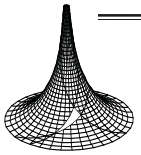

6/1 (2008), 83–109
tmcs@inf.unideb.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Decision based examination of object-oriented programming and Design Patterns

Szabolcs Márien

Abstract. On the basis of our examination experience of Design Patterns the existing interpretations and descriptions of Design Patterns do not realise a clear and understandable answer for their aims. The reason for this is that the existing interpretation of the object-oriented paradigms is used for their description and formulation. In order that clear answers could be found for the aims of using Design Patterns, a new conception of their interpretation has to be established. In order to create a new conception, we have to analyze object-oriented paradigms.

According to our new conception the object-oriented methodology is based on the elimination of decision repetition, thus sorting the decisions to class hierarchy, with the help of which the data structure and methodology of decision options can be determined by the subclasses of the given class. Sorting the decisions and decision options to a class and its subclasses only the first decision case will be executed, which will be archived and enclosed by instantiation of one of the subclasses. For the following decision cases the archived decision result can be used without knowledge of which decision option was used, so to say which subclass was instantiated, because it is enclosed by using the type of the parent class.

The aim of the object-oriented technology is the elimination of decision repetition, which can be realized by sorting the decisions. The derivations are the abstract definitions of decisions, so the derivations can be interpreted as decision abstractions. The Design Patterns offer recipes for sorting the decisions. With the help of the decision concept the aim of Design Patterns can be cleared and a more natural classification of Design Patterns can be realized.

Key words and phrases: object-oriented paradigms, inheritance, Design Patterns.

ZDM Subject Classification: P50.

1. Motivation

The reduction of design failures can be achieved by experience in design.

The Design Patterns are realized as a result of collecting the design experience. The Design Patterns give recipes for the designers how the designing cases can be solved appropriately and realize the professionally accepted answers for the similar designing cases.

The professionally accepted Design Patterns are specified in [1].

In order to be able to use the experience gained from the Design Patterns, we have to understand the Design Patterns. The formalizations of Design Patterns—which try to represent their different aspects—aim at supporting their understanding. The formalizations are based on different description solutions with graphical presentation concepts (LePus [4], PDL [10]).

In [1] the description of Design Patterns is based on natural language, examples and OMT diagrams [11]. The description of Design Patterns by natural language and examples is not accurate [4, 5], so the essence of Design Patterns is often missing from their descriptions. Therefore the profession has published numerous solutions for a more accurate and formalized description of Design Patterns. (LePUS [4], DisCo [5], BPSL [3]). But the new formalizations do not engage in the interpretation of Design Patterns, so the existing interpretation is untouched, therefore the aims of Design Patterns remain unclarified.

What is the reason for the lack of answers to the aims of Design Patterns? The answer can be that the object-oriented paradigms, based on their present interpretations, obstruct the extended examination of the object-oriented methodology. According to the present interpretation of the object-oriented paradigms there are no clear answers to the aims of Design Patterns. Thus the present interpretations of object-oriented paradigms (Inheritance [9, 12, 13, 15], Polymorphism [9, 12, 15], Encapsulation [9, 12, 15], Message-passing [9, 12, 13, 15]) and the present interpretation of Design Patterns [1] must be reconsidered.

In [7] a new concept is invented for the interpretation of Design Patterns, which states that the aim of Design Patterns is the elimination of the repetition of the code and data redundancy. This new concept is proper as it tries to reinterpret Design Patterns. However, it does not abstract from the object-oriented paradigms, therefore the given answers are not holistic.

In this paper a new interpretation of the object-oriented paradigms is described which can help us comprehend the aims of Design Patterns. According

to this concept a new classification of Design Patterns has been realized, which is more natural than the existing ones.

2. Introduction

In this paper a new idea is shown, with the help of which the aims of Design Patterns can be clarified. Accordingly the object-oriented paradigms will get new interpretations.

What do the decisions of the program code decide? The decisions decide the appropriate methodology and the data structures, so the data structures and the functionalities are specified in the decision options of the decisions. The decision options are the optional facilities of the decision. The decision option consists of a data structure and methodology and a decision predicate by which the appropriate decision option is determined in the decision cases. The main concept of the object-oriented methodology is the elimination of the decisions' repetition by sorting them to a "common place". This "common place" is a class with its subclasses, so the decision repetition can be eliminated by the class hierarchy, which is a more abstract definition of the decisions.

After sorting the decisions, the decision is executed only once about the necessary functionality and data structure. Based on the instantiation of the subclass with the appropriate functionality and/or data structure the archiving of the decision is realized. The result of the decision (the archived decision) as an instance of the appropriate subclass can be used at the other decision places (henceforth: decision cases) without knowing anything else about it. Accordingly the decisions can be enclosed in class hierarchy.

The example shows that two decisions' decision options are organised in the same class hierarchy by sorting decision—Purchase. (The contraction conditions of two decisions' decision options are described below.) Using the Purchase class hierarchy the decision can be used more times inclosing it into the purchase object with the parent class type of the class hierarchy. First, the decision is used for setting the decision specific data (purchase.setPurchaseInfo()) and then, for printing them (purchase.printBill()). If decision option specific data structure or methodology is required after archiving decision, type-casting has to be used for achieving the subclass (according to the decision option) specific data structures and methods.

The decision cases are important parts of the programs where the appropriate decision option can be decided using the actual values in the decision predicates.

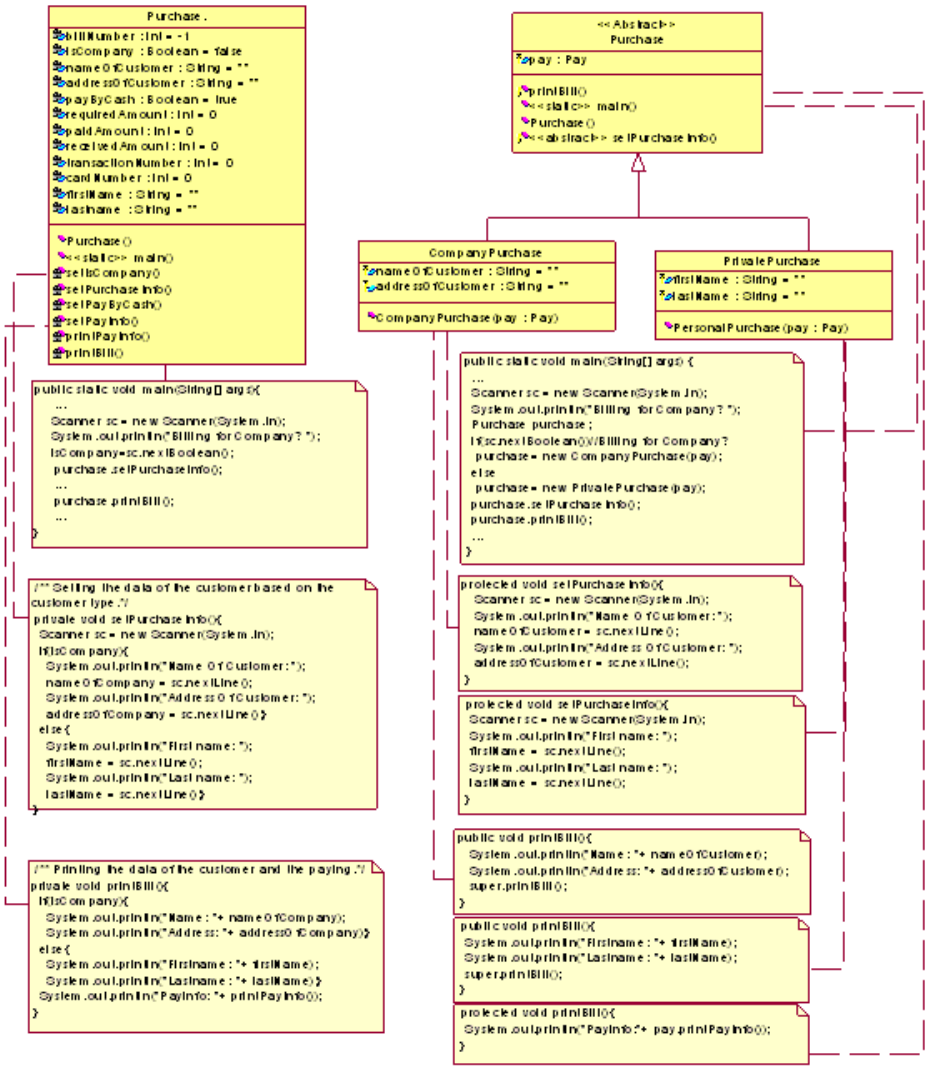

Diagram 1. Sorting decision into class hierarchy

In order that a program would be good structured, we should note the following:

- The methodology and/or the data structure of the decision options have to be defined just once, so the code of the decision options will be defined just once, except when the sorting of the decisions is impossible. It is important to consider handling, because the introduction of the new decision option can only be solved easily, if it is built in at just one place in the program.
- The decisions should not reoccur, so a decision—during the same running—can be executed just once. The elimination of the decision repetition has two aspects:
  - The result of the decision—as the data structures or/and methodology of the decision options—can be used several times.
  - The result of the decision can be used several times afterwards, but every time a new instance of the structures or/and the methodology of the decision options is created. The archived decision can be used later for creating an instance of the decision options.
- The decision predicates of decisions are equivalent, but altering in their decision options' definitions should not reoccur. (The decisions predicates are equivalent if the evaluation of predicates is equal in every state.) This differs from the previous case, because although the decision predicates are equivalent, the methodology and/or the data structures of the decision options are different. In these cases the decisions can be contracted too, so the definitions of the different decision options can be defined by contracting them in the same class hierarchy according to the decision predicates.

The Design Patterns give us recipes for accomplishing the above listed requirements of the good structured programs in order to reduce the decision repetitions.

The Design Patterns show us how the decisions can be archived and/or related.

According to the previously determined two main concepts of Design Patterns, there are two groups of them:

- Decision archiving Design Patterns: If the data structure and/or the methodology of decision options of a the realized decision is used more times, the inheritance/derivation as the main concept of the object-oriented technology can assure using the result of the decision more times by enclosing the decision to the class hierarchy (to the class and its subclasses). The enclosed decision by derivation can be used more times by aggregation in a certain scope. If new instances of the appropriate data structure and/or methodology have to be created according to the decision for every decision cases,

the archiving Design Patterns are used. The types of the products' structure determine which type of archiving Design Pattern can be applied. It means that the associations between the data structures and/or the methodology of the decision options determine the type of the usable Design Pattern.

- Design Patterns, which determine the relations of decisions: The structures of the related decisions and the relations of them determine which Design Pattern can be used. Therefore the second group of Design Patterns can be realized based on the relations of Design Patterns.

The two main groups of Design Patterns can be classified separately.

In the next part of our paper the new interpretations of object-oriented paradigms will be described based on new conception (Section 3). In Section 4 the basis of decision based theory can be seen. Based on this concept, a new classification of Design Patterns can be realized (Section 5). In the final part of the paper, in Section 6, an example shows how decision redundancy can be eliminated by applying Design Patterns.

3. New interpretation of the object-oriented paradigms

"The object-oriented programming is a programming methodology. Programs —which based on it—organize the collaboration of objects, which are instances one of the classes. The classes are parts of hierarchies, which are built by inheritance connections." [15]

The most important object-oriented paradigm is inheritance. Inheritance can improve the design logically and free it from code redundancy. [7]

In order that the analysis can be realized based on the decisions, it is important, how the basic paradigms of the object-oriented technology (inheritance [9, 12, 13, 15], polymorphism [9, 12, 15], encapsulation [9, 12, 15]) and its basic tools (class hierarchy, aggregation) can be joined to the decision based concept.

Inheritance as Decision Abstraction

The inheritance means that the data structure and the methodology—which are defined in a class—can be inherited by its subclasses. The subclasses can define new data structures and methods as complements of the inherited properties [13, 14] and can overwrite the inherited data structure and methodologies.

The decision can choose the running program code and the data structure. In order that a decision can be archived, it has to be sorted, which means that the data structure and methodology of decision options have to be defined in a class hierarchy, as a parent class and its subclasses. The derivation/inheritance ensures the enclosing and archiving of decision to the class hierarchy with the help of which the definitions of the decisions can be contracted and the decision repetitions can be eliminated.

According to this interpretation the inheritance—the class with its subclasses —is the abstract form of the decision.

If the decision is defined in a class hierarchy, the following is realized:

- Elimination of the code repetition, which defines the decision options, so the conditions of the decision options can be defined just once.

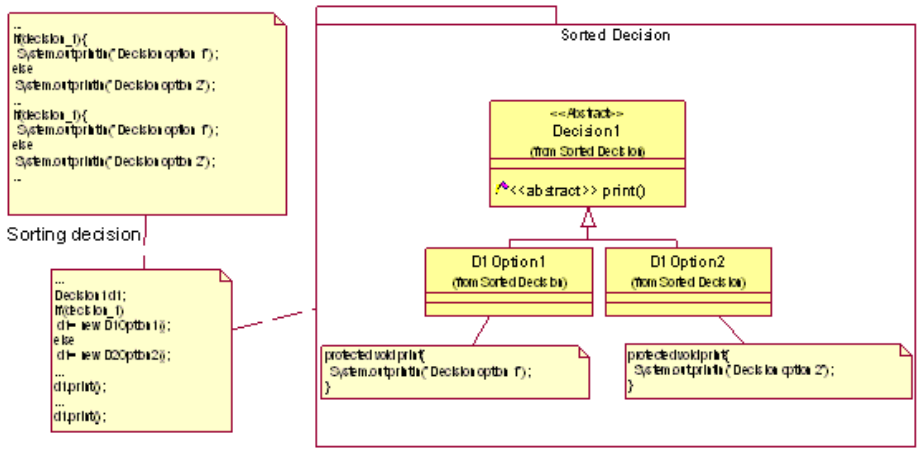

Diagram 2. Elimination of code repetition by decision sortingy

In Diagram 2 the definitions of decision options of decision_1 are duplicated, which is eliminated by sorting them into a class hierarchy.

- Archiving the decision, with the help of which the result of the decision can be used later on. Unless the required data structure or methodology is specified by just one of the decision options.
  In Diagram 1 the purchase object is created the type of which is Purchase parent class, and it archives the decision about the purchase type (company or private purchase). The archived decision about purchase is applied twice.

(purchase.setPurchaseInfo(), purchase.printBill()). The purchase object is the instance of one of the subclasses of the Purchase parent class (CompanyPurchase, PrivatePurchase), by which the decision is archived and the decision option specific data structure and methodology are accessible.

- Enclosing the decision. The result of the decision is not known in the next decision cases. Except when the required data structure or methodology is specified by just one of the decision options.
  In Diagram 1 according to the decision about purchase type one of the subclasses (PrivatePurchase, CompanyPurchase) is instantiated (purchase object) enclosing the decision. The type of the purchase object is the Purchase parent class. In the following decision cases the decision is enclosed, so there is not knowledge about the current decision option (about the type of the instantiation). The type (one of the subclasses) of the instantiation realizes the current decision option determining the current methods and data structures of one of the subclasses, which are actually run by the purchase.setPurchaseInfo(), purchase.printBill() method invocations, as it can be seen in the following example code:

```
/** Setting the customer's type. */
System.out.println( "Billing for Company? " );
Purchase purchase;
if ( sc.nextBoolean() ) // Billing for Company?
    purchase = new CompanyPurchase( pay );
else
    purchase = new PrivatePurchase( pay );
```

- By the introduction of the new subclass, the decision options can be extended easily. By creating a new subclass, just the first decision case has to be fit for handling the new decision option, because the decision will be enclosed on the next occasions. Unless the required data structure or methodology is specified by just one of the decision options.

As can be seen, if the data structure or/and methodology is specified by just one of the decision options, the advantages of the decision sorting can be realized partly. The forceful usage of the polymorphism can realize the advantages of the decision sorting as inheritance completely.

Polymorphism as Decision enclosing

Polymorphism means that the classes' methods can be overwritten by their subclasses, so the method—which gets the control—is selected just in runtime (Late Binding) [12]. Late Binding—using another terminology—means that an object sends similar messages to different objects (an instance of a class or its subclasses) and a different code will be executed [9]. The message-passing (using Smalltalk interpretation it is message-passing, but based on C++ terminology it is method calling [9]) means that one object gets another object to execute a method [13]. Late Binding depends on the programming-language or it is optional (C++), because the resource effort is too large, but the maintenance of the code will be reduced [14]. The polymorphism can increase the reusability, because the introduction of the new subclasses to the program is easier [14].

If the decision is realized in the first decision case, one of the subclasses will be instantiated based on the chosen decision option. The instance of the appropriate subclass archives the decision and the visible type of the instance will be the parent class of the subclass. With the help of this the enclosing of the decision can be realized, because the result of the decision can be used without of the knowledge of the decision on the next occasions.

In Diagram 1 after sorting decision the purchase object archives the decision instantiating one of the subclasses (PrivatePurchase, CompanyPurchase) according to the first decision case. The instantiation determines the subclass the methods of which are executed the following method invocations (purchase.setPurchaseInfo(), purchase.printBill()). The polymorphism ensures the enclosing of decisions, because we don't have to know the subclass—decision option the methods of which are executed the following method invocations (decision cases before sorting of decisions), because the instantiation (the first decision case) realizes the decision enclosing. In the following decision cases the polymorphism means the different methodologies according to the decision options.

Encapsulation

Encapsulation means that the data structure and the methodology are defined together enclosing them in units as objects. The encapsulated data structure and methodology can be defined in the classes, the instances of which are the objects. Based on the encapsulation the modularized construction can be realized, with the help of which there will not be any side-effects in other objects—if the methodology of one of the objects is changed. [9]

The decision options can be defined by data structure and methodology. The decision is defined in a method, if the appropriate If-Else command's blocks define the data structure and the methodology of the decision options. If the decision is sorted defined by the class hierarchy as an abstract form, the decision options are realized by the subclasses. Using this the changing of the data structure and the methodology of the decision option has not got any side-effects in other decision cases and other decision options, so the decision option can define the data structure and methodology by a subclass which encloses them. As it can be seen in Diagram 1, the CompanyPurchase class defines the data structures and methodology of company purchase decision option declaring the nameOfCustomer, addressOfCustomer variables and setPurchaseInfo(), printBill() methods.

Aggregation as dynamic decision embedding

Aggregation is not an Object-Oriented specific concept. If a language supports record structures, it supports aggregation, too. The class hierarchy defines "is a" hierarchy, and the aggregation defines "part of" hierarchy. [15]

The sorted decision can be referred by aggregation. If there is a decision case, where the appropriate decision option is chosen (with the proper data structure and methodology), and next time the operations are executed based on the chosen methodology and data structure, the sorted decision can be used on the next decision cases by aggregation. The result of the decision will be referred by aggregation.

When we talk about aggregation, we have to know that this is the tool of relating decisions. If there are two related and sorted decisions (D1, D2), d1 decision has an aggregation and the type of aggregation is the parent class of the other D2 sorted decision's class hierarchy, the precondition of D1 decision will be extended with the precondition of the D2 decision by an logical "AND" operation.

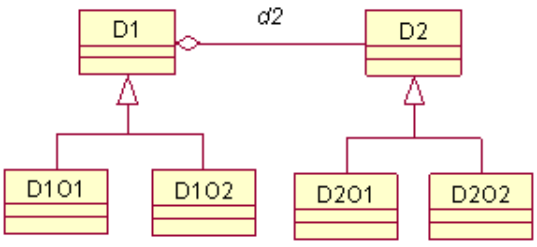

Diagram 3. Sorted decision related by aggregation

Multiple Inheritance as static relation of decision options

If the decision predicates of two decisions' decision options are equivalent, the decision options can be contracted, so the definitions of the decision options can be realized together. This is the static relation of decision options. If the decision options as subclasses of two sorted decisions as class hierarchies are defined by the same class together, it is called Multiple Inheritance. In this case the decision predicates of two related decision options of two decisions are equivalent, or the first decision option's decision predicate with the second decision option's negated predicate are equivalent.

The alternative case of the related decision options is when the decision option refers to another decision option using aggregation (as types of the Adapter Design Pattern [1]).

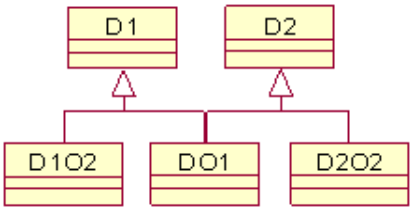

Diagram 4. Relation of decision options by Multiple Inheritance

4. Decision and Sorting decision

Decision

The decision consists of decision options, in which the data structure and methodology is defined.

In order to simplify the problem, every decision consists of two decision options so every decision tree is a binary tree. As every tree can be transformed to a binary tree, this simplification does not restrict the examination. The decision has a predicate (decision predicate), with the help of which the appropriate decision option can be chosen. The decision predicate or its negate applies to decision options, so these are the preconditions of the decision's decision options.

The decisions are in methods (if-else). The data structure and the methodology of the decisions' decision options can be defined in the methods or in the class hierarchy (if they are sorted) referred to by an aggregation.

By every decision:

- Variables are determined — As mentioned, the decisions consist of decision options. If one of the decision option's decision predicate is true, the decision option will get on. Accordingly, the assigned variables of the decision option will be realized with the consequence of other following decisions based on the aggregations in the appropriate decision option. The data structures of the decision options are those variables the data of which are used as the data-source or the state of which is modified. If the decision is sorted, the decision—as class-hierarchy—and the decision options—as subclasses—define common and decision-option specific variables, which are used (according to the previously mentioned case) as data-source or the state of which is modified.

- A methodology is selected — If the decision options are defined in a method, the methodologies of options consist of a sequence of commands. If the decision is sorted, the methodology of decision options consists of methods.

The class hierarchies can be interpreted as decision abstractions. If the decision is sorted, it is defined by a class with its subclasses as a class hierarchy, which is the abstract form of the decision. If there is a variable, its type is the parent class in a hierarchy, it can enclose and archive the decision, because it can store one of the subclasses' instances enclosed by the parent class type. (It is possible that a parent class typed variable does not enclose a decision, so the subclasses of the parent class are not used.)

In the case of sorting decisions to a class hierarchy, the decision will be executed with the instantiation of one of the subclasses, so the appropriate decision option will be selected, the tag objects of which will be initialized as data structure and the methodology as methods of subclasses can be accessed.

In the Diagram 1 after sorting decision the following code contains the instantiation:

```
/** Setting the customer's type. */
System.out.println( "Billing for Company? " );
Purchase purchase;
if( sc.nextBoolean() ) // Billing for Company?
    purchase = new CompanyPurchase( pay );
else
    purchase = new PrivatePurchase( pay );
```

If the type of the purchase is company purchase, the data structure of CompanyPurchase—nameOfCustomer, addressOfCustomer will be initialized and the methods of the CompanyPurcase as subclass specific methodology will be accessible.

The decision's predicate (Diagram 1: sc.nextBoolean(), Diagram 2: decision_1) will determine the appropriate decision option with the proper methodology and data structure by which the variable will be initialized with the type of the appropriate subclass.

Every decision option's precondition is that the decision predicate is true or false in a decision case. If the decision option gets on, that is to say the decision predicate according to the decision option is true or false, the decision is resolved, and a new state is realized by the methodology and data structure of the selected decision option.

The decisions are state-transitions, with the help of which the state-space is reduced, so the possible number of state-rows decreases (the number of available states). The end of the running will be just one possible state, because the programs are deterministic (for the same input we get the same output). The decision options determine the possible directions of the behavior as state rows. With every decision, a new state will be achieved by defining the data structure and methodology of the selected decision option, with the help of which the state-space is reduced.

If the decisions are built in another decision, the precondition of the related decisions will be their preconditions connected by logical "AND": $P_{D_1O_2} \wedge (P_{D_2O_1} \vee P_{D_2O_2}) \ldots$. The following UML diagram [2] can represent the class hierarchy, which shows the previously mentioned related decisions.

Two decision predicates P_{D_1}, P_{D_2} are equivalent $P_{D_1} \equiv P_{D_2}$ if and only if they are equal in every evaluated state.

Decision case

The decision instance—where we execute the decision based on the evaluated decision predicate and initialize the decision option as its data structure and methodology—is based on its definition. The result of the first decision case can be used in the course of the following decision cases. The decision cases are those decision instances, where the decision is executed, archived and the following decision instances, where the result of the decision is reused.

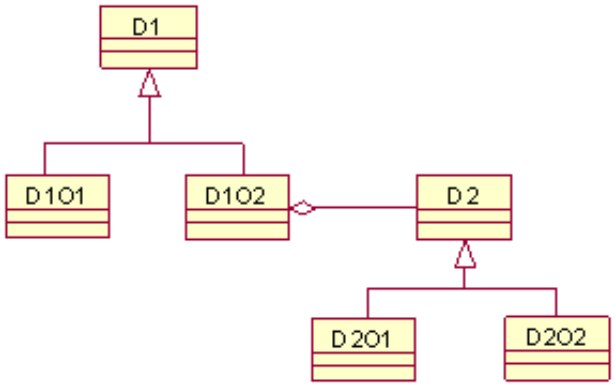

Diagram 5. Object composition—Aggregation—Decision built in another decision—UML diagram

Sorting of decisions

In order to understand the substance of the decision based program designing, the reasons of sorting the decisions have to be collected.

- The methodology and the data structure of the decision options as a code should be defined just one time, except when the elimination of the decision repetition is not possible.
  - If there are similar decision option definitions in the methods, those decision definitions should be contracted by sorting.
  - If there are repetitions in the class hierarchy definitions of the decisions, the similar class hierarchies of decisions should be contracted.
- One decision should be realized just once. Accordingly, the decision repetition should be eliminated. In this case the decisions' predicates are equivalent and the decision options define the same data structure and functionality, which should be sorted in the same class hierarchy, which can enclose these decisions eliminating the decision repetition.
- The decisions with equivalent decision predicates consist of different data structure and functionality. In this case the definitions of these decisions can be contracted in the same class hierarchy in order that the decisions are executed together.
  - If the decisions with equivalent decision predicates are built in the same decision's options, and the functionality and data structure depends on the container decision, the built in decisions can be contracted, but the

contracting definitions have to be differentiated depending on the container decision options.

5. New method of Design Pattern classification

In the Introduction the two main groups of Design Patterns were described, which can be classified as follows:

Design Pattern classification is based on the relations between decisions

In this group the Design Patterns can be realized on the basis of the relations of the decisions.

The state of the decisions—that is, the decision is sorted into the class hierarchy or is defined in a method—is not important considering classification. It means that the detecting of the design patterns has to be based on the decision relations without known constructions of Design Patterns which are defined in [1]. Accordingly, the decision relations determine one of the Design Patterns in the non sorted state of decisions. Based on the structure of the determined Design Pattern the elimination of decision repetition can be realized.

There are Design Patterns in this group, in which the decisions decide between different data structures, and others, which consist of decisions, which decide between different functionalities and there are some patterns, which consist of mixed decisions. The examination of the equivalence of the decision predicates is important in order to realize the classification.

- The decisions, which have equivalent decision predicates should be contracted. Even if the decisions concern different data structures or functionality, the contraction is possible.
    - State, Bridge, Composite, Interpreter, Iterator, Mediator, Observer, Template Method, Decorator, Chain of Responsibility, Command, Strategy.
- The decisions—which have equivalent predicates and their decision options do not define the same data structures and functionality—can not be contracted, if the decisions define basically different aspects of the program. These decision options can be defined by the same class using multiple inheritance or one of the decision options refer to the other decision option using aggregation.

    * Adapter

- The decisions, which have decision options referring to the same data structure and functionality, should be contracted. It is not important if the decision predicates of these decisions are equivalent or not. Naturally, the decisions may concern data structures and functionalities as well,
  - if the contracted decisions define the same data structures:
    * State, Bridge, Composite, Iterator, Mediator, Observer, Template Method.
  - and/or if every contracted decision realizes the options of the same functionalities:
    * Bridge, Composite, Decorator, Chain of Responsibility, Command, Iterator, Mediator, Observer, Strategy, Template Method, Interpreter.
- The case of the complex decisions, as decision contains decision(s):
  - If there are built-in decisions with equivalent decision predicates in the decision options of the sorted decision, but the definitions—as data structures and functionality—of these decision options are different and the built-in decisions' decision options depend on the container decision options, the built-in decisions can be contracted into the same class hierarchy with respect to the container decision option.
    Consequently, the Visitor Design Pattern can be used in the case of a two-level decision hierarchy, where the first level decision options contain second level decisions, the predicates of which are equivalent and depend on the first level decision. In this case the contracted second level decisions can be separated depending on the first level decision.
    * Visitor
  - If the decision options of the decision contain other decisions, which define the same functional and/or data structure options.
    * Bridge, Template Method
  - If the options of the decision contain more decisions. All of these define the same functional and/or data structure options.
    * Observer, Mediator
  - If the decision contains decisions which define such data structures and/or functional options as the container decision with the same, but not equivalent decision predicates.

    * Chain Of Responsibility

  – If just one of the decision options (or all of them as Interpreter) contains an embedded decision which defines such data structures and functionality as the container decision and the decision predicate of the embedded decision and the container decision are the same, but not equivalent.

    * Composite, Decorator, Interpreter

  – If one of the decision options contains an evaluated decision which defines such data structures and functionality as the container decision, but this is a statically evaluated decision.

    * Proxy

Design Pattern classification based on decision archiving

What are the reasons for using archiving Design Patterns?

- If the result of the decision is used out of scope.
- If the decisions have to be executed every time, because the result of the previous decisions can not be used again.

In these cases the archiving Design Patterns give us recipes for the solutions.

There are two solutions: According to the first solution, the decision can be archived by a producer class hierarchy, and later the "product decision" can be realized based on the archived producer decision. According to the second solution, the decisions can be archived by a "decision stamp" by the help of which the decisions can be received from the decision container in the following decision cases.

As for the first case, the construction type of the product decisions determines the appropriate Design Pattern. So the functionality, the data structure and the decision predicate—which are defined by the product decisions—determine the method of decision archiving and the use of the appropriate Design Pattern.

According to this, the classification can be described as follows:

- If the products are not defined in the same decision's decision options as the same class hierarchy, the choice of the appropriate product can be realized by the archiving decision using the archiving class hierarchy.

  – Builder

- If the products can be defined in the same decision's decision options as the same class hierarchy, and the sorted decisions (defined in a product class

hierarchy) can be created by the archiving decision as an archiving class hierarchy, which is symmetric to the product class hierarchy.

  – Factory Method, Iterator

- If the products can be defined in the same decision as the sorted decisions in the class hierarchy (like the previously mentioned case), the sorted decisions can be created by the archiving decision. However, the sorted product decision class hierarchy gets the producer role, so there is not another producer class hierarchy.

  – Clone

- If the decision predicates of the different sorted decisions—which are defined by different class hierarchies—are equivalent, the same archiving decision as an archiving class hierarchy can be used in order to archive these decisions. In this case, the archiving class hierarchy can archive the equivalent decisions together, and can create them. (In this case Factory Method Design Patterns are contracted, accordingly producer decisions will be contracted.)

  – Abstract Factory

There is just one option in [1] for reusing the stored decisions in the decision container.

- If the result of the contracted decisions can be used more times out of scope, the results of the decisions can be stored in the decision container and the decisions can be received from the container using the decision stamps.

  – Flyweight

6. Example

The example program—which is described in this section—contains decision repetitions. In this case the decision repetition means that the decisions have equivalent decision predicates. As previously mentioned, the decisions which have equivalent decision predicates can be contracted. The decision repetition will be eliminated by contraction.

In the example, the functionality of the purchase is realized: Paying—By Cash/By Credit card; type of purchase—As a Private customer/As a Company customer. The decision predicate of the first decision of the program is evaluated by purchase.setIsCompany() invocation, which will determine the type of the

purchase accordingly the mode of the printing as a voucher or an account with different data.

The related decision's decision predicate is evaluated by purchase.setPayBy-Cash() invocation, based on it the type of the payment will be determined: paying by cash or by credit card. We can interpret this situation as decision options of the first decision case contain the cases of other decisions. So the first decision will determine the type of the purchase as a company or a private purchase, which determines the printing mode, and the second decision—as a contained decision—will determine the type of the payment, that is, by cash or by credit card. The other two decisions in the source have equivalent decision predicates to the first or the second decisions' predicates, so the other two decisions are decision repetitions.

Accordingly there are equivalent decisions in the code, which have to be eliminated by sorting the decisions into two class hierarchies. If we examine the decisions and their relations, we realize that, the Bridge Design Pattern can be detected based on the relations of the decisions. In this case the Bridge Design Pattern is "hidden" by defining the relating decisions in the methods, but sorting these decisions into the class hierarchies, the definitions of the relating decisions will be adopted to the known definition form of the Bridge Design Pattern. So the "hidden" Bridge Design Pattern will be introduced by sorting the decisions and defining them into more abstract forms, by which the decision redundancies will be eliminated and the Bridge Design Pattern will be realized in its better-known form.

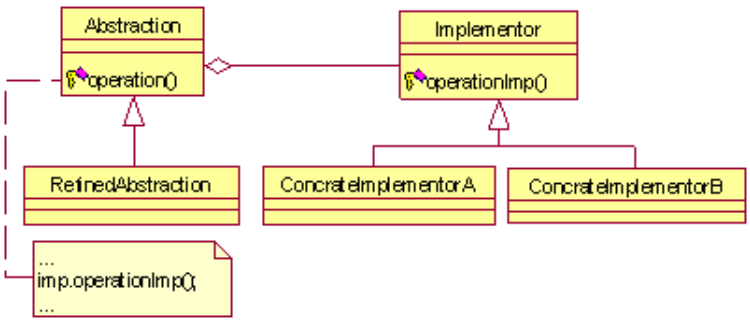

Diagram 6. Class diagram of Bridge Design Pattern [1]

In our interpretation the Bridge Design Pattern shows the method of decision elimination for the given decision relations.

As we can see, the use of a Design Pattern can be noticed based on the relations of the decisions when it is defined in "hidden" mode in a method.

The example is based on Java syntax [8].

Level 1

As it was mentioned, there are decision repetitions in the source, so there are decisions with equivalent decision predicates. These decisions can be sorted, and the data-structure and methodology—which they define—can be contracted. In the example, the first decision determines the type of printing. The decision options of the first decision contain the decisions about the type of payment such as paying by cash or by credit card.

Level 2 — Sorting the "Company or Private Customer?" decision

Two decisions were contracted. The first is defined in the setPurchaseInfo method, where, based on the type of the purchase (isCompany) the program determines the parameters, which can be asked from the customer. (nameOfCustomer, addressOfCustomer or firstName, lastName). The other decision, the decision predicate of which is equivalent to the previously mentioned decision's decision predicate, is defined in the printBill method, where according to the previously mentioned decision, the type of the purchase (isCompany) determines the printing data.

The container decisions are sorted into a class hierarchy eliminating the decision repetition.

The type of the customer is determined in the main method, which is archived by the purchase object.

```
/** Setting the customer's type. */
Scanner sc = new Scanner( System.in );
System.out.println( "Billing for Company? " );
Purchase purchase;
if( sc.nextBoolean() ) // Billing for Company?
    purchase = new CompanyPurchase( pay );
else
    purchase = new PrivatePurchase( pay );
```

The decision predicate is an input parameter, which is determined by sc.nextBoolean(). It is archived by the instantiation of one of the Purchase subclasses

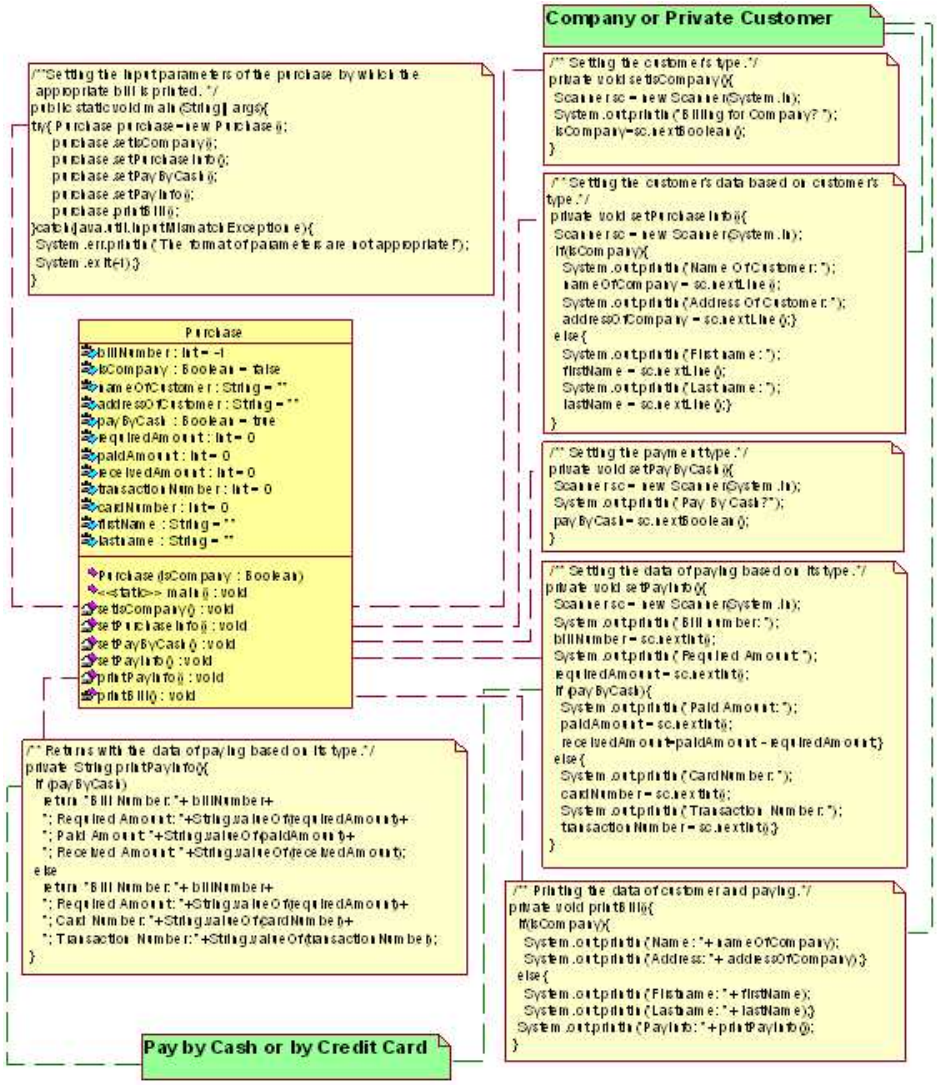

Diagram 7. Source with decision repetitions

using purchase object. The archived decision by purchase object is reused two times:

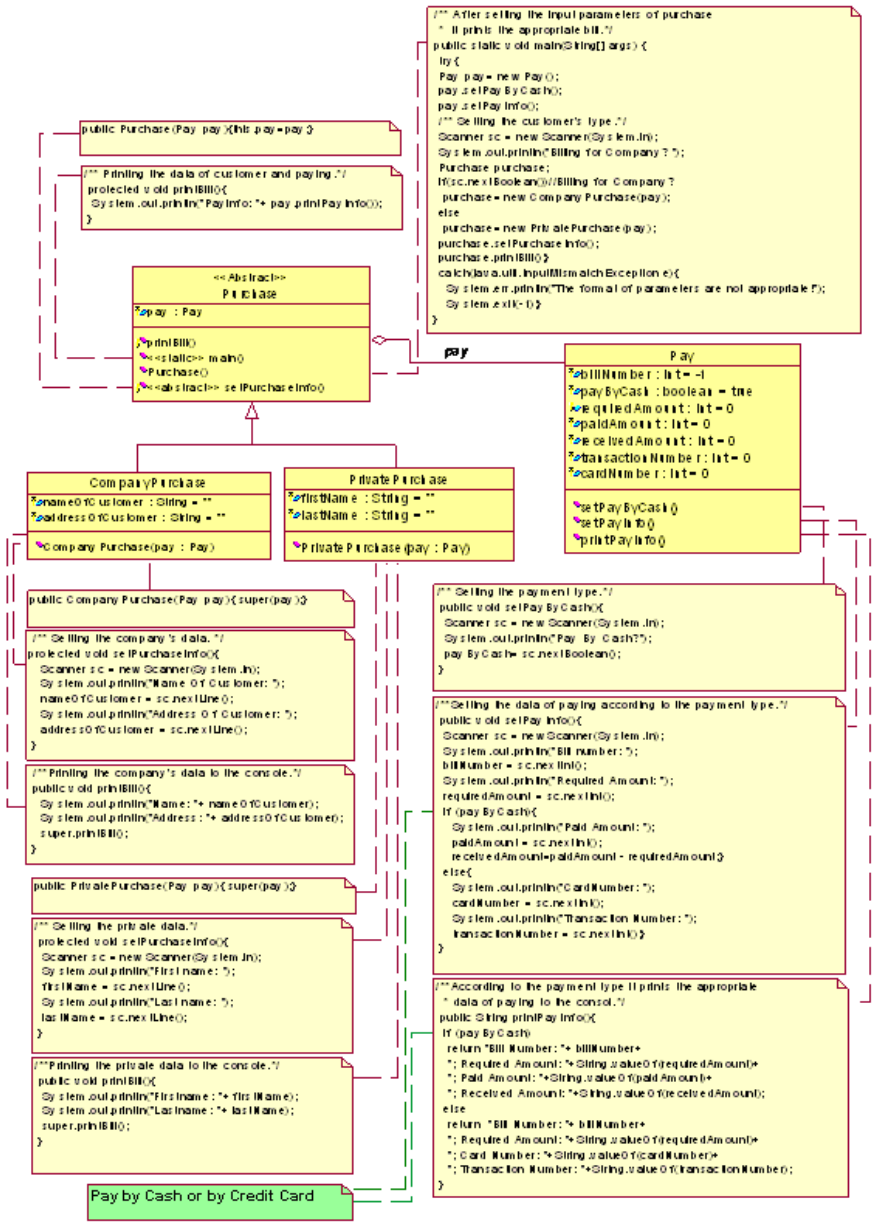

Diagram 8. Sorting the decision cases of container decision type

```
purchase.setPurchaseInfo();
purchase.printBill();}
```

In the setPurchaseInfo method, according to the type of the customer the private or company data are set. In the printBill method according to the type of the customer the private or company data are printed.

The CompanyPurchase class—as the subclass of the Purchase class—is available, if the type of the customer is a company as it is decided in the Main method. It is necessary to receive and print the name and address of the company The PrivatePurchase class—as the subclass of the Purchase class—is available, if the type of the customer is not a company as it is decided in the Main method. It is necessary to get and print the firstName and lastName of the customer.

Level 3 — Sorting the "Paying by cash or credit card" decision

At this Level the known form of the Bridge Design Pattern will be realized by sorting the decisions about the payment type. The decisions of the payment type are sorted into the class hierarchy, in which the different paying modes are defined in the subclasses as the decision options. If somebody pays by cash, the number of the credit card and the transaction number are not necessary, but the paid and received amounts are required. In the case of paying by credit card the received and paid amounts are not necessary, but the credit card number and the transaction number are required. After the contraction of the equivalent decisions of the paying mode was executed (which were in the setPayInfo and the printPayInfo methods as it was defined in Level 2), the decision about the paying mode will be executed just once, which will be enclosed and archived by the "Pay" class hierarchy and the enclosed decision will be reused on the next occasions.

The payment type is determined in the main method, which is archived by the pay object.

```
/** Setting the type of paying. */
System.out.println( "Pay By Cash?" );
Pay pay;
if( sc.nextBoolean() ) // Pay By Cash?
  pay = new PayByCash();
else
  pay = new PayByCreditcard();
```

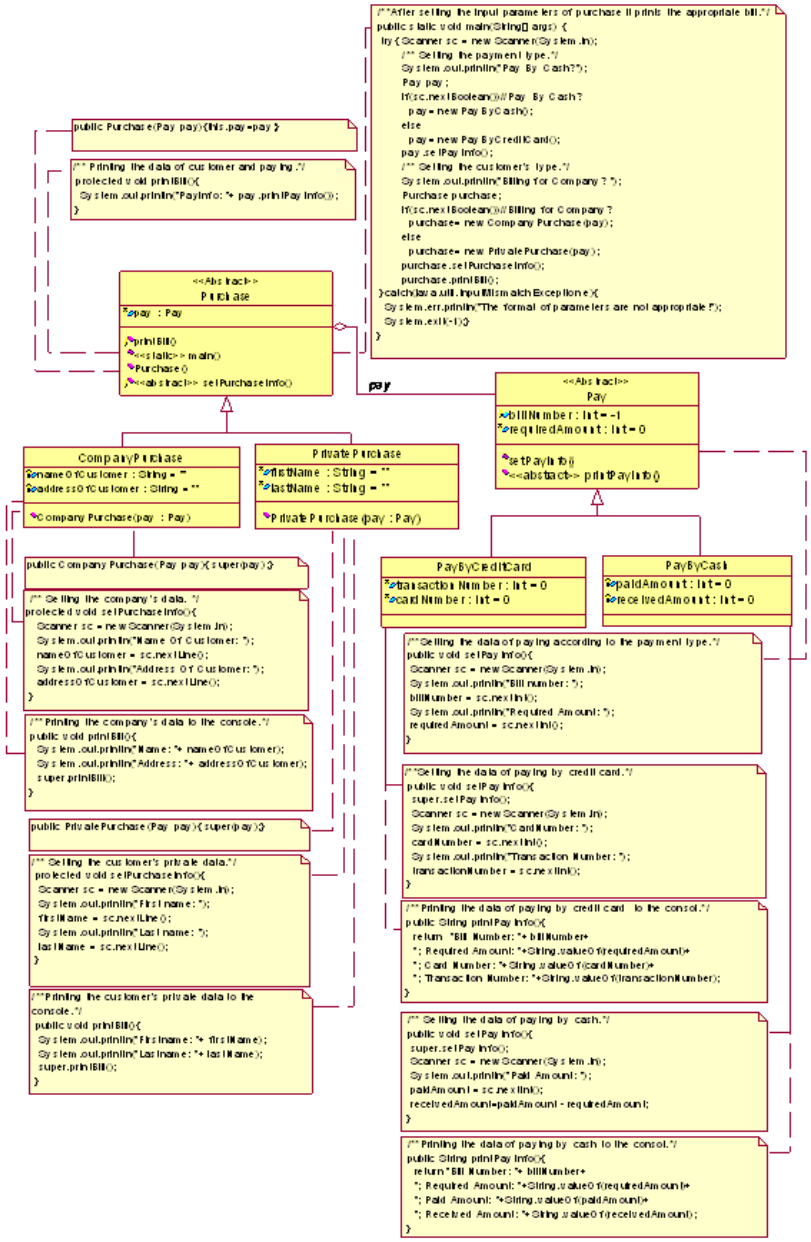

Diagram 9. Sorting the decision cases of second decision type

The decision predicate is an input parameter, which is determined by sc.nextBoolean(). It is archived by the instantiation of one of the Pay subclasses using pay object. The archived decision by pay object is reused two times:

```
pay.setPayInfo();
pay.printPayInfo();
```

In the setPayInfo method according to the payment type the data of paying by cash or by credit card are set. In the printPayInfo method (invocated in the printBill method) according to payment type the data of paying by cash or by credit card are printed.

The decisions of the payment mode with different methodologies will be defined in the Pay class hierarchy. The two decision options differ in receiving and printing data about paying.

The PayByCash class, the subclass of the Pay class is available, if the customer pays by cash as it is decided in the Main method.

The PayByCreditCard class, as the subclass of the Pay class is available, if the customer pays by credit card as it is decided in the Main method.

7. Conclusion

The new interpretation of the object-oriented paradigms was described, which makes the object-oriented designing and programming easier.

As it was described, there are connections between the decision based interpretation of the object-oriented paradigms and Design Patterns, so Design Patterns give us recipes to eliminate the decision redundancy and archive the decisions. The applicability of the conception was presented through an example, in which the decision redundancies were eliminated by the introduction of the Bridge Design Pattern.

In the next papers we are going to examine the relationship between the application of Design Patterns and the decline of the decision repetition and between the program quality and (the number of) decision repetition. Based on the described interpretation of Design Patterns, their aims are realized clearly, which will help their better understanding. We will also determine the formalization of Design Patterns according to the decision-based conception based on the JML [6]. Using the formalization method we will realize the description of Design Patterns more exactly and the influence of the decision-based conception on the quality of the programs can be analyzed.

According to our plan, we will examine whether the decision repetition in the design and in the source can be eliminated by sorting them automatically, by the help of which the upgrading of the quality of the design and the source can be realized automatically, too.

Using the new decision-based conception a new, more natural classification of Design Patterns was described, by which we would like to launch the discussion about establishing the existing classification [1]. In order to examine the correctness of the briefly described classification we are planning to publish further articles and compare it with the existing classification.

References

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

[2] J. Rambaugh, I. Jacobson, G. Booch, *The unified modeling language reference manual*, Addision-Wesley, 1998.

[3] T. Taibi, D. Chek Ling Ngo, Formal Specification of Design Patterns—A Balanced Approach, *Journal of Object Technology*, Multimedia University, Malaysia (2003).

[4] A. H. Eden, J. Gil, Y. Hirshfel, A. Yehudai, *Towards a Mathematical Foundation For Design Patterns*, Computer Science Department and Department of pure mathematics, Tel-Aviv Univerity, IBM Research and Technion.

[5] T. Mikkonen, *Formalizing Design Patterns*, ICSE'98—IEEE Computer Society Press, 1998.

[6] L. Lamport, *The temporal logic of Actions*, 1994.

[7] G. Kusper, Programtervezési minták értelmezése normálformaként, *Networkshop konferenciakiadvány*, Miskolc, Hungary (2006).

[8] JavaTM 2 Platform Standard Edition 5.0,
`http://java.sun.com/j2se/1.5.0/docs/api`,
`http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html`.

[9] K. Fisher, J. C. Mitchell, *Notes on typed object-oriented programming*, Computer Science Dept., Stanford University, Stanford, 1994.

[10] H. Albin-Amiot, Y. Guéhéneuc, Meta-Modeling Design Patterns: Application to Pattern Detection and Code Analysis, *Workshop on Adaptative Object-Models and Metamodeling Techniques, ECOOP (European Conference on Oriented Programming)* (2001).

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall International Editions, New Jersey, 1991.

[12] M. Piefel, *Object Oriented Software Development—Coursework 'Information Engineering'*, Department of Computing, University of Bradford, 1996/97.

[13] Software Quality Metrics for Object Oriented System Environments, Software Assurance Technology Center as SATC, 1995.

[14] O. Nierstrasz, *Survey of Object-Oriented Concepts*, University of Geneva.

[15] G. Booch, *Object Oriented Analysis and Design with Applications*, Adison-Wesly, 1994.

SZABOLCS MÁRIEN
H–3433, NYÉKLÁDHÁZA
JÓZSEF ATTILA ÚT 2
and
UNIVERSITY OF DEBRECEN
H–4032, DEBRECEN
HUNGARY

E-mail: mariensz@hotmail.com

(Received August, 2007)