Transformation of Binary relations into Associations and Nested Classes

Jamal Said, Eric Steegmans
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven
Belgium
Tel: +32-(0) 16-32 76 56
Fax: +32-(0) 16-32 79 96
E-mail: jamal.said@cs.kuleuven.ac.be

Abstract
In object-oriented paradigm, as the complexity of the software system increases, it's cost to develop and to maintain goes exponentially. This complexity emerges from the continuous evolution in the software systems to cope with changing requirements. Throughout our study we found that maintaining traceability between the evolved software processes (e.g. analysis, design) in parallel with examining the ultimate software quality factors needed is an efficient way to cope with this crucial problem. To maintain traceability requires keeping the line between the analysis and the design phase crisp and distinct. This line can be defined by performing an active transformation of the elements (i.e. classes and relations) of the conceptual model to produce optimum design model. This transformation requires the structure and the semantics of the predefined elements to be kept consistent with their equivalent ones in the design model. The transformation process ends up with an optimum design model, thereby reducing complexity and finally reducing the cost.
In this paper we will show two transformations for a simple conceptual model consisting of three inter-related classes having a binary relation. Each of these transformations satisfies particular software quality factor(s), from which the software engineer can choose the one that matches the system intended functional requirements. The added value of this approach is that less manual optimization is required and high maintenance is achieved.

1. Introduction

In the mid-nineties the idea of design patterns started to attract considerable attention in the area of object-oriented software development. Design patterns [1] are architectural ideas applicable across abroad range of application domains; each pattern enables the software engineer a solution to a certain design issue. In fact, the patterns developed in the past few years are only incremental additions to the software professional's bag of standard tricks [2]. To put it more precisely, the underlying representation of a design pattern and of its application, and the binding between these two levels is not exactly defined and thus can be interpreted in different ways [7]. Other researchers [3] have followed the qualitative design trends, which lead to designs that exhibit a desirable quality and forms a movement from bad design model to good design model. These two approaches (i.e. design patterns and the qualitative heuristics) have common basis since both strive to reuse general knowledge rather than domain-specific code. Although these two approaches show interest to software engineers, they lack the ability to keep traceability and maintainability between the analysis and design models.
The analysis phase usually ends up with the conceptual model, in which the external world with corresponding classes and objects is represented. In the traditional approach where we have a chasm between analysis and design, the major input to the design process is the Software Requirements Specification *document*. Because incompatible and non-integrated notations are used from analysis to architectural design, a lot of rework is required, discovering the same ambiguities again, maybe committing the same errors again, and (hopefully) correcting the same

errors again. This paper proposes an approach where the architectural design model, doesn't start off with an empty design, but it starts off with a design model, which is a copy of the analysis model in the design notation. This model represents a complete description of the way the system could work, covering all functional requirements. It does not represent a solution that meets all the other requirements. It is then an approach to transform analysis model into a description of the way we want the system to work. This approach can be worked out by considering the elements of the conceptual model as a collection of simple conceptual model's fragments and based on object-oriented concepts and the software quality factors, these fragments are transformed into design model. The transition from the conceptual model to the design model is often an iterative process; thus it is crucial to be able to develop a framework that performs a reliable and convenient transition between the two models.
Currently, software developers based on the conceptual model try to accomplish some actions manually, which in most cases leads to a big distinction between experienced and inexperienced developers and increases the cost of the software system due to maintenance. Given the fact that software engineering is aiming at building robust and reliable software systems, an approach that supports modeling and provides insights into understanding the software requirements and the software design is crucial. This approach should not restrict the software engineer to a particular phase of the software life cycle but it maintains link between the early phases (analysis and design).
Without necessarily inhibiting choices of the design, taking a copy of the analysis model as an initial design model is likely to enable smother transition from requirement modeling to design. It also prevents unnecessary and non-justified differences between the analysis and design model. It guarantees a better traceability between the analysis model and final design model. It also makes design choices more explicit, as these are highlighted as justified changes between the analysis and the design model.

2. Binary Relations at the Level of Analysis

The early stages of object-oriented analysis is mainly concerned with specification of the objects that are relevant to the application being developed, then comes the refinement step in which the relationships among those corresponding objects are examined in parallel with the study of the events by means of which these relationships are manipulated.
In our view relationships are considered as characteristics of the involved objects. Consequently, relationships of the same sort are grouped in a class. As an example, relationships between persons and companies, expressing that companies employ persons, first of all lead to the introduction of a class of employments. As a result a relation is said to refine objects of a given class, a refinement expressing that these objects cannot exist without being related to objects of the classes participating in the given relation [8, 9]. In our example, a relation will be introduced refining the objects of the class of employments, in order to express that no employment can exist without being related to a person on the one hand, and to a company on the other hand. Such kind of relations is called binary relations which involve two participating classes and one refined class. For example in banking application both classes persons and banks as illustrated in Figure1 are known as the participating classes and the class of accounts as the refined class.

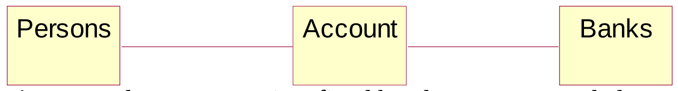

Figure 1: class accounts is refined by class Persons and class Banks.

As mentioned before, any specified relation between objects is complemented with a specification of operations for manipulating those involved objects.
For classes refined by a binary relation, at least a constructor, destructor and two queries must be introduced. The constructor will initialize the binding of the new refined object with the given

objects of the two participating classes; the inspectors will return the objects of the two participating classes involved in the refined relation. Furthermore, the refined class may introduce mutators for changing the binding of refined objects to some other objects of the participating classes. For example constructing a new account requires specifying the Person that will hold this account and the grantor (bank) that will grant this account. Furthermore, a destructor for closing the given account, a query (e.g. getBank, getPerson) for retrieving the owner (getPerson) and the grantor (getBank) of this Account, and a mutator (e.g. transferTo) to transfer accounts from one person to the other is required. Besides the constraint of mutability, constraint of multiplicity is also important at the early stages of the analysis. For classes refined by a binary relation, the multiplicity specifies how many objects of the one participating class can be associated at most with the same object of the other participating class through objects of the refined class. The resulted structural and behavioral aspect of the pattern shown in Fig. 1 is illustrated in Figure 2 below.

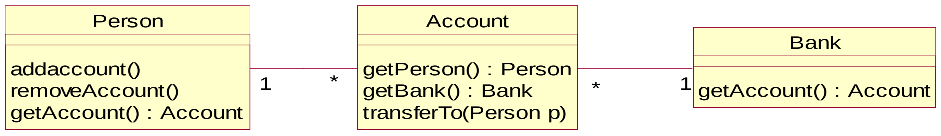

Figure 2: Structural and behavioral aspects of three classes involved in a binary relation.

3. Transformation of binary relations

During analysis the software engineer focuses on the issue of specifying the needed objects for the system to meet its requirements and lining these objects with appropriate relationships to construct a meaningful and complete conceptual model. In other words, the software developer is only interested in which objects are needed not how these objects should be implemented, the later is the subject matter of the design phase which will give the description of the involved objects and relationships between them. The description of the classes and their relations are prime items of the design model.

This paper presents a transformational approach to object-oriented design. Basically, a design model is obtained by transforming fragments, as they can be observed in conceptual models. Because a single fragment can be designed in many different ways, the designer chooses the most appropriate one, based on quality factors for the ultimate system being developed.

This paper discusses transformations for a simple conceptual model defining the refinement of a class by means of binary relation. For a pattern consisting of binary relation (Fig 3), there exist different alternatives to transform it to design elements. In this paper we will focus on the association and nesting transformations.

3.1 Association Transformation

The binary relation involves two participating classes and a refined class can be design in terms of an association between the refined class and the participating classes. Associations represent relationships between instances of classes (e.g. a person holds accounts in Banks; a bank grants accounts to person From the conceptual perspective, associations represent conceptual relationships between classes. In Figure 3, the diagram indicates that an

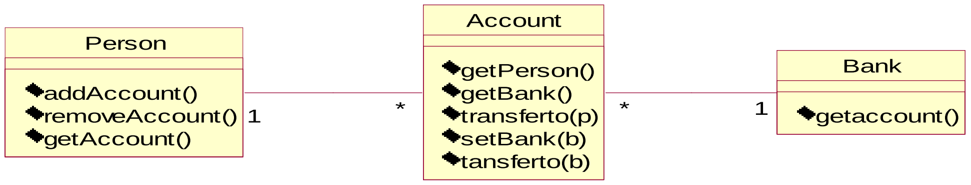

Figure 3: Class diagram with association relationships.

account has to reference one person and must be granted by one bank. As far as the multiplicity is concerned, which is an indication of how many objects may participate in the given relationship. In Figure 3, the * between person and accounts indicates that a Person may have many accounts associated with it; 1 indicates that an account related to only one person. The multiplicity between accounts and Bank indicate that a Bank grants many accounts and an account has to be granted by only one Bank.

- Within the specification perspective, associations represent responsibilities. Figure 3, implies that there are one or more methods (i.e. getAccount) associated with Person that will tell us know what accounts a given Person is holding. Similarly, there are methods (getPerson, getBank) within Account that will let us know which Person holds this account and which Bank grants a given account
- The given structure explained above would be transformed into design taking into consideration both structural and behavioral aspects defined at the level of analysis.
- Because of the property of existential dependency-accounts cannot be created without being attached to a Person on one hand and to a Bank on the other hand-- the construction of objects of the refined class (Account) must initialize references to objects of the participating classes (Person and Bank). Objects of the participating classes (Person, Bank), on the other hand, can exist without being involved in associations with objects of the refined class., Consequently, the constructor at the level of the participating class initializes a new object without any association to objects of the refined class.
- The destructor for the refined class objects as it is specified at the level of analysis is transformed into the method removeAccount. Notice that the reference to the destroyed object is removed from the participating object.
- The existential dependency should also be considered when destroying the participating objects. Before any participating object is destroyed one must check whether this object is holding references to a refined object or not. If so all these refined objects must be destroyed beforehand. For example, when a Person is removed from a Bank, it means his account will also be.
- The inspectors defined at the level of the participating classes are transformed into the method getAccount applicable to objects at the level of design. Notice that the method returns an array in which references to all the Account's objects are stored.
- Similarly the inspectors defined at the level of the refined class is transformed into the method getPerson and getBank applicable to Account objects. This method and will return the Person and the Bank attached to this account.

The mutator defined at the level of analysis is transformed into the method transferAccount. This method transfers this Account to the specified Person and Bank.

Below we will show some methods with their specification implemented in Java. Notice the specification's notation used here is widely used in the literature [11].

```
import java.util.*;
/**
 * A class of person.
 */
public class Person {
  /**
   *  Initialize a new Person with no Account nor bank objects attached to It.
   *  @post   No Bank object and account-objects are attached to the new person.
   *        | new.getAccounts().size()= 0
   */
  public Person()
}
//Definition of the refined class Account
import java.util.*;
```

```
/**
 * A class for dealing with accounts attached to a Person and a Bank
 * @invar  An account must all times be attached to a Person and a Bank.
 * @invar  The Person and the Bank to which this account is
 *         attached, must reference back to that account.
 *       | getPerson().hasAccount(this)
 *        | getBank().hasAccount(this)
 */
public class Account {
  /**
   * Initialize a new account attached to the <person> and the <bank>.
   * @param  <person>
   *         The Person to which the new account will be attached.
   * @pre    <person> must be effective
   *        | person <> null
   * @post   The new account is attached to <person> and vice versa.
   *        | (new.getPerson() = person )
   *           and (((Person)((new person).getAccounts()).contains(this)) = true )
   * @param  <bank>
   *         The Bank to which the new account will be attached.
   * @pre    <bank> must be effective
   *        |bank <> null
   * @post   The new account is attached to <bank> and vice versa
   *        | (new.getBank() = bank )
   *        |(((Bank)((new bank).getAccounts()).contains(this)) = true)
   */
  public Account( Person person, Bank bank)
  /**
   * Transfer the new account to specified person
   * @param  <person>
   *         The specified person to become participant to this account
   * @pre    The specified person must be effective
   *       | person <> null
   * @post   The specified person is associated with this account
   *       | new.getPerson() = person
   * @post   This Account is no longer referenced by the person
   *         to which it was associated before.
   *       | for each i in 0..(this.getPerson()).getAccounts().size() - 1:
   *      (this.getPerson()).getAccounts.elementAt(i) != this
   *          and (this.getPerson()).getAccounts.size()
   *           =(this(this.getPerson())).getAccounts.size() –1
   */
  public void transferAccount(Person person)
}
//definition of class Bank
/**
 *  Definition of participating class Bank
 */
public class Bank {
  /**
   * Initialize a new bank with no accounts attached to it
   * @post No accounts attached to the new bank
   *       | new.getNbAccounts() = 0
   */
  public Bank()
  }
```

Implementation1: Implementing association transformation.

With the association transformation the software engineer selects for the quality factors flexibility, and re-usability over efficiency and simplicity.

- Limiting each of the involved classes to a specific area of interest (i.e. cohesion) highlights flexibility. Furthermore, flexibility is stressed by allowing future modifications to the software system. As far as coupling is concerned this transformation strives to have high coupling by allowing the components to cooperate via message passing.
- This transformation is considered to be highly reusable since most of the structural and behavioral aspects of the classes specified at the level of analysis are transformed at the level of design with limited loss of information.
- As far as the efficiency is concerned this type of transformation is not the most efficient one in terms of time and space since the memory requirement is high. Part of the objects of the involved classes needs a separate location in memory, which in turn affects the performance of the software system . The creation of new objects of the classes and the message passing between them requires the execution to take more time than if they were integrated in one class.
- Simplicity is not supported by this transformation since it requires message passing between objects of the classes involved. The message passing might lead to inconsistencies, if bi-directional associations are not designed and implemented with great care.

3.2 Nesting Transformation

Nesting transformation occurs when one class is fully defined inside the other the concept which known in Java as inner classes. Inner classes are powerful abstraction mechanism [5] that facilitate much more convenient and manageable software than it would be when using only top-level classes. They are remarkable as they allow to group classes and control the visibility of one within the other.
Classes with binary relation can be transformed by defining one class inside the other. For example, Figure 4, shows the participating class Bank having association with class Account which is nested inside the participating class Person. Class Person serves as the outer class through which the refined class Account (inner class) can be accessed. Notice also that the outer class is responsible for creating and the Account objects.
The account objects are created by applying the method openAccount to Person objects. This method when applied to person object will also initialize a bank object with the created Account object due to the existential dependency. Notice that since the creation of the accounts depends on the person objects then the accounts will automatically store implicit references to person objects. Therefore, an object of the refined class is directly associated with the object of the outer class; objects that created them. As a result the inner class object has direct access to the instance variables of the enclosing class object. Notice that the compiler does the implicit reference to the outer class objects itself. Concerning mutation Accounts cannot be transformed at the level of design since the refined objects are nested in person objects, which are designated, immutable.

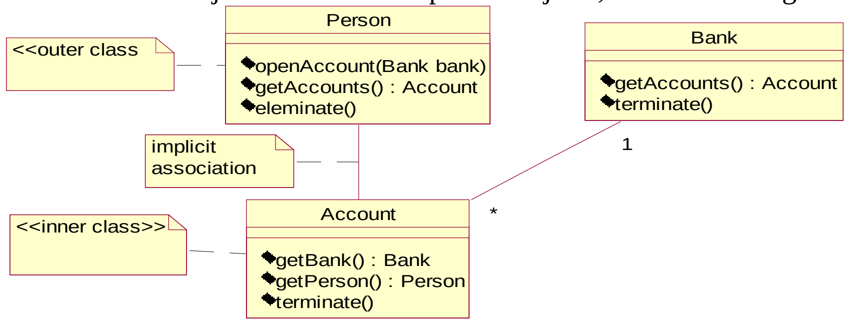

Figure 4: Account class nested in Person class and has an association with class Bank.

```
/**
 * The participating class Person.
 */
 public class Person {
   /**
    *  Initializes a new Person with no Account nor bank objects
    *  attached to it.
    *  @post   No Bank object and account-objects are attached to
    *          the new person.
    *        | new.getAccounts().size()= 0
    */
   public Person()
   }
   /**
    * Definition of the inner class Account.
    */
   public class Account {
      /**
       * initialize a new Account
       * @post  Bank object must also be initiated
       *       | this.getBank() == bank
       */
      Account(Bank bank)
      /**
       * Terminate this account
       * @post This account is terminated and detached from its participating
       *       objects.
       *       |!((new getPerson()).getAccounts().contains(this)
       *       |!((new getBank()).getAccounts().contains(this)
       */
      public void terminate ()
    }//end of inner class
  /**
   * Creation of the account object
   */
  public void openAccount(Bank bank)
  }//end of outer class
/**
 *  Definition of participating class Bank
 *  Class bank has an association relationship with class Account
 */
public class Bank {
  /**
   * Initialize a new bank with no accounts attached to it
   * @post No accounts attached to the new bank
   *       | new.getNbAccounts() = 0
   */
  public Bank()
}
```

Implementation 2: Implementing nesting transformation.

The added value to the object oriented software design by nesting transformation is that it increases modularity as will as simplicity over efficiency.

- Modularity is the term that covers reusability and extendibility. Nesting transformation helps in making these two classes easy to change. In association when one of the two associated classes is expected to change we must take the navigability under consideration whether the involved class is bi-directional or unidirectional, whereas, in nesting we know already that the inner class objects have implicit references to their outer ones.

Concerning the reusability, nested transformation is highly reusable particularly for applications where accessibility constraints are important

- This transformation is considered to be simple since it decreases the number of classes developed at the package level. Which make the model easier to understand and maintain, also it limits the number of message passing between the associated classes
- As far as efficiency is concerned it helps in time efficiency because both inner and outer classes are stored in one file which makes message passing requires less time than if they were stored in two separate files. However this transformation doesn't help so much in space efficiency since both the classes are stored in different places in memory

4 conclusion

In this paper we have shown that designing convenient and transparent software system can be handled easily by keeping the line between the design and the analysis definite and distinct. This line can be defined by performing an active transformation of the conceptual model's elements and relations (i.e. fragments) to produce a design model that perform the system intended functionalities. We have seen that this technique offers the user to select among different transformations the one that meets the design goals. As a result, this new technique doesn't require the software engineer to optimize the design model, which is lacking in the current methodologies. Furthermore, this technique establishes a strict correspondence between conceptual models at the level of analysis and design models at he level of design, which results in high maintenance throughout the software system.

In this paper we have discussed binary relations and their transformations, and in the future work this will be extended to cover the classes involved in generalization specialization and statics.

5. References

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object Oriented Software*. Addison- Wesley.

[2] Bertrand Meyer 1997. *Object Oriented Software Construction*. Prentice Hall.

[3] Arthur J. Riel (1996). *Object Oriented Design Heuristics*. Addison Wesley.

[4] Rumbaugh, J. Jacobson, I. & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.

[5] Martin Fowler with Kendall Scott (1998). UML Distilled : Applying the standard object modeling language. Addison Wesley.

[6]Charles Richter (1999). *Software Engineering Series. Designing Flexible Object –Oriented System with UML*. Macmillan Technical.

[7] Gerson Sunye, Alain le Guennec, and Jean-marc Lezequuel. *Design Patterns application in UML. ECOOP' 2000 – Object oriented Programming 14th European Conference, Sophia Antipolis and Cannes, France , , volume 1850 of lecture notes in Computer Science, pages 44 –62*. Springer – NY, June 2000.

[8] Van Baelen, S., Lewi, J., and Steegmans, E., *Constraints in Object-Oriented Analysis and Design, Technology of Object-Oriented Languages and Systems TOOLS 13*, eds. Magnusson, B., Meyer, B., Nerson, J.-M., and Perrot, J.-F., Prentice-Hall, Hertsfordshire, UK, 1994, pp. 185-199.

[9] Van Baelen, S., Lewi, J., Steegmans, E., and Swennen, B., *Constraints in Object-Oriented Analysis, Object Technologies for Advanced Software*, LectureNotes in Computer Science 742, eds. Nishio, S., and Yonezawa, A., Springer-Verlag, Berlin, D, 1993, pp. 393-407.

[10] Said J., Steegmans E., Transformation of Unary Relations: Proceeding ICSSEA 2000, 13th International Conference on Software & System Engineering and their Applications , Paris – France (2000).

[11] Bruegge B., Dutoit A.H., Object-Oriented Software Engineering. Prentice-Hall, Inc. 2000, pp. 239.