Annales Mathematicae et Informaticae
***35** (2008) pp. 95–121*
http://www.ektf.hu/ami

Decision based examination of object-oritented methodology using JML

Szabolcs Márien

University of Debrecen, Debrecen, Hungary

Submitted 28 September 2007; Accepted 10 October 2008

Abstract

The aim of object-oriented conception is to make sure that the program is well-structured, so as to become perspicuous; it can be extended easily, so that it could be maintained more easily; and its reusability can be increased in order to be modularized. There are lots of measuring methods, by which the realization of the mentioned aims is measured. The measuring methods are the metrics that give us indicators showing the complexity of the program structure.

Can the existing object-oriented metrics really indicate the structural quality of the program? As we know, these metrics examine structural properties like the number of inheritance levels, the number of subclasses, or the number of methods, which can not be the basis of real quality examinations. The reason of this is that the aim of the object-oriented conception is not clarified. In order to realize the aims of object-oriented technology, object-oriented paradigms should be reinterpreted.

According to our new conception object-oriented methodology is based on the elimination of decision repetitions, that is, sorting the decisions to class hierarchy, so that the data structure and methodology of the decision options could be determined by the subclasses of the given class. When 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 the instantiation of one of the subclasses. For the following decision cases the archived decision result can be used without knowing which decision option was used, that is, which subclass was instantiated, as it is enclosed by using the type of the parent class, except the necessary data structure and/or methodology is decision option specific.

There are two states of decisions depending on the place of their defining: the decision options and their data structures and methodologies can be defined in the method, but the sorted decision can be defined by a class and its subclasses.

In order to support the practical benefit of our conception, we are going to show how decisions can be formalized (that is, whether the decision states are defined in a method or by a class hierarchy) based on JML. Using the JML formalization those cases should be identified where decisions can be sorted, thus the elimination of decision redundancy is suggested.

According to our new conception the aim of object-oriented technology is the elimination of decision repetitions, which can be realized by sorting decisions. Therefore inheritances are the abstract definition forms of decisions, so the inheritances can be interpreted as decision abstractions.

1. Motivation

Object-oriented programming is a programming methodology. The programs based on it organize the collaborations of objects that are instances of one of the classes. Classes are built in class-hierarchies, where the connections between the classes are realized by inheritance relationships. [1]

The base paradigms of object-oriented technology are encapsulation [3, 6], inheritance [3, 4, 6], polymorphism [3, 6] and message-passing [3, 4, 6]. Encapsulation means that the data structures and the methodology are defined together, enclosing them in units as objects. Encapsulated data structures and methodology can be defined in classes, the instances of which are called objects. Modularized construction can be realized with the help of encapsulation, and as a result – if the methodology of one of the objects is changed – there are no side-effects in other objects. [6] Inheritance means that the data structures and the methodology, defined in a class, can be inherited by its subclasses. Subclasses can define new data structures and methods as complements of the inherited properties [4, 5] and can override inherited data structures and methodologies. Polymorphism means that the classes' methods can be overridden by their subclasses, so the method, which gets the control, is selected just in runtime (Late Binding). [3] Late Binding – according to another terminology – means that an object sends similar messages to different objects (classes and their subclasses), which results in the execution of a different code. [6]

There are lots of metrics in order to control the programs' quality. The quality of the design, the program and the efficiency of the testing can be checked by using these metrics. [4] The metrics defined in [2] (MOOD) and [4] are based on object-oriented paradigms. Accordingly, these metrics are used as a base concept of encapsulation (MHF[2], AHF[2]), inheritance (MIF[2], AIF[2], DIT[4], NOC[4]), polymorphism (POF[2]), message-passing (COF[2], LCOM[4], CBO[4]) in order to check the software quality.

But the metrics based on the base paradigms can be used for checking the quality subsequently. If the result of the checking shows that structure of the program is bad, it can be repaired by reconstruction. Using the solution in [7] reconstruction can be solved automatically based on inheritance checking.

By supporting the work of the program designer in the designing phase lots of designing failures could be avoided, and designing experience as designing 'recipes' could be reused. According to this concept, the creation of quality programs is guaranteed by rules based on designing experience. These generic recipes are Design Patterns [8], which is used to create a quality design and program. Are there any appropriate answers for the aims of Design Patterns beyond the general descriptions? We do not think there are, as the profession tried to define the rules of the programs' quality by collecting Design Patterns, but there are no clear answers for what the main concepts of Design Patterns are.

What is the reason for the deficiency of object-oriented metrics and why are not there any clear answers for the aims of Design Patterns? The answers can be found in [11]. According to that conception, the grounds for the answers can be found in the existing interpretation of object-oriented paradigms that block the extended examination of object-oriented methodology. Between the metrics – which are based on the interpretations of these paradigms – and the program quality there is no obvious connection, because these metrics depict the complexity of the program. Nevertheless, there are no clear answers for the aims of Design Patterns based on the existing interpretations of paradigms.

In order to resolve the problems described in [11], a new interpretation of the basic object-oriented paradigms is described, by which the basic concepts of object-oriented methodology can get another approach. According to this, we give new options for controlling the program quality and for repairing the programs as new guidelines are realized (Introduction, [11]) that improve the structures of programs and make their maintenance.

The new conception gives Design Patterns a clear interpretation. Accordingly, Design Patterns give us recipes for accomplishing the requirements of well-structured programs by reducing the number of decision repetitions. So Design Patterns give us recipes how decision repetitions can be eliminated in different decision construction cases [11].

In order to examine program structures and the performance of the guidelines of well-structured programs, we need a formalization tool that examines the definitions of decisions. Formal examinations are based on the Java programs' behaviour interface specification language – JML [12, 13, 14, 15]. JML specifies the data structures and the methodology of decision options based on logical expressions. JML formalized decisions have already been examined according to the decision-based conception and the guidelines of well-structured programs.

2. Introduction

Based on the decision-based interpretation of object-oriented concept [11], in this paper a new formalization method of decisions is realized using JML.

In this section, according to [11], the new interpretation of object-oriented concept is shown.

The decisions of the program code decide about the data structure and functionality are specified in the decisions. The main concept of object-oriented methodology is the elimination of decisions' repetition by sorting them to a “common place". This "common place" is a class with subclasses, so decision repetition can be eliminated by class hierarchy. After sorting the decisions, the decision about the necessary functionality and data structure is executed only once. Decision archiving is realized by the instantiation of the subclass with the appropriate functionality and/or data structure. The result of the decision (the archived decision) - as an instance of the appropriate subclasses - can be used at other decision places without having any specific information about it. Accordingly, the decisions can be enclosed in class hierarchy.

Decision cases are important parts of programs, where the appropriate decision option can be decided by using the actual values.

In order to ensure that a program is well-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, unless sorting the decisions is impossible. It is important to consider manageability, because the introduction of a new decision option can be solved easily if it can only be built in one place of the program.

- Decisions having equivalent decision predicates but differing in their decision option definitions should not reoccur. This case is different from the previous one, as though 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.

- Decisions should not reoccur, so a decision should be executed just once during the same running, if the predicates of decisions are equivalent and the decision options define the same data structure and functionality. The elimination of decision repetitions has two aspects:

  - The result of the decision - as the data structures or/and the methodology of the decision options - can be used several times.
  - The result of the decision can be used later more times, but a new instance of the structures or/and methodology of the decision options is created each time. The archived decision can be used later for creating an instance of the decision options.

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

2.1. Inheritance as decision abstraction

Inheritance means that the data structure and the methodology 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 [4, 5] 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 could be archived, it has to be sorted, which means that the data structure and methodology of the decision options have to be defined in a class hierarchy, as a parent class and its subclasses. Derivation/inheritance ensures the enclosing and archiving of the decision to the class hierarchy, therefore the definitions of the decisions can be contracted and decision repetitions can be eliminated.

According to this interpretation class hierarchy – the class with its subclasses – based on inheritance 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 one time.
- Archiving the decision, so that the result of the decision could be used for the next occasions, unless the required data structure or methodology is specified by one of the decision options only.
- Enclosing the decision. The result of the decision is not known in the next decision cases, unless the required data structure or methodology is specified by only one of the decision options.
- By introducing a new subclass, decision options can be extended easily. When creating a new subclass, only 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 only one of the decision options.

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

2.2. Polymorphism as decision enclosing

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 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.

2.3. Encapsulation

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 defined in an abstract form sorted in class hierarchy, the decision options are realized in the subclasses. If there is a change in the data structure and the methodology of the decision option, no side-effects occur in other decision cases and other decision options, accordingly the decision option can define the data structure and the methodology by a subclass enclosing them (the data structure and the methodology).

2.4. Aggregation as dynamic decision embedding

By aggregation the sorted decision can be referred to. If there is a decision case, in which 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 as the result of the decision, the sorted decision can be used in the decision cases. The result of the decision will be referred to by aggregation.

When we talk about aggregation, we have to know that it is the tool of relating decisions.

In the following sections of the paper we will show how the described decision based conception can be supported by JML. In Section 3 the JML specification language is described, and on the basis of this, Section 4 introduces the formalization method of the two states of the decisions (defined by method or by class hierarchy). In the final part of the paper, in Section 6, an example shows how the decisions can be formalized before and after decision sorting.

3. JML

JML – Java Modelling Language is a behaviour interface specification language [12, 13], by which the syntactical interface and the behaviour of Java programs is specified. [12]

The syntactical interfaces are the Java interfaces and the programmer interfaces of Java programs, that is, the signatures of the methods, the names and types of the variables. The behaviour of the interfaces can be specified by JML annotations, which define how classes and methods can be used. [12]

The JML specification language combines the Eiffel-style syntax with the model-based semantics as in VDM and Larch. Eiffel-style assertions are extended to use Java expressions. JML combines this with the model-based approach of VDM and Larch. [13, 15]

Accordingly, JML contains many state-based specification languages' core specification constructions, for example, pre- and post-conditions, assertions, invariants. These constructions are not able to realize the formal modular verification of object-oriented programs. Therefore, JML uses extra constructions such as frame-properties, data groups, ghost and model variables. [14]

JML specifications can either be written in separate – specification – files or as annotations in Java program files (the Java compiler interprets these annotations as comments, which are ignored by the compiler). Specification files and their specifications can be organized into inheritance-hierarchies, which make the creation of the well-structured specification easier.

There are two kinds of specification cases in JML: Lightweight and Heavyweight specifications. Lightweight specification cases are useful when giving partial specifications, but if the complete specification is necessary, we should use the heavyweight specification option.

In the following part the main concepts of JML specification constructions are described, in order that the Reader could interpret the examination of the decision-based extension of object-oriented concepts by JML more easily.

There are two kinds of specification constructions of JML:

- Behaviour specification constructions, such as 'assert', 'assume', 'require', ...
- Specification constructions of classes and interfaces, such as invariants, models, ...

3.1. Behaviour specification constructions

The basic constructions of JML are the pre- and post-conditions of the commands and methods, which determine the program states before and after the executions of the commands or methods.

The pre- and post-conditions can be described as a contraction between a method (its implementer) and its caller (user) as follows:

- Pre-condition:
  - The method or the command assumes that the pre-condition has been realized.
  - The caller of the method or the command ensures the realization of the pre-condition.
- Post-condition:
  - The method or the command ensures the realization of the post-condition.
  - The caller of the method or the command assumes the realization of the post-condition.

The appearances of the pre- and post-conditions in JML specification are:

- Conditions in the methods:
  - ‘Assume’: Assertion that the program requires.
  - ‘Assert’: Assertion that the program ensures.
- Conditions between methods:
  - ‘Requires’: It specifies the pre-condition of the method.
  - ‘Ensures’: It specifies the post-condition of the method.

The pre- and post-state of the variables can be distinguished as follows:

- Pre-state: The starting state of the variables is signed by enclosing the variable name with the ‘\old()’ expression.
- Post-state: The ending state of the variables is signed by the variable name.

The variables with modified values in the methods are specified by assignable annotations such as frame conditions, which can define the “frame” of the possible state-transitions. JML behaviour specification constructions are based on the requirements of Hoore calculus.

3.2. Specification of interfaces and classes

JML can specify invariants, such as general conditions, that help to narrow the state-space of classes and interfaces.

The initial conditions of the variables can be specified.

History constraint specifies the relations between pre- and post-states, which are realized by every state-transition.

The data group is a set of fields (locations). The data-groups, such as the grouped fields, are the basic-units of the states and the state-transitions.

JML has an abstract construction. It is the model variables that can be used in the model specification. The ‘represents’ clause can join the model variable with the implementation variable as its implementation representation.

4. Decision formalization

In order to formalize the decisions of object-oriented programs, the formalization of the data structure and the behaviour of the programs can be solved, because it is necessary to compare the data structure and the behaviour of decision options. The analysis of decision predicates [11] is necessary for the examination of redundant decisions.

JML has constructions to realize behaviour specification and the specification of data structures. Because the behaviours are specified by logical formulae as post-conditions, the equivalence of the decisions’ decision options can be examined based on the data structures and the behaviours. Based on the formalized behaviours as post-conditions, the examination of the decision-predicated are realized, too.

In this section the formalization of the decisions by JML is described. We describe the JML formalization of one-level decisions and the formalization of two- or more-level decisions.

In order to show the connections between the two states of the decisions (the decision is defined by a method or by class hierarchy), we describe the JML formalization of non-sorted and sorted decisions. Based on the formalized sorted decisions, we can see how the result of the first decision case is in-closed, archived, which can be reused further on, in other decision cases of that decision.

4.1. Formalization of non-sorted decisions

If the decision is not sorted, the methodology and the data structure of the decision options are defined in the method, not in the class hierarchy.

The pre-conditions and the post-conditions of the behaviours and the consequences of the decision options are defined in the specification of the method, where the definition of the decision options is separated by the keyword ‘also’.

```
/*@ public normal_behavior //D
  @ requires $p_1$; //$D_{L_1}$
  @ assignable $v_1,\ldots,v_a,v_b,\ldots,v_c$;
  @ ensures $\text{condition}_1$ &&...&& $\text{condition}_i$ && $\text{condition}_j$ &&...&& $\text{condition}_k$;
  @ also
  @ requires !($p_1$); //$D_{L_2}$
  @ assignable $v_1,\ldots,v_a,v_e,\ldots,v_f$;
  @ ensures $\text{condition}_1$ &&...&& $\text{condition}_i$ && $\text{condition}_l$ &&...&& $\text{condition}_m$;
@*/
```

The two decision options of the decisions are separated by the keyword ‘also’.

The pre-conditions of the decision options are $p_1, !p_1$. The pre-condition determines the appropriate decision option, by which the appropriate data structure and behaviour is realized. The ‘assignable’ assertion defines the data structure, which is modified by the decision option. The behaviour of the decision option is defined by ‘ensures’ assertions as post-conditions.

The data structures and the behaviour of the decision options have common and decision option-specific parts. It is important, because if we sort the decisions, the common parts of the decision options are specified by the parent class in the class hierarchy, and the decision option specific parts are defined by the subclasses. The data structures of decision options are:

$v_1, \ldots, v_a$ – Variables, which are modified by all decision options.

$v_b, \ldots, v_c$ – Variables, which are modified in D_{L_1} decision option.

$v_e, \ldots, v_f$ – Variables, which are modified in D_{L_2} decision option.

The behaviours of the decision options are:

condition_1 && ... && condition_i – Common behaviours of the decision options.

condition$_j$ &&…&& condition$_k$ – Behaviour, which is specified by D_{L_1} decision option.

condition$_1$ &&…&& condition$_m$ – Behaviour, which is specified by D_{L_2} decision option.

Formalization of the decision which contains other decisions (Complex decision).

```
/*@ public normal_behavior //D_2 in D_1
  @ requires p_1; //D_1L_1
  @ {|
  @    requires p_2; //D_2L_1
  @    assignable v_1,...,v_a,v_b,...,v_c, //D_1L_1
  @                v_g,...,v_h,v_i,...,v_j; //D_2L_1
  @    ensures condition_1 &&...&& condition_o && condition_p &&...&& condition_q && //D_1L_1
  @              condition_t &&...&& condition_u && condition_v &&...&& condition_w; //D_2L_1
  @  also
  @    requires !(p_2); //D_2L_2
  @    assignable v_1,...,v_a,v_b,...,v_c, //D_1L_1
  @                v_g,...,v_h,v_k,...,v_l; //D_2L_2
  @    ensures condition_1 &&...&& condition_o && condition_p &&...&& condition_q && //D_1L_1
  @              condition_t &&...&& condition_u && condition_x &&...&& condition_y; //D_2L_2
  @  |}
  @ also
  @ requires !(p_1); //D_1L_2
  @ {|
  @    requires p_2; //D_2L_1
  @    assignable v_1,...,v_a,v_e,...,v_f; //D_1L_2
  @                v_g,...,v_h,v_i,...,v_j; //D_2L_1
  @    ensures condition_1 &&...&& condition_o && condition_r &&...&& condition_s && //D_1L_2
  @              condition_t &&...&& condition_u && condition_v &&...&& condition_w; //D_2L_1
  @  also
  @    requires !(p_2); //D_2L_2
  @    assignable v_1,...,v_a,v_e,...,v_f; //D_1L_2
  @                v_g,...,v_h,v_k,...,v_l; //D_2L_2
  @    ensures condition_1 &&...&& condition_o && condition_r &&...&& condition_s && //D_1L_2
  @              condition_t &&...&& condition_u && condition_x &&...&& condition_y; //D_2L_2
  @  |}
@*/
```

The decision options ($D1_{L_1}, D1_{L_2}$) of D_1 decision contain the decision options ($D2_{L_1}, D2_{L_2}$) of D_2 decision. Complex decisions can be specified just like simple decisions. There are common parts and decision option specific parts of the decision options' behaviours and data structures. The common and the decision option specific variables and conditions of behaviours – as it can be seen in the specification of simple decisions – are signed by indexes.

4.2. Formalization of sorted decisions

If the decision is sorted, the decision is specified by the parent class and its subclasses. The parent class defines the common parts of the decision options, and the decision option specific parts are defined by the subclasses. The parent class as a type can archive the decision result of the decision case, accordingly, the variable that encloses the decision gets the parent class type (in this case its type is 'o'). The further decision cases can use the o-variable – which encloses the decision – in order to achieve the functions of the decisions and decision options.

4.2.1. One-level sorted decisions

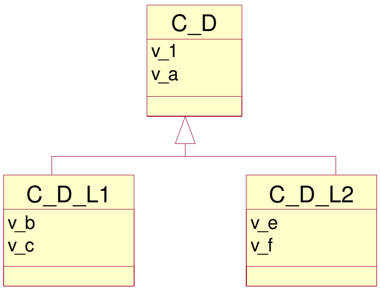

Diagram 1. C_D, $C_{D_{L_1}}$, $C_{D_{L_2}}$ class-hierarchy by UML diagram [9].

The JML formalization of sorted decision on the place of decision-sorting:

```
/*@ public normal_behavior //D
  @ requires o instanceof $C_{D_{L_1}}$; // $\Leftrightarrow$ $p_1$ //$D_{L_1}$
  @ assignable o;
  @ ensures $condition_1$ && ... && $condition_i$ && $condition_j$ && ... && $condition_k$;
  @ also
  @ requires o instanceof $C_{D_{L_2}}$; // $\Leftrightarrow$!$p_1$ //$D_{L_2}$
  @ assignable o;
  @ ensures $condition_1$ && ... && $condition_i$ && $condition_l$ && ... && $condition_m$;
 @*/
```

There are not many differences between the formalizations of the decisions' two states (the sorted and the non-sorted states), because the formalization of the sorted decision shows the specification of decision options, too. So the decision options of the "enclosed" decision are specified in the JML formulae of the decision. It is useful, because it shows the decision options of enclosed decisions.

In the first decision case, the decision predicate stays in its original form, but in the other decision cases – according to the decision based conception – the type of the object determines the behaviour of the decision (such as the behaviour of the appropriate decision option).

By sorting the decision, the behaviours of the decision options are separated in two subclasses, and the common parts are sorted into the parent class. The common and the decision option specific parts are united providing the appropriate behaviour in the decision cases.

The JML formalization of the parent class of the sorted decision is as follows:

```
/*@ public normal_behavior //D
  @  assignable v_1, v_a;
  @  ensures condition_1 && ... && condition_i;
@*/
```

The parent class provides only the common behaviour of the sorted decision.

The subclasses specify the decision option specific parts of the sorted decision completing the common behaviour:

```
/*@ also //D_L1
  @ public normal_behavior
  @  assignable v_b, v_c;
  @  ensures condition_j && ... && condition_k;
@*/

/*@ also //D_L2
  @ public normal_behavior
  @  assignable v_e, v_f;
  @  ensures condition_l && ... && condition_m;
@*/
```

As it can be seen, if the decision is sorted and defined by class-hierarchy, the decision formalization is transformed. The following differences can be found between the formalization of the sorted and non-sorted decisions:

- Predicates of the decision options: The first decision case keeps the original $p_1, !p_1$ decision predicates. The result of the first decision case is archived by an instance of one of the subclasses, and it is enclosed by the type of the parent class in the class hierarchy. The archived decision will be reused by the $\mathrm{o\ instanceof\ C_{D_{L_1}}}, \mathrm{o\ instanceof\ C_{D_{L_2}}}$ predicates on the next decision cases.
- The data-structure which is modified by the decision option will be specified by the keyword 'assignable':
  - The data-structure is the content of the "o" object, which contains the common data structure of the parent class and if the "o" object is the

instance of one of the subclasses, the content is completed with the data structure of the subclass.

- The variables will be referred to in the post-conditions of the decision options (in the subclasses) as it can be seen in the following list:

$$D_{L_1} \Rightarrow ((C_{D_{L_1}})o).v_b, \dots, ((C_{D_{L_1}})o).v_c$$
$$D_{L_2} \Rightarrow ((C_{D_{L_2}})o).v_e, \dots, ((C_{D_{L_2}})o).v_f$$

  The decision option specific variables of the "o" object – the type of which is the parent class – are achieved by type-forcing.

  Accordingly, the object – standing for the parent class in the class hierarchy – encloses the result of the decision. Its decision option specific options can be achieved as already shown.

It is not clear why the usage of type-forcing in the 'assignable' assertions is faulty, but in the post-conditions the usage of type-forcing is required if the data structure of one of the subclasses is required.

4.2.2. More-levels, complex sorted decisions

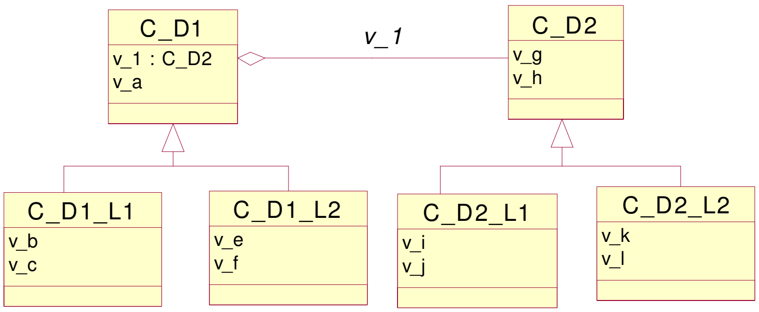

Diagram 2. C_{D_1}, C_{D_2} class-hierarchies by UML diagram [9].

The JML formalization of complex sorted decision on the place of decision-sorting:

```
/*@ public normal_behavior //$D_2$ *in* $D_1$
  @ requires o isntanceof $C_{D_{1_{L_1}}}$; //$D_{1_{L_1}}$ $\Leftrightarrow p_1$
  @ {|
  @   requires o.$v_1$ instanceof $C_{D_{2_{L_1}}}$; //$D_{2_{L_1}}$
  @   assignable o; //$D_{1_{L_1}}$
  @   ensures condition$_1$ && ... && condition$_o$ && condition$_p$ && ... && condition$_q$ && //$D_{1_{L_1}}$
  @           condition$_t$ && ... && condition$_u$ && condition$_v$ && ... && condition$_w$; //$D_{2_{L_1}}$
  @  also
  @   requires o.$v_1$ instanceof $C_{D_{2_{L_2}}}$; //$D_{2_{L_2}}$
```

```
 @    assignable o
 @    ensures condition$_1$ &&...&&condition$_o$ &&condition$_p$ &&...&&condition$_q$ &&//D$_{1L_1}$
 @              condition$_t$ &&...&&conditionl$_u$ &&condition$_x$ &&...&&condition$_y$;//D$_{2L_2}$
 @   |}
 @ also
 @  requires o isntanceof C$_{D_{1L_2}}$;//D$_{1L_2}$ $\Leftrightarrow$!p$_1$
 @  {|
 @    requires o.v$_1$ instanceof C$_{D_{2L_1}}$;//D$_{2L_1}$
 @    assignable o;
 @    ensures condition$_1$ &&...&&condition$_o$ &&condition$_r$ &&...&&condition$_s$ &&//D$_{1L_2}$
 @              conditionl$_t$ &&...&&condition$_u$ &&condition$_v$ &&...&&condition$_w$;//D$_{2L_1}$
 @   also
 @    requires o.v$_1$ instanceof C$_{D_{2L_2}}$;//D$_{2L_2}$
 @    assignable o;
 @    ensures condition$_1$ &&...&&condition$_o$ &&condition$_r$ &&...&&condition$_s$ &&//D$_{1L_2}$
 @              condition$_t$ &&...&&condition$_u$ &&condition$_x$ &&...&&condition$_y$;//D$_{2L_1}$
 @   |}
@*/
```

4.3. The conditions of well-structured programs based on JML specification

In the following part we describe the facilities of the JML specification of decisions, by which the decision repetitions and the redundant decision definitions can be detected. The full description of these facilities is out of scope of this paper, in the following we just describe the basis of this methodology:

In the Introduction part the following guidelines of a well-structured program were described:

- 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 it is impossible to sort decisions.

  If the data structures and methodologies of decisions are equivalent, these decisions have to be sorted in the same class-hierarchy. By using JML specification, decisions are equivalent when the data structures of the decisions – which are specified by “assignable” – are equal, and the methodologies of the decisions as the post-conditions of the decision options (specified by “ensures”) are equivalent. The decision can be the extension of another one. In this case, one of the data structures is a subset of the other one and there is an implication relation between the postconditions. In this case, the examination of decision predicates is not important.

- Decisions with equivalent decision predicates and different data structures and/or methodologies should not be repeated. In this case, the data structures and the methodologies of the JML specifications of decisions are not equal, but the decision predicates – specified by "requires" – are equivalent. The decision options have to be contracted by sorting them into the same class hierarchy, which will be the common decision abstraction of the contracted decisions. (This case is shown in the Example Code.)

- Decision cases should not be repeated. One decision should be executed just once. (It is the union of the previously mentioned two cases, because the definitions of the decision options are equal, and the decision predicates are equivalent, too.) In this case, the JML formulae of the decisions' data structures and the methodologies are equal and the decision predicates of the decisions (specified by "requires") are equivalent, too.

5. Example

The example shown in this section contains decision-repetition. These decisions have equivalent decision predicates and different data structures, methodologies. According to the previously mentioned conditions of well-structured programs these decisions can be contracted and sorted into class hierarchy, by which the decision-repetition is eliminated.

In the example, the functionality of the purchase is realized: Paying – By Cash/ By Bankcard

The decision about paying mode will be reused later more times. The paying mode determines the parameters, which get as program arguments and it determines the printing data.

The example is based on Java syntax [10].

The two levels of the example code – before and after decision sorting – are also specified by JML, therefore the JML formalization of the two states can be examined and compared.

Accordingly, the Pay class and the Pay class-hierarchy are formalized by JML, by which the differences of the formalizations between the not-sorted and sorted decisions can be described.

5.1. Before decision sorting

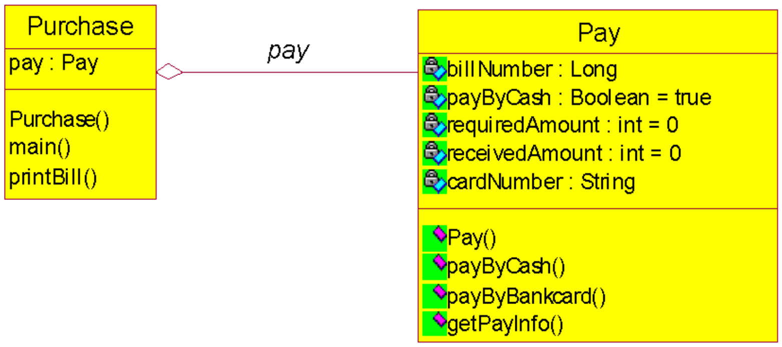

Diagram 3. The two classes of the example before decision sorting by UML diagram [9].

```
package hu.decision.example;
//@ model import org.jmlspecs.models.*;

/** Printing the payment data.
 */
public class Purchase {
  /*@ public static pure model boolean parseable( String s ) {
  @ try { int d = Integer.parseInt(s); return true; }
  @ catch (Exception e) { return false; }}
  @*/

  /*@ public static pure model Pay desidePayingType(String[] args) {
  @   return new Pay(args);
  @}
  @*/

  /**  Payment data - according to the payment type - is got
   *  using the instance of Pay class
   */
  public Pay pay;
  //@ instance invariant pay != null;

  public static void main(String[] args) {
    Purchase purchase=new Purchase();
    purchase.init(args);
    //Printing bill.
    purchase.printBill();
  }

  /** Checking the number of arguments and creating the Pay instance,
   *by which the payment data is printed.
   */
  /*@ private normal_behavior
  @   requires args==null||args.length<4;
  @   assignable \nothing;
  @   ensures false;
  @ also
  @ private normal_behavior
  @   requires args.length>=4&&pay==desidePayingType(args)&&
  @     pay instanceof Pay ;
  @ {|
  @   {|
  @     requires args[0].equals("true")&&parseable(pay.args[1])&&
  @       parseable(pay.args[2])&&parseable(pay.args[3]);
  @     assignable pay, pay.payByCash, pay.billNumber,
  @       pay.requiredAmount, pay.receivedAmount, System.out;
  @     ensures pay.payByCash==true &&
  @       pay.billNumber==Integer.parseInt(args[1])&&
  @       pay.requiredAmount==Integer.parseInt(args[2]);
  @     ensures pay.receivedAmount==Integer.parseInt(args[3]);
  @   also
  @     requires args[0].equals("false")&&parseable(pay.args[1])&&
  @       parseable(pay.args[2]) && parseable(pay.args[3]);
  @     assignable pay, pay.payByCash, pay.billNumber,
  @        pay.requiredAmount, System.out;
  @     ensures pay.payByCash==false &&
  @       pay.billNumber==Integer.parseInt(args[1])&&
  @       pay.requiredAmount == Integer.parseInt(args[2]);
  @     ensures pay.cardNumber == args[3];
  @   |}
  @ also
  @   requires !parseable(pay.args[1])||!parseable(pay.args[2])||
  @     !parseable(pay.args[3]);
  @   assignable \nothing;
  @   ensures false;
  @ |}
  @*/
  private void init(String[] args)
  {
    //If there are not enough arguments.
    if(args == null || args.length < 4 ){
      System.err.println("There are not enough arguments!");
      System.exit(-1);
    }
    try{
      //Creating the Pay object by which the payment behaviours are realized.
      pay = new Pay(args);
    }
    catch (java.lang.NumberFormatException nfe)
    {
      System.err.println("The format of Arguments is not appropriate!");
      System.exit(-1);
    }

  }

  /** Based on the pay instance payment data is printed.
   */
  /*@ private normal_behavior
  @   requires pay.payByCash==true;
  @   assignable System.out;
  @   ensures (* Prints the Bill Number, Required Amount,
  @       Received Amount*);
  @ also
  @ private normal_behavior
  @   requires pay.payByCash==false;
  @   assignable System.out;
  @   ensures (* Prints the Bill Number, Required Amount,
  @       Card Number*);
  @*/
  private void printBill(){
    String payInfo =pay.getPayInfo();
    System.out.println("Payinfo: "+ payInfo);
  }
}

/*-----------------------------------------------------------------*/
package hu.decision.example;
//@ model import org.jmlspecs.models.*;

/** Determining payment type (as by cash or by bankcard).
```

```
*/
public  class Pay {

  /*@ public static pure model boolean parseable( String s ) {
  @ try { int d = Integer.parseInt(s); return true; }
  @ catch (Exception e) { return false; }
  @ }
  @*/

  public String[] args;
  //@ invariant args!=null && args.length==4;
  public long billNumber = 0;
  //@ private instance initially billNumber == 0;
  public boolean payByCash=true;
  //@ private instance initially payByCash == true;
  public int requiredAmount=0;
  //@ private instance initially requiredAmount == 0;
  public int receivedAmount=0;
  //@ private instance initially receivedAmount == 0;
  public String cardNumber="";
  //@ private instance initially cardNumber == "";

  /** Determining payment type as by cash or by bankcard.
   *  Getting the bill number and the required amount is
   *  necessary in every case.
   */
  /*@ public behavior
  @ {|
  @   requires args[0].equals("true");
  @   assignable args, payByCash, billNumber, requiredAmount,
  @     receivedAmount;
  @   ensures payByCash==true &&
  @     billNumber==Integer.parseInt(args[1])&&
  @     requiredAmount == Integer.parseInt(args[2]);
  @   ensures receivedAmount == Integer.parseInt(args[3]);
  @ also
  @   requires !parseable(args[1])||!parseable(args[2])||
  @     !parseable(args[3]);
  @   assignable args, payByCash, billNumber, requiredAmount;
  @   ensures false;
  @   signals_only java.lang.NumberFormatException;
  @ |}
  @ also
  @ public behavior
  @ {|
  @   requires args[0].equals("false");
  @   assignable args, payByCash, billNumber, requiredAmount,
  @     receivedAmount, cardNumber;
  @   ensures args==in_args && payByCash==false &&
  @     billNumber==Integer.parseInt(args[1])&&
  @     requiredAmount == Integer.parseInt(args[2]);
  @   ensures cardNumber == args[3];
  @ also
  @   requires !parseable(args[1])||!parseable(args[2]);
  @   assignable args, payByCash, billNumber, requiredAmount;
  @   ensures false;
  @   signals_only java.lang.NumberFormatException;
  @ |}
    @*/
  public Pay(String[] in_args) throws NumberFormatException
  {
    this.args=in_args;
    if(args[0].equals("true"))
      payByCash =true;
    else if(args[0].equals("false"))
      payByCash =false;
    System.out.println("Pay By Cash?:(true/false) "+payByCash);

    billNumber = Integer.parseInt(args[1]);
    System.out.println("Bill Number:(Number) "+billNumber);

    requiredAmount = Integer.parseInt(args[2]);
    System.out.println("Required Amount:(Number) "+requiredAmount);

    if (payByCash)
      payByCash();
    else
      payByBankcard();
  }

  /** If the customer pays in cash, then getting the
   *  received amount is necessary.
   */
  /*@ private normal_behavior
  @   requires parseable(args[3]);
  @   assignable receivedAmount;
  @   ensures receivedAmount == Integer.parseInt(args[3]);
  @ also
  @ private exceptional_behavior
  @   requires !parseable(args[3]);
  @   assignable receivedAmount;
  @   signals_only java.lang.NumberFormatException;
  @*/
  private void payByCash() throws NumberFormatException
  {
    receivedAmount = Integer.parseInt(args[3]);
    System.out.println("Received Amount:(Number) "+receivedAmount);
  }

  /** If the customer pays by bankcard, then getting
   *  the card-number is necessary.
   */
  /*@ private normal_behavior
  @ assignable  cardNumber;
  @ ensures  cardNumber == args[3];
  @*/
  private void payByBankcard()
  {
    cardNumber = args[3];
    System.out.println("cardNumber:(String)"+cardNumber);
  }

  /** Printing payment data according to payment type.
   */
  /*@ public normal_behavior
  @   requires payByCash == true;
  @   assignable \nothing;
  @   ensures \result == "Bill Number: "+String.valueOf(billNumber)+
  @     "; Required Amount: "+String.valueOf(requiredAmount)+
  @     "; Received Amount: "+String.valueOf(receivedAmount);
  @ also
  @ public normal_behavior
  @   requires payByCash == false;
  @   assignable \nothing;
  @   ensures \result == "Bill Number: "+String.valueOf(billNumber)+
  @     "; Required Amount: "+String.valueOf(requiredAmount)+
  @     "; Card Amount: "+String.valueOf(cardNumber);
  @*/
  public String getPayInfo()
  {
    if (payByCash)
      return "Bill Number: "+String.valueOf(billNumber)+
      "; Required Amount: "+String.valueOf(requiredAmount)+
      "; Received Amount: "+String.valueOf(receivedAmount);
    else
      return "Bill Number: " + String.valueOf(billNumber)+
      "; Required Amount: "+ String.valueOf(requiredAmount)+
        "; Card Number: " + String.valueOf(cardNumber);
   }
}
```

The decision predicate of the decision about getting paying data is realized in the Pay constructor as follows:

```
    @ requires args[0].equals("true");
      @   …
      @ also
      @   requires args[0].equals("false");
      @   …
```

The decision predicate of printing data decision in the getPayInfo method is equivalent with the predicate of the decision about getting paying data:

```
    @ requires payByCash == true;
      @   …
      @ also
      @   requires payByCash == false;
      @   …
```

The decision predicate of the decision about printing data (payByCash variable) is evaluated in the decision options of the other decision (about getting paying data) based on its decision predicate (`args[0].equals("true")`). Therefore the two decision predicates are eqivalent, accordingly the two decisions can be contracted sorting them into the same class hierarchy.

5.2. After decision sorting

The decisions about payment type are sorted into the class hierarchy, where the different paying modes are defined in the subclasses as the decision options. If somebody pays in cash, the number of the bankcard and the transaction number are not required, but the paid and received amounts are required. In case of paying by bankcard, the received and paid amounts are not required, but the bankcard number and the transaction number are needed. After the executing the contraction of the equivalent decisions of the paying mode (which were in the 'Pay' and the 'getPayInfo' methods), the decision about paying mode will be executed just once. This will be enclosed and archived by the 'Pay' class hierarchy and the enclosed decision will be reused on the next occasions.

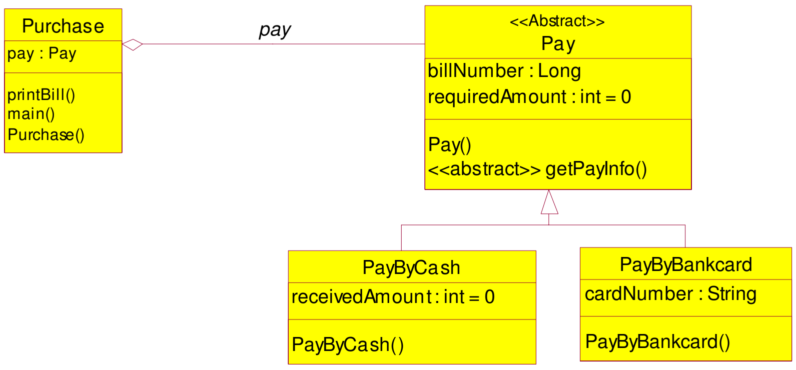

Diagram 4. Classes of the example after decision sorting by UML diagram [9].

```
package hu.decision.example2;
//@ model import org.jmlspecs.models.*;

/** Printing the payment data.
 */
public class Purchase {

  /*@ public static pure model boolean parseable( String s ) {
  @ try { int d = Integer.parseInt(s); return true; }
  @ catch (Exception e) { return false; }}
  @*/

  /*@ public static pure model Pay desidePayingType(String[] args) {
  @ if(args[0].equals("true")||(!args[0].equals("true")&&
    @    !args[0].equals("false")))
  @   return new PayByCash(args);
  @ else
  @   return new PayByBankcard(args);
  @}
  @*/

  /**  Payment data - according to the payment type - is got
   *  using the instance of PayByCash or PayByBankcard class
   */
  public Pay pay;
  //@ instance invariant pay != null;

  public static void main(String[] args) {
      Purchase purchase=new Purchase();
      purchase.init(args);
      purchase.printBill();
  }

  /** Checking the number of arguments and creating the instance
   *  of PayByCash or PayByBankcard class, by which the payment data
   *  is printed.
   */
  /*@
  @ private normal_behavior
  @   requires args==null||args.length<4;
  @   assignable \nothing;
  @   ensures false;
  @ also
  @ private normal_behavior
  @   requires args!=null&&args.length>=4&&pay==desidePayingType(args);
  @ {|
  @     {|
  @     requires pay instanceof PayByCash;
  @     assignable pay, System.out;
  @     ensures pay.billNumber==Integer.parseInt(pay.args[1])&&
  @       pay.requiredAmount==Integer.parseInt(pay.args[2]);
  @     ensures ((PayByCash)pay).receivedAmount==Integer.parseInt(pay.args[3]);
  @   also
  @     requires pay instanceof PayByBankcard;
  @     assignable pay, System.out;
  @     ensures pay.billNumber==Integer.parseInt(pay.args[1])&&
  @       pay.requiredAmount==Integer.parseInt(pay.args[2]);
  @     ensures ((PayByBankcard)pay).cardNumber==pay.args[3];
  @   |}
  @ also
  @   requires !parseable(args[1])||!parseable(args[2])||
  @     !parseable(args[3]);
  @   assignable \nothing;
  @   ensures false;
  @ |}
  @*/
  private void init(String[] args){
    //If there are not enough arguments.
```

```
      if(args == null || args.length < 4 ){
        System.err.println("There are not enough arguments!");
        System.exit(-1);
      }
      try{
        //Creating the Pay object by which the payment behaviours
        //are realized.
        if(args[0].equals("true"))
          pay=new PayByCash(args);
        else if(args[0].equals("false"))
          pay=new PayByBankcard(args);
        else
          pay=new PayByCash(args);
        System.out.println("PayByCash?:(true/false) "+args[0]);
      }
      catch (java.lang.NumberFormatException nfe){
        System.err.println("The format of the Arguments is not appropriate!");
        System.exit(-1);
      }
    }

    /** Based on the pay instance the payment data is printed.
     */
    /*@ private normal_behavior
    @   requires pay instanceof PayByCash;
    @   assignable System.out;
    @   ensures (* Prints the Bill Number, Required Amount,
    @     Received Amount*);
    @ also
    @ private normal_behavior
    @   requires pay instanceof PayByBankcard;
    @   assignable System.out;
    @   ensures (* Prints the Bill Number, Required Amount,
    @     Card Number*);
    @*/
    private void printBill(){
      String payInfo = pay.getPayInfo();
      System.out.println("Payinfo: "+ payInfo);
    }
  }

  /*-----------------------------------------------------------------*/
  package hu.decision.example2;
  //@ model import org.jmlspecs.models.*;

  /** The parent class of payment type class hierarcy.
   *It determines the common data structure and the behaviour
   *of the subclasses (payment types).
   */
  public abstract class Pay {

    /*@ public static pure model boolean parseable( String s ) {
    @ try { int d = Integer.parseInt(s); return true; }
    @ catch (Exception e) { return false; }}
    @*/
    protected /*@ spec_public @*/ String[] args;
    //@ invariant args!=null && args.length==4;
    protected /*@ spec_public @*/ long billNumber = 0;
    //@ private instance initially billNumber == 0;
    protected /*@ spec_public @*/ int requiredAmount=0;
    //@ private instance initially requiredAmount == 0;

    /**
     *Getting the bill number and the required amount, which are the
     *common data structure of payment types.
     */
    /*@ public behavior
    @   requires parseable(args[1])&&parseable(args[2]);
    @   assignable args, billNumber, requiredAmount, System.out;
    @   ensures args==in_args && billNumber==Integer.parseInt(args[1])&&
    @   requiredAmount==Integer.parseInt(args[2]);
    @ also
    @   requires !parseable(args[1])||!parseable(args[2]);
    @   assignable args;
    @   ensures args == in_args;
    @   signals_only java.lang.NumberFormatException;
    @*/
    public Pay(String[] in_args) throws NumberFormatException
    {
      this.args=in_args;
      billNumber = Integer.parseInt(args[1]);
      System.out.println("Bill Number:(Number) "+billNumber);
      requiredAmount = Integer.parseInt(args[2]);
      System.out.println("Required Amount:(Number) "+requiredAmount);
    }

    /** Getting the payment data according to payment type. The behaviour
     *   is realized by the subclasses of the Pay class.
     */
    abstract public String getPayInfo();
  }


  /*-----------------------------------------------------------------*/
  package hu.decision.example2;
  //@ model import org.jmlspecs.models.*;

  /**
   * The PayByBankcard class as the subclass of the Pay class is available,
   * if the customer pays by bankcard as it is decided in the Main method.
   */
  public class PayByBankcard extends Pay{

    public  String cardNumber;

    /** If the customer pays by bankcard,
     *  then getting the card-number is necessary.
     */
    /*@ also
    @ public behavior
    @   requires parseable(args[3]);
    @   assignable cardNumber, System.out;
    @   ensures cardNumber==Integer.parseInt(args[3]);
    @ also
    @   requires !parseable(args[1])||!parseable(args[2]);
    @   assignable args;
    @   ensures args == in_args;
    @   signals_only java.lang.NumberFormatException;
    @*/
    public PayByBankcard(String[] args)
    { super(args);
      cardNumber = args[3];
      System.out.println("cardNumber Amount:(String)"+cardNumber);
    }

    /** Printing the payment data according to the payment type.
     */
    /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result == "Bill Number: "+String.valueOf(billNumber)+
    @   "; Required Amount: "+String.valueOf(requiredAmount)+
    @   "; Card Number: "+String.valueOf(cardNumber);
    @*/
    public String getPayInfo()
    {
      return "Bill Number: " + String.valueOf(billNumber)+
           "; Required Amount: "+ String.valueOf(requiredAmount)+
           "; Card Number: " + String.valueOf(cardNumber);
    }
  }

  /*-------------------------------------------------------------------*/
  package hu.decision.example2;
  //@ model import org.jmlspecs.models.*;

  /**
   * The PayByCash class as the subclass of the Pay class is available,
   * if the customer pays in cash as it is decided in the Main method.
   */
  public class PayByCash extends Pay{

    protected /*@ spec_public @*/ int receivedAmount=0;
    //@ public instance initially receivedAmount == 0;

    /** If the customer pays in cash,
     *  then getting the received amount is necessary.
     */
      /*@ public normal_behavior
    @   requires parseable(args[3]);
    @   assignable receivedAmount ,System.out;
    @   ensures receivedAmount == Integer.parseInt(args[3]);
    @ also
    @ public exceptional_behavior
    @   requires !parseable(args[3]);
    @   assignable receivedAmount, System.out;
    @   signals_only java.lang.NumberFormatException;
    @*/
    public PayByCash(String[] args) throws NumberFormatException
    { super(args);
      receivedAmount = Integer.parseInt(args[3]);
      System.out.println("Received Amount:(Number) "+receivedAmount);
    }

    /** Printing the payment data according to the payment type.
     */
    /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result == "Bill Number: "+String.valueOf(billNumber)+
    @     "; Required Amount: "+String.valueOf(requiredAmount)+
    @     "; Received Amount: "+String.valueOf(receivedAmount);
    @*/
    public String getPayInfo()
    {
      return   "Bill Number: "+String.valueOf(billNumber)+
           "; Required Amount: "+String.valueOf(requiredAmount)+
           "; Received Amount: "+String.valueOf(receivedAmount);
    }
  }
```

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

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

The instantiation can be found in the 'init' method, by which the decision can be enclosed and archived by sorting it referring to an aggregation as a variable (pay object). The archived decision can be used in the next decision occasions without knowing about the result of the decision.

```
    if(args[0].equals("true"))
      pay=new PayByCash(args);
    else if(args[0].equals("false"))
      pay=new PayByBankcard(args);
    else
      pay=new PayByCash(args);
```

The JML foramlization of enlosing:

```
  @ public static pure model Pay desidePayingType(String[] args) {
  @ if(args[0].equals("true")||(!args[0].equals("true")&&
  @    !args[0].equals("false")))
  @    return new PayByCash(args);
  @ else
  @    return new PayByBankcard(args);
  @}
...
  @    requires args!=null&&args.length>=4&&pay==desidePayingType(args);
...
```

The archived decision can be reused in the next decision cases based on pay object as follows:

```
   @      requires pay instanceof PayByCash;
   @      ...
   @    also
   @      requires pay instanceof PayByBankcard;
   @      ...
```

The type of the pay object determines the appropriate decision option for the next decision occasions, accordingly the decision enclosing is realized.

6. Conclusion

The new interpretation of inheritance – as an extension of the old interpretation – is introduced, and described by an example. Accordingly, the aim of the application of inheritance and the object-oriented paradigms is the elimination of decision repetition by sorting the decisions' definitions into class hierarchy. By using the object-oriented paradigms, the consistence of the decisions can be solved making the maintenance of the program easier.

In the Introduction, we showed the properties of well-structured programs. In order that these properties could be examined, the formalization of the decisions is introduced by JML. Based on JML, the non-sorted and sorted states of the decisions can be described realizing the formal differences between them.

We have used the JML formalization method in order to examine the cases of decision repetitions and the relations of complex decisions.

As it was mentioned in [11], there are connections between the decision based interpretation of object-oriented paradigms and Design Patterns, accordingly Design Pattern gives us recipes to eliminate decision redundancy and to archive decisions. As JML is adapted to examine the decisions and the decision repetitions of object-oriented programs – as it was mentioned in this paper – we think JML is adapted to formalize Design Patterns more exactly than the existing formalization tools.

As for the idea – which was introduced by [11] and examined in this paper by

JML formalization – was created in the course of analyzing of Design Patterns, we intend to examine Design Patterns based on JML formalization, and to examine the additional connections between the applicability of Design Patterns and decision repetitions

Based on the new decision-based conception, we can realize more manifest and exact explanations for the aims of Design Patterns. By using the new idea, a new, more natural classification of Design Patterns is described in [11], by which we would like to launch a discussion about a new interpretation of the existing classification [8].

According to our plan, we will examine whether the decision repetition in the design and the source can be eliminated by automatic sorting, that helps to upgrade the quality of the design and the source automatically.

References

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

[2] Brito e Abreu, F., Melo, W., Evaluating the Impact of Object-Oriented Design on Software Quality - *Originally published in Proceedings of the 3rd International Software Metrics Symposium (METRICS'96)*, IEEE, Berlin, Germany, 1996.

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

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

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

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

[7] Moore, I., Automatic Inheritance Hierarchy Restructuring and Method Refactoring, *Conference on Object-Oriented Programming Systems Languages and Applications San Jose*, California, United States, 1996.

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

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

[10] JavaTM 2 Platform Standard Edition, `http://java.sun.com/j2se/1.4.2/docs`

[11] Márien, Sz., Decision Based Examination of Object-Oriented Programming and Design Patterns, *Teaching Mathematics and Computer Science*, Debrecen, Hungary, 2008.

[12] Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K., Rustan, M., Poll, E., An overview of JML tools and applications, 2004.

[13] Leavens, G.T., L. Baker A., Ruby, C., JML: A Notation for Detailed Design, 1999.

[14] Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E., Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2.

[15] Leavens, G.T., Baker, A.L., Ruby, C., Preliminary Design of JML: A Behavioural Interface Specification Language for Java, 2006.

Szabolcs Márien
University of Debrecen
Debrecen, Hungary
e-mail: `marienszＧhotmail.com`