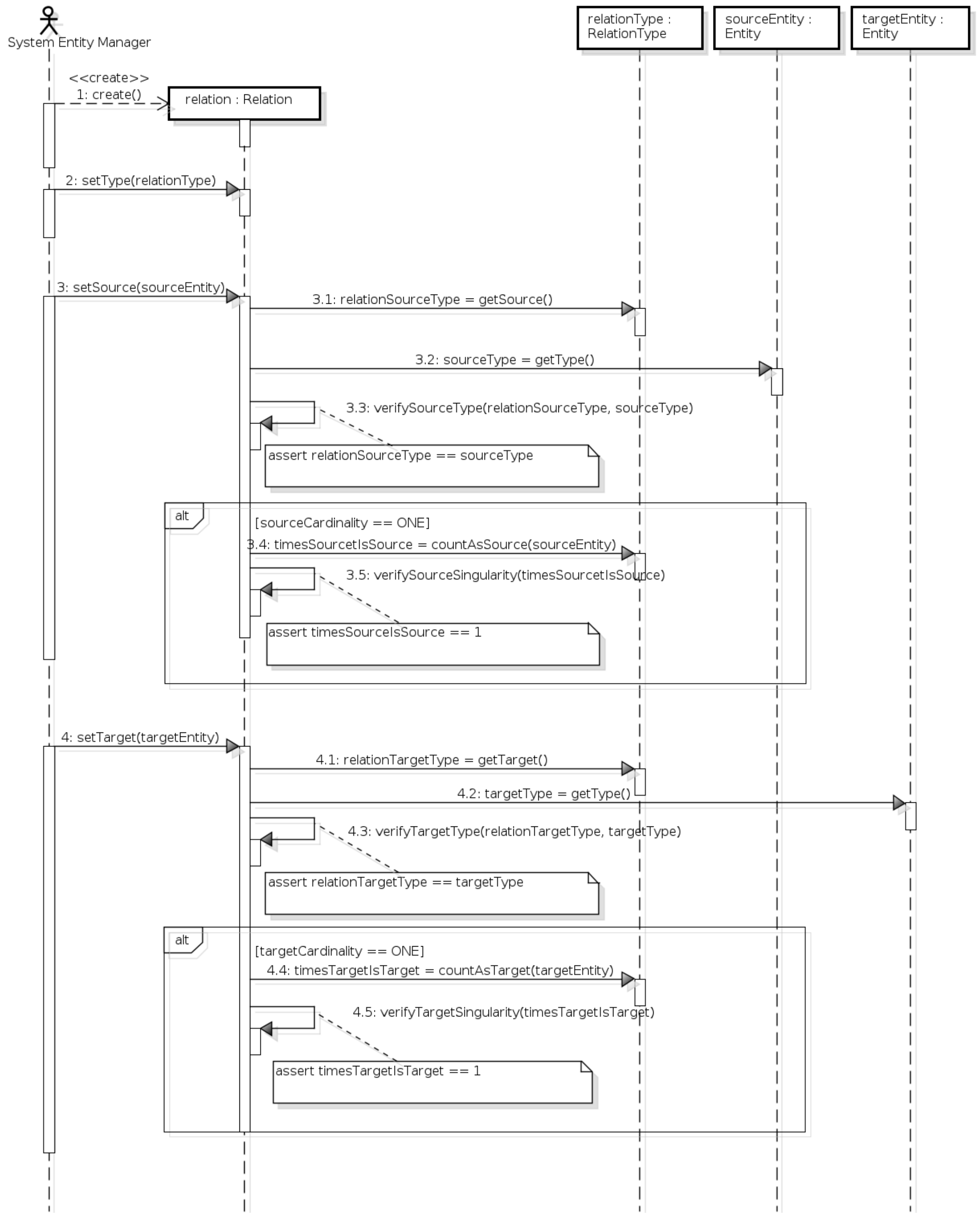
System Entity Manager
relation : Relation
relationType : RelationType
sourceEntity : Entity
targetEntity : Entity
<<create>>
1: create()
2: setType(relationType)
3: setSource(sourceEntity)
3.1: relationSourceType = getSource()
3.2: sourceType = getType()
3.3: verifySourceType(relationSourceType, sourceType)
assert relationSourceType == sourceType
alt
[sourceCardinality == ONE]
3.4: timesSourcetIsSource = countAsSource(sourceEntity)
3.5: verifySourceSingularity(timesSourcetIsSource)
assert timesSourceIsSource == 1
4: setTarget(targetEntity)
4.1: relationTargetType = getTarget()
4.2: targetType = getType()
4.3: verifyTargetType(relationTargetType, targetType)
assert relationTargetType == targetType
alt
[targetCardinality == ONE]
4.4: timesTargetIsTarget = countAsTarget(targetEntity)
4.5: verifyTargetSingularity(timesTargetIsTarget)
assert timesTargetIsTarget == 1

Fig. 3. Dynamics of the Dynamic Relations pattern

9. EXAMPLE

To exemplify the *Dynamic Relations* pattern, we will now consider a system that manages information about people and that was developed in Java [Gosling et al. 1996]. The entities considered for this example are Person and PhoneNumber.

Our goal with this example is to explain how the *Dynamic Relations* works in a real scenario. Assuming Person and PhoneNumber are types of entities (`EntityType`), so there are instances of `EntityType` that can be called "PersonType" and "PhoneNumberType", based on the ideas of AOM and Type Square. The relationship between them is a "one to many", where "one Person has many PhoneNumbers". Figure 4 shows how this relationship should be represented using an object-oriented design.

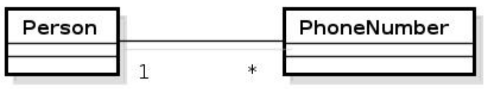

Fig. 4. Example of the object-oriented modeling of part of the system

It is important to remember that *Dynamic Relations* is a pattern to implement relationships between entities that are represented as metamodels, as AOM does. So, we have two `EntityType`s (PersonType and PhoneNumberType) and their instances, where Joe and JoesPhoneNumber are entities (Entity) that reference PersonType and PhoneNumberType respectively.

To implement dynamic relationships between PersonType and PhoneNumberType, we need to use the `RelationType` and `Relation` classes. In the Structure session of this paper, the purpose of `RelationType` and Relation was discussed. Figure 5 presents an object and classes diagram that illustrates how the *Dynamic Relations* pattern implementation works in conjuction with the Type Square pattern, and, how the entities will relate at code level for the Person and PhoneNumber example.

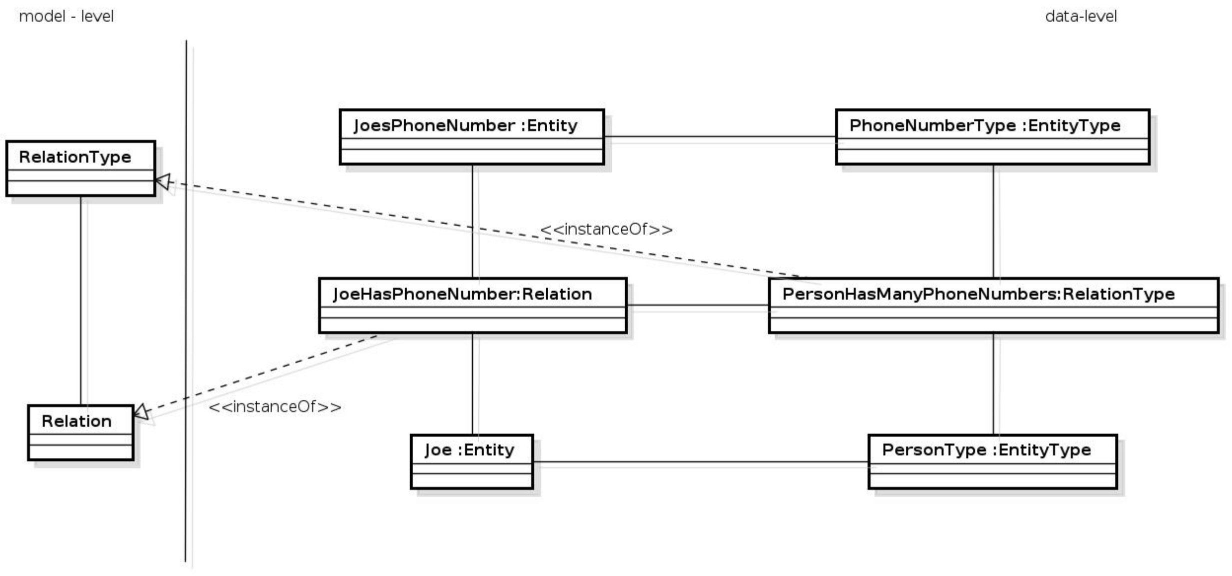

Fig. 5. Representation of the example in an object and classes diagram

The following source code illustrates the creation process for Person and PhoneNumber entities and their types and also how to establish the relationship between these entities.

```
EntityType personType = new EntityType("PersonType");
EntityType phoneNumberType = new EntityType("PhoneNumberType");

Entity joe = new Entity(personType);
Entity joesPhoneNumber = new Entity(phoneNumberType);

RelationType personHasManyPhoneNumbers =
    new RelationType("PersonHasManyPhoneNumbers", personType,
    phoneNumberType, Cardinality.ONE, Cardinality.MANY);

Relation joeHasPhoneNumber =
    new Relation(personHasManyPhoneNumbers, joe, joesPhoneNumber);
```

10. RELATED PATTERNS

—Systems based on Adaptive Object Model represent attributes, classes and relationships as metadata [Yoder et al. 2001]. AOM can be used to represent all the system or just for some parts of the system that have to be adaptable. The *Dynamic Relations* pattern is totally related to AOM, once it uses the Type Square pattern and it is also intended to be used in the creation of dynamically adaptable systems based on metamodels.

—Accountability represents a more powerful, and also more complex solution to deal with various types of relationships in organization structures in a flexible manner [Fowler 1997]. So, like with other powerful tools, it is not advisable to use these tools unless you really need them.

—The Type Object pattern [Johnson 1997] is related to the way the RelationType and Relation classes work together. Type Object has the intent to separate the classes from their instances. Therefore, these classes (Objects) can be implemented as instances of other classes (Types). The pattern allows the dynamic creation of “new classes”. Thus, it is possible to create new types of relationships between entities at runtime, and this allows a system to provide its own rules for type checking, which can lead to smaller and simpler systems.

11. KNOWN USES

The *Dynamic Relations* pattern is used in LOM (Living Object Model) [1], a framework based on AOM which is under development and that allows the creation of different and totally adaptable systems at runtime.

REFERENCES

FERREIRA, H. S. 2010. Adaptive object-modeling: Patterns, tools and applications. Ph.D. thesis, University of Porto, Faculty of Engineering.

FOOTE, B. AND YODER, J. 1998. Metadata and active object-models. In *Proceedings of Plop98. Technical Report# wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science.*

FOWLER, M. 1997. *Analysis Patterns: Reusable Objects Models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GOSLING, J., JOY, B., AND STEELE, G. L. 1996. *The Java Language Specification* 1st Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

JOHNSON, M. R. 1997. Type object. In *Pattern Languages of Program Design 3.* AddisonWesley, 47–65.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 2004. *Unified Modeling Language Reference Manual, The.* Pearson Higher Education.

YODER, J. W., BALAGUER, F., AND JOHNSON, R. 2001. Architecture and design of adaptive object-models. *SIGPLAN Not. 36,* 12, 50–60.

Received September 2014; revised October 2014; accepted October 2014

[1] https://github.com/rodrigovilar/lom