Using Annotations in the Naked Objects framework to explore data requirements

Marcos E. B. Broinizi
Department of Computer Science
University of São Paulo
Rua do Matão, 1010 - Cidade Universitária
05508-090 São Paulo - SP - Brazil
55 11 3038-8571
broinizi@ime.usp.br

João Eduardo Ferreira
Department of Computer Science
University of São Paulo
Rua do Matão, 1010 - Cidade Universitária
05508-090 São Paulo - SP - Brazil
55 11 3091-6111
jef@ime.usp.br

Alfredo Goldman
Department of Computer Science
University of São Paulo
Rua do Matão, 1010 - Cidade Universitária
05508-090 São Paulo - SP - Brazil
55 11 3091-6111
gold@ime.usp.br

ABSTRACT

The creation of conceptual data design that appropriately represents specific application domain is one of the main challenges in requirements engineering. An initiative to help designers is the *Naked Objects* framework, where it is possible to interact with conceptual model in a limited way. The interactions are restricted to entity creations and single object-relations. We created an extension of the Naked Objects framework using annotations to allow manipulation of higher level abstractions as specialization and object-relationship. These abstractions allow better interactions between the domain specialist and designers. The use of our approach to explore and validate data requirements has several benefits: 1) It reduces conceptual specification problems (like poorly data requirements identification); 2) It narrows the distance among domain and design specialists; 3) It allows the simultaneous exploration of the conceptual data design and the system requirements.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/ Specifications; H.2.8 [**Information Systems**]: Database Management—*Logical Design*

General Terms

Data requirements exploration

Keywords

Agile methods of software development, conceptual data design, data abstractions, requirements engineering, requirements specification

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

1. INTRODUCTION

The creation of conceptual data design involves transformation of real problems into representations that can be implemented [6]. The process abstracts data from the real world in order to build a schema to represent the data. Semantically integrated data abstractions compose the schema to represent the real data [10].

We base our concepts on Naked Objects approach which reduces requirements identification problems and improves conceptual data design precision and agility. In order to reach these goals, the domain specialist participation is essential to validate the conceptual data design.

Our validation proposal is based on data abstractions [10] in order to improve precision of the conceptual data design. Besides precision, the agility of validation process is also improved by the use of concepts inspired on the **Naked Objects** (NO) [15]: *the role of the user of a system should be as a problem-solver*. The domain specialist is the only one who has the information to solve conflicts in the conceptual project. Using Naked Object concepts appropriately, information becomes clearer as soon as possible when system requirements are gathered. This allows the simultaneous exploration of the conceptual data design and the system requirements.

The ordinary Naked Objects framework [13] does not cover all necessary abstractions for creating the conceptual data design. We developed an extension of the Naked Objects framework to better support not only the several object relationship types, but also the generalization and composition abstractions. The extension is a set of semantic labels which implementation is based on *annotations*.

The paper is organized as follows: Section 2 covers previous related works; Section 3 presents the developed annotations; Section 4 explores our approach agile characteristics; finally, the conclusion can be found in Section 5.

2. RELATED WORK

The Naked Objects [14, 15] approach has great synergy with agile methods of software development. According to [16], it is a set of ideas that can be understood as:

> ... an architectural pattern whereby core business objects are exposed directly to the user instead of being masked behind the conventional constructs of a user interface.

The Naked Objects framework has been designed to support the naked objects pattern and allows the definition of objects as *Java* classes following a previously defined set of code conventions [13]. The code conventions make the automatic object-oriented interface creation possible. The framework has an execution environment that includes a visualization mechanism to create manipulable object representations.

The most remarkable aspect of the framework is its use for system requirements exploration [16]. Objects can be identified through conversations between the domain specialist and the developer. Then, the developer implements a prototype in the framework. The prototype is presented to the domain specialist that may manipulate the identified requirements. A fast cycle is established: new requirements are identified, the prototype is updated and eventually manipulated again by the domain specialist. The cycle is repeated until the conceptual design appropriately represents the necessary system requirements.

The framework allows quick prototyping of computer systems through definition of objects in the specific application domain. The user actions in the prototype are object creation and recovery, attribute value specification, association between objects, or object (or collection of objects) method invocation. The framework represents the system prototype as an object-oriented interface that has the following characteristics:

- icons represent classes or entity types and can be used to create new instances;
- an instance is also represented by an icon or a form listing the attributes and values of the instance;
- the form may be used to change attribute values;
- context menus may be used to invoke methods. Method parameter that is an instance of other object can be executed by *dragging* and *dropping* the parameter instance on the target instance.

We use data abstractions to assure the necessary precision in the conceptual project. Data abstractions are a common denominator between the developer and the domain specialist and allow the mapping of specific behaviors for each abstraction in the validation environment. These abstractions have been widely studied by the data community [5,7,10,18,19] and they are the foundation of an appropriate conceptual design.

Data abstractions may have different name or meaning in computer science. We briefly present the abstractions we use in order to avoid misinterpretations and make the reader understanding easier.

The most basic abstraction is the *classification abstraction* [10] which represents a domain object as a class or an entity type. Other abstraction represents relationships between entities types. The relationships are called *relationship abstractions*. Other abstractions are also used: *generalization specialization*, *composition* and *object-relationship*.

Generalization-specialization abstraction has been studied in the literature [10]. For the purpose of our study, we take into account only specializations that build hierarchies that do not contain multiple inheritance. More details covering how this type of data specialization hierarchies are used can be found in [8, 12].

Composition abstraction can be understood as the relationship between the whole and its parts [10]. This abstraction can also be classified as **physical** or **logical**. The main structural difference between these two classifications is due to the referential integrity. When we remove an instance of the composite entity defined in a given logical composition, the component entities may continue to exist. In the case of the physical composition, this exclusion means excluding all the component entities. There is no consensus regarding the denomination of this abstraction. For example, in the object-oriented modeling language UML [20], the concept of logical composition is also called *aggregation* and the physical composition is referred to as *composition aggregation*.

We still use the term *object-relationship* as the abstraction that represents an entity type relationship. This abstraction has different denominations depending on the used model. For example, in the Entity-Relationship Model, Korth calls it as, also, *aggregation* [11].

3. ABSTRACTIONS AS ANNOTATIONS

Our proposal extends the Naked Objects framework allowing its use as a prototyping tool to represent the data abstractions mentioned above. Some data abstractions can be directly represented in the Naked Objects framework. We have increased the representation capacity of the framework to include all necessary abstractions. Moreover, a great amount of code is necessary to implement the data abstractions just following the ordinary framework conventions.

We have created a set of semantic labels to identify the data abstractions, describing these new semantic values and attaining more dynamism to implement it.

A data abstraction is represented as a programming language annotation [4]. Annotations [1] are meta-data that allow the registration of additional information regarding a given programming class. The meta-data does not interfere with the semantics of the program. Annotations only affect how the programs are dealt with by tools and libraries.

We have developed a tool that is able to identify the annotation and automatically generate the necessary functionalities to represent the data abstraction behavior. In this section, we introduce the annotations that have been implemented, clarifying the improvements they bring and how these improvements influence agility and precision during conceptual design creation.

The first and most simple annotation is called **Entity**. It directly refers to the classification abstraction. This annota-

[1]The specification of the annotations and its use is obtained using a pseudo-language. The pseudo-language direct mapping is easily identified in programming languages like Java 5.0 [4] because an *annotation* is defined as a *public @interface* and its use is the same in the pseudo-language (@<*Annotation_Name*>).

Example:

```
@Entity
class Dentist
   ...
   @RelationshipObject(
      cardinality = Cardinality.ManyToMany,
      relatedWith = Patient,
      fieldRelatedName = "dentist",
      compositeClass = Appointment,
      compositeFieldName = "appointments",
      compositeFieldRelatedName = "patient"
   ) Collection appointments;
...
@Entity
class Patient {
   ...
   Collection appointments;
...
@Entity
class Appointment {
   ...
   Date date;
   Patient patient;
   Dentist dentist;
...
}
```

(a)

```
@Entity
class Treatment {
   ...
   Collection apointments;
...
}
```

(b)

Definition:

```
annotation RelationshipObject {
   Cardinality cardinality();
   class relatedWith();
   string fieldRelatedName();
   class compositeClass();
   string compositeFieldName();
   string compositeFieldRelatedName();
}
```

/ This annotation has six parameters: the first one is the* `cardinality` *constraint that can have the values* `OneToOne`, `OneToMany`, `ManyToOne` *and* `ManyToMany`*; the second one is the other class of the relation,* `relatedWith`*; the third one is the name of the attribute of the other class that makes the association,* `fieldRelatedName`*; the fourth one is the class that represents the relationships as an entity,* `compositeClass`*; the fifth and the sixth ones are the names of the atributes that represent the association in the entity of relationship,* `compositeFieldName` *and* `compositeFieldRelatedName`*.*/*

(c)

Figure 1: Object-relationship annotation.

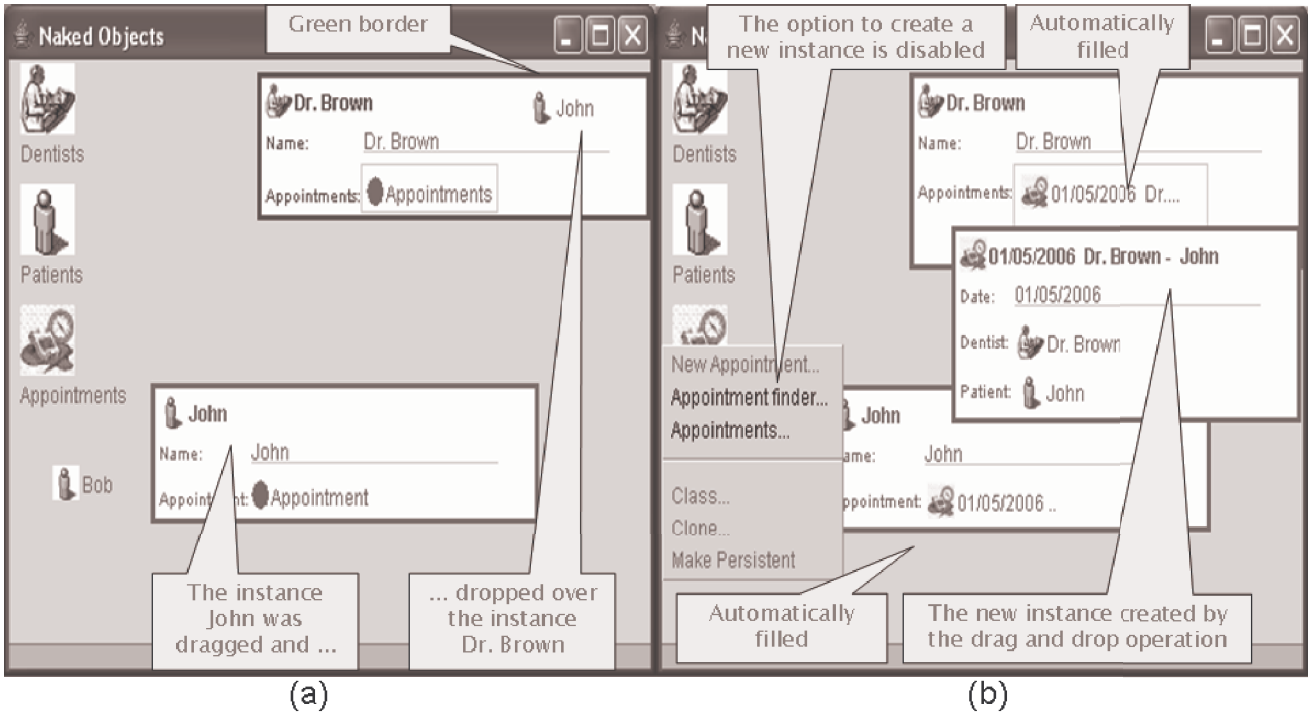

Figure 2: Naked Objects graphical environment

tion should be applied in a class to emphasize its role as an entity type of the application domain. Associations that represent any relationship between entity types are described by annotating an attribute from one of the participating entity types with the annotation **Relationship Association**. Similarly, associations that represent compositions between entity types are described by annotating an attribute from one of the participating entity types with the annotation **Composition Association**.

The annotations bring relevant improvements to the conception of conceptual data designs, but the details of their use and the description of the improvements are not included herein due to space limitations. However, the annotation **Object-relationship** and the annotation **Generalization-specialization** are described with simple examples of their usage in the following sections so that the power of representation and the advantages of the approach are illustrated.

3.1 Object-relationship

When the relationship has its own attributes, many approaches suggest placing the attributes in one of the entity types of the relationship. However, this is not an appropriate solution as it detaches the attribute from the place where it truly belongs to. This constraint is evident when the relationship is represented by an annotation **Relationship Association**.

This situation has led to the need of creating another annotation to allow the appropriate represent relationships that have their own attributes or that are directly related to other entity types. We call this **Object-relationship**.

Relationships between entity types that need to be represented by their own entity types are described by annotating an attribute from one of the participating entity types with the annotation **Object-relationship**. This annotation represents the abstraction of the same name described in Section 2.

In Figure 1, there are three entities that are appropriately annotated with the **Entity** annotation: `Dentist`, `Patient` and `Treatment`. There is a relationship between a `Patient` and a `Dentist`. We could define this relationship as an appointment. If the appointment did not have any additional attribute, we could represent it using the **Relationship Association** annotation. However, we want to represent the date in which the said relationship took place and perhaps link the appointment with another entity, `Treatment`. In this case, we can use the **Object-relationship** annotation. We define an entity type to represent this relationship at first: `Appointment`. We include the attribute `date` in the `Appointment` entity type. In Figure 1a, we complete the definition of the entity type `Appointment` preparing a relationship between `Patient` and `Appointment` and another one between `Dentist` and `Appointment`. Finally, we can annotate the `Appointment` relationship attribute in class `Dentist` with the **Object-relationship** annotation, according to the definition presented in Figure 1c.

On the left of Figure 2a, the icons that represent the entity types can be seen: `Dentists`, `Patients` and `Appointments`. Immediately below them, an instance of the entity type `Patient`, `Bob`, is represented as an icon. On the right, another instance of `Patient`, `John`, is represented by a form, as well as an instance of `Dentist`, `Dr Brown`, above. When the instance `John` is dragged, the cursor becomes an icon. When it is positioned on the instance `Dr Brown`, the frame around the form that represents that instance becomes green. This

Example:

```
@Generalization(
   completeness = Completeness.Partial,
   disjointness = Disjointness.Overlapping
)
class Person {
   WholeNumber age;
   ...
}
```

(a)

Definition:

```
annotation Generalization{
   enum Completeness{Total, Partial}
   enum Disjointness{Disjoint, Overlapping}
   Completeness completeness();
   Disjointness disjointness();
}
```

/ This annotation has two parameters: the first one,* `completeness`*, indicates if the generalization is* `Total` *or* `Partial`*; the second one,* `disjointness`*, indicates if the relation is* `Disjoint` *or* `Overlapping`*. */*

(b)

Figure 3: Generalization annotation.

Example:

```
@PredicatedSpecialization(
   specializes = Person,
   fieldName = "age",
   operator = Operator.greaterThanEqualTo,
   value = "18"
)
@Entity
class Employee{
   ...
}
...
@Specialization(
   specializes = Person
)
@Entity
class Student{
   ...
}
```

(a)

Definition:

```
annotation PredicatedSpecialization {
   class specializes();
   string fieldName();
   Operator operator();
   string value();
}

enum Operator {
   equalTo, notEqualTo, lessThan,
   lessThanEqualTo, greaterThan,
   greaterThanEqualTo
}
```

/ This annotation has four parameters: the first one,* `specializes`*, indicates the parent type of entity of the annotated class; the second one is the name of the attribute of the parent type of entity that belongs to the predicate,* `fieldName`*; the third one is the conditional operator of the predicate,* `operator`*; finally, the fourth one is the value that belongs to the predicate,* `value`*. */*

(b)

Figure 4: Predicated Specialization annotation.

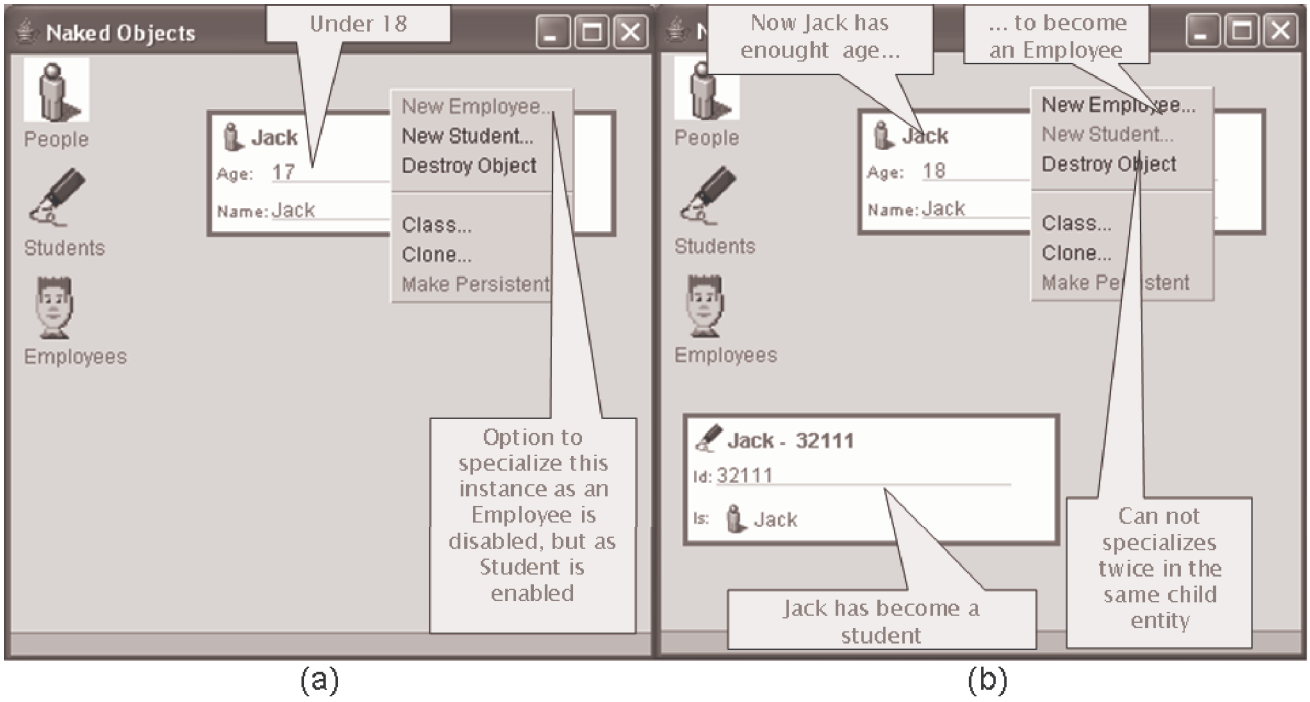

Figure 5: Naked Objects graphical environment

indicates an action execution is possible if we dropped the dragged instance on the form – the creation of a relationship that is represented by an instance of `Appointment`. In Figure 2b, the result of the action can be seen, establishing an instance of `Appointment` and automatically filling the fields responsible for the relationship.

Although there is a representation of the entity type `Appointments`, the option of creating an instance through its context menu is disabled. The only way to create an instance of an object-relationship is by dragging and dropping action as described previously. In the graphic interface, the annotation represents the constraints of the abstraction concept it refers to, allowing the domain specialist to test and validate the prototype as the concept is able to appropriately represent domain requirements.

Improvements: This annotation allows the exploration and identification of one of the most difficult concepts in conceptual data design: a relationship that has to be represented as an entity type. At the beginning of the process, the need for a relationship between two entity types of the domain such as, for example, a patient and a dentist can be identified. The relationship can initially be represented by the **Relationship Association** annotation. It is even common to use a name for the relationship such as, for example, `Appointment`. However, other aspects become visible when the relationship is tested attesting the necessity of representing this relationship as an **Object-relationship**.

3.2 Generalization-specialization

We develop two annotations to represent the generalization-specialization abstraction: the first should be applied to the parent entity type of the relation and the other one to the children entity types.

The generalization annotation should be applied to the class that plays the role of the parent entity type in the relation.

There are two annotations that can be applied to a child entity type of the generalization relation: `Specialization` and `PredicatedSpecialization`. The latter is defined below and the former is a simplification of the latter, having only one attribute to identify the parent entity.

Consider the following example with three classes: `Person` (Figure 3), `Employee` and `Student` (Figure 4). An instance of `Person` can be specialized as an instance of `Student` or an instance of `Employee` initially by annotating the class `Person` with a generalization annotation (Figure 3). Then, we annotate the classes `Student` and `Employee` with a specialization annotation (Figure 4).

The behavior of the interface depends on the generalization parameters. The interface of the example is illustrated in Figure 5. The generalization is overlapping and partial. By being partial, the generalization allows the creation of an instance of the entity type `Person` without being an instance of any of the two children entity types. In Figure 5a, the instance `Jack` is an entity type `Person` instance. If the generalization were total, we would have to create an instance of `Employee` or `Student` in order to obtain an instance of the entity `Person`. Additionally, the generalization is overlapping allowing an instance of `Person` to be specialized as more than one child entity type. That is, a single instance, like `Jack`, can be specialized as an instance of `Student` and as an instance of `Employee` at the same time. In Figure 5b, the instance `Jack` is already specialized as `Student` and could still be specialized as `Employee`. If the generalization were disjointed, only a single specialization would be allowed in

one single instance of the parent entity type.

Finally, we can tell the difference between the use of annotation `Specialization` in class `Student` and the annotation `PredicatedSpecialization` in class `Employee`. As there is no predicate associated with the class `Student`, it is always possible to specialize an instance of `Person` (in Figure 5a, the context menu is enabled as `New Student`) as long as it has not already been specialized as `Student` (in Figure 5b, the menu is disabled as `New Student`). On the other hand, to specialize an instance of `Person` as an `Employee`, the predicate must be met. In this case, the `age` must be at least `18` (disabled in Figure 5a and enabled in Figure 5b).

Improvements: These annotations elucidate generalizations and specializations through the interface and allows the exploration of advanced concepts in these relations. The first concept is the totality or partiality of generalization. The second concept is whether the generalization is disjointed or overlapping. The third concept is whether the specialization is defined by the predicate or by the user. The accurate identification and validation of these concepts allow a better domain requirements understanding, reaching higher levels of precision in the conceptual data design.

4. RESPONDING TO CHANGE

The possibility of changing the represented by doing few small modifications allows better interaction between the domain specialist and the designer. With our Naked Object extension, the domain specialist can have a better understanding of abstract data concepts using the graphical interface as well as observing its behavior.

Responding to change prioritization instead of simply following a plan is one of the four values upheld by the Agile Manifest [2]. Therefore, our approach can be considered closer to some values of agile software development.

4.1 Changes in annotations

Consider the design described in Figures 3 through 5 as an example of the capacity of our approach to respond to change. Consider it would be interesting to change the generalization shown in Figure 3 in such a way that it is not possible for a `Person` to be both a `Student` and an `Employee` simultaneously during the exploration of requirements by any other reason.

The generalization was previously described as overlapping. Changing it to a disjoint generalization is sufficient to appropriately represent the new requirement. The behavior of a disjoint generalization avoids the same instance of a parent entity type can be specialized as more than one child entity type. A single instance like `Jack` can be specialized as an instance of `Student` or as an instance of `Employee`, but not an instance of both at the same time.

In order to disjoint the generalization, we simply change the value of the disjoint parameter in the generalization annotation of `Person` in Figure 3 to `Disjointness.Disjoint`. The behaviour change of this modification takes place in Figure 5b. The instance `Jack` was already specialized as `Student` and can not be specialized as an `Employee` anymore. The respective menu option is disabled. Since the generalization becomes disjoint, only one specialization is allowed in one single instance of the parent entity type.

Despite the only visible effect of this change is a disabled menu item, other behaviour modifications come to scene. New drag and drop operations constraints are created. The implementation of these behaviour modifications in the Naked Object framework without the use of annotations would demand the change of many lines of code in different methods of several classes.

4.2 Agile Values

Characteristics	Agile values
The conceptual project is available as a manipulable prototype, eliminating the explicit need for additional documents	Working software over comprehensive documentation
Valuing the work of the specialist as a problem solver	Customer collaboration over contract negotiation
The interaction between specialist and developer - mandatory to achieve the precision - is facilitated due to the validation interface based on data abstractions	Individuals and interactions over processes and tools

Figure 6: Agile values

Naked Objects also make the use of agile development techniques [1,3,17] easier. Agile methods respect and follow the values and principles described in the Agile Manifest [2]. Agile Manifest upholds the four values: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan. Figure 6 presents a table that summarizes other characteristics of our proposal and the respective agile values.

The evolution of the conceptual data design also happens through successive interactions, validations and changes in the project. This reinforces the incremental design which is essential for several agile methods.

5. CONCLUSION

Our work uses the hypothesis of the domain specialist participation in the requirements identification and validation processes to reach a more appropriate agility and precision in the process of creating the conceptual data design.

The proposed extension of Naked Objects framework also makes the use of agile development techniques easier, improving the agility of the data requirements exploration. The work is based on a complete set of abstractions (entity, relationship, composition, object-relationship and generalization-specialization) assuring precision for the requirements representation and appropriate data requirements representation. The abstractions are described as annotations.

One of the main contributions of our work is to narrow the distance between the domain specialist and the data designer through the definition of annotations to create conceptual data designs. The annotations have the following characteristics:

- they establish a dynamic and manipulable alternative to the conceptual design description through a functional prototype;

- the domain specialist can manipulate the conceptual data design and the requirements;
- the design can be directly and dynamically validated since the beginning of the development process;
- the conceptual data design creation and the system requirements exploration can happen simultaneously. The prototype encapsulates the data model and the system requirements, eliminating unnecessary conceptual data design creation postponements;
- the domain specialist plays the role of a problem solver, not only to explore requirements, but also to conceive the conceptual data design.

The work allow the possibility of automatically generating an initial version of the physical database design. A feature typically available in most commercial database design tools. This is essential to reduce the operational work of mapping the conceptual and logical designs to the physical model. Our tool currently generates SQL *scripts* allowing the creation of relational databases in database management systems (DBMS).

The approach can even be extended by creating new annotations to represent different data abstractions, or even by improving the developed annotations. More information about it as well as our prototype tool source code can be found at `http://www.ime.usp.br/~jef/mbroinizi/`.

Finally, many other interesting aspects have been observed, but they are beyond the scope of this work. Some of them are: Naked Objects framework extensions to explore functional requirements; more complex data generalization hierarchies representations (including support to multiple inheritance); and extensions of our tool to generate the physical design for other models, such as EJB [9].

6. ACKNOWLEDGMENTS

This work has been supported by FAPESP (São Paulo State Research Foundation) and CAPES (Brazilian Coordination for Improvement of Higher Level Personnel). The paper presentation at the Symposium is supported by UOL (Universo Online). Also, we would like to thank everybody who helped us in this work, specially Prof. Alberto H. F. Laender for his important contribution and motivation to improve this work.

7. REFERENCES

[1] Agile Alliance. www.agilealliance.org/. Last seen on October 29, 2007.

[2] Agile Manifesto. http://www.agilemanifesto.org/. Last seen on October 29, 2007.

[3] S. Ambler. *Agile Database Techniques*. Wiley Publishing, Inc, 2003.

[4] Java 5.0 Annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. Last seen on October 29, 2007.

[5] M. R. B. Araujo and C. Traina Jr. et al. Editor de Esquemas com Suporte para Hierarquia de Classificação. In *XIII Simpósio Brasileiro de Banco de Dados*, pages 135 – 149, Maringá, Brazil, Oct. 1998.

[6] P. P. Chen. The entity relationship model - toward an unified view of data. *ACM Trans. on Database Sys.*, 1(1):9, Mar. 1976. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[7] E. F. Codd. Extending the relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):394–434, Dec. 1979.

[8] A. S. da Silva, A. H. F. Laender, and M. A. Casanova. On the relational representation of complex specialization structures. *Inf. Syst*, 25(6-7):399–415, 2000.

[9] Enterprise Java Beans. http://java.sun.com/products/ejb/. Last seen on October 29, 2007.

[10] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 4th edition, 2004.

[11] H. F. Korth and A. Silberschatz. *Sistemas de Banco de Dados*. MAKRON Books do Brasil Editora Ltda, McGraw-Hill Ltda, 1989. Portuguese translation of Database System Concepts.

[12] A. H. F. Laender, M. A. Casanova, A. P. de Carvalho, and L. F. G. G. M. Ridolfi. An analysis of SQL integrity constraints from an entity-relationship model perspective. *Information Systems*, 19(4):331–358, 1994.

[13] Naked Objects. http://www.nakedobjects.org/. Last seen on October 29, 2007.

[14] R. Pawson. *Naked objects*. PhD thesis, University of Dublin, Trinity College, 2004.

[15] R. Pawson and R. Matthews. *Naked Objects*. Wiley and Sons, 2002.

[16] R. Pawson and V. Wade. Agile Development Using Naked Objects. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*, volume 2675 of *Lecture Notes in Computer Science*, pages 97–103. Springer, 2003.

[17] R. Pawson and V. Wade. *Agile Development Using Naked Objects*. Springer Berlin / Heidelberg, 2003.

[18] T. J. Teorey. *Database Modeling & Design*. Morgan Kaufmann Publishers, Inc., 1999.

[19] C. Traina Jr., A. J. M. Traina, and M. Biajiz. O Papel da Abstração de Instanciação em um Meta-Modelo de abstrações para BDOO. In *Anais do IX Simpósio Brasileiro de Banco de Dados*, pages 173–187, São Carlos, Brazil, Sept. 1994.

[20] UML. http://www.uml.org/. Last seen on October 29, 2007.