Annales Mathematicae et Informaticae
47 (2017) pp. 149–176
`http://ami.uni-eszterhazy.hu`

Decision structure based object-oriented design principles

Szabolcs Márien

Eszterházy Károly University of Applied Sciences
Institute of Mathematics and Informatics
Eger, Hungary
`szabolcs.marien@innovitech.hu`

Submitted August 24, 2016 — Accepted January 3, 2017

Abstract

The major part of program complexity is based on the logic of conditions, but the existing refactoring methods do not detail the options of decision merging according to the cases of decision redundancies, which are the main options of optimizing the decision structures by refactoring. To extinguish decision redundancies in the source code, we have an option to merge decisions, which can be interpreted as refactoring tools, by which the quality of code structures can be optimized. I intend to complete the definitions of decision, decision raising, and introduce a novel concept, decision merging, based on the concept of behavioural contract. According to the decision merging cases, new design principles can be created. The principle "Using inheritance to dissolve decision redundancy" identifies the cases, when the usage of inheritance as an object-oriented tool is more reasonable than object composition. The other new principle is "Avoid decision redundancy", by which decision redundancies can be eliminated based on the decision merging rules. I initiate new object-oriented metrics as well, giving the opportunity to determine the degree of decision redundancies in the software. The properties of these metrics are analysed empirically.

Keywords: Design principles, metrics, inheritance, decision raising, decision merging, decision redundancy.

1. Introduction

1.1. Optimizing decision structures by refactoring

The refactoring of conditional statements by polymorphic methods ("Replace Conditional with Polymorphism") is an interesting, existing refactoring method [11, 16], where the branches of conditional ("if-then-else") statements can be realized as a class with an abstract polymorphic method, which is overridden by the subclasses. The interface of a decision will be an abstract polymorphic method in the parent class [11]. The advantage of replacing a conditional statement with a polymorph method is prevailed if the conditional statement has equal occurrences in the program. In this case the subclasses are not necessary to be known, which reduces dependencies significantly [11]. Consequently, the introduction of new decision options does not result in the change of places where they are used, only the introduction of a new subclass is necessary [11].

The "Replace Type Code with Subclasses" and the "Replace Type Code with State/Strategy" – as conditional statement specific refactoring methods – are based on the previously described "Replace Conditional with Polymorphism" refactoring method [11]. Furthermore there are the "Move Embellishment to Decorator" and the "Replace Conditional Dispatcher with Command" conditional statement optimizing refactoring methods, which are also based on design patterns [16].

According to my concept, class hierarchies can be viewed as abstract decisions [24, 25], based on which I define decision raising and decision merging as the extended interpretations of decision structure optimization methods. When we define a decision, we give the functionality and/or the data structure (state description) of decision options. Decision predicate decides which decision option will set off at a given decision location [24, 25]. 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 into a binary tree, we do not lose generality (see Section 4, where behavioral contract based definitions of decisions and decision raisings are introduced).

The following rules for avoiding decision redundancy are defined in Section 5:

- Decisions should not Recur Rule 1 (DnR Rule 1): "decisions with equivalent decision predicates and equivalent or partly equivalent data structures and behaviors should not recur"
- Decisions should not Recur Rule 2 (DnR Rule 2): "decisions with equivalent decision predicates that define diverse data structures and behaviors should not recur".

I will introduce the decision merging cases as new refactoring tools (see Section 6), the help of which the defined decision redundancies (see Section 5) can be eliminated. The cases of decision merging are determined based on the cases of decision redundancies. The merging of nonraised decision can be realized after raising decisions. The decision raising is a transformation method, by which the decisions can be defined by class hierarchies without conditional statements. The subclasses of hierarchies define the decision options, where the interfaces of decisions are the polymorph methods of parent class. There are the following cases of decision merging:

- The merging/partial merging of fully or partially equivalent decisions: Decision merging/partial merging is necessary if the decision predicates of decisions are equivalent, and decision option declared data structures and behaviors are equivalent or partially equivalent.
- The merging of decisions with equivalent decision predicates and nonequivalent behavioral contracts: If there are two nonraised or raised decisions, which have equivalent decision predicates, then these decisions can be merged.

1.2. Behavioral contracts

In order to examine decision structures and optimization transformations based on the optimization cases (when the transformations are justified) the introduction of the behavioral contracts of decision structures is necessary. There is a contract ("Design by contract" – DBC) between a class and its client, according to which the client has to realize the declared conditions in the course of the calling method of the class. In addition, the class guarantees specified conditions in the course of the returning of the method, which specifies the required behavior of the method [18]. The behavior of a program/object can be specified by a behavioral contract [18, 27], where the contract declares a set of possible behaviors [27] ("Concept of a behavioral contract" – "Design by Contract"). These contracts can be specified by the pre- and post-conditions of methods. Method behavior independent conditions, which are always satisfied, can be specified as class invariants, which describe the general aspects of the behavior contracts of classes [22]. Accordingly invariants specify the general constraints of the values of class variables. General state transition constraints can be specified by history constraints [27]. The parts of state describing data structures which are changed by methods as state transitions are declared by frame conditions [27].

1.3. Object-oriented design principles

Cohesion, coupling and complexity are highlighted as the main targets of quality ensuring metrics [1]. Cohesion examines the inner-consistencies of parts [8]. Accordingly cohesion determines the collaboration levels of elements inside modules. In case of high cohesion, the major part of components realizes the same functionality [1, 5]. In conformity with this, the functional cohesion of a component is high if it serves a properly encompassed behaviour [4]. In case of good program design the cohesion of program structure is high and its coupling is low [8]. According to Wand and Weber's coupling definition [30], there is coupling between two "things" if there is at least one connection between their state histories. The strongest type of coupling is inheritance. Based on loose coupling, the realization of independent system components can be facilitated, meaning that the changing and the maintenance of programs become easier. The aim of object-oriented design principles is to eliminate of dependencies and couplings, to increase cohesion and to decrease complexity. Using object-oriented design principles as abstract concepts, the mentioned designing failures can be avoided.

Liskov substitution principle – LSP [12, 19, 27, 28]: The LSP was extended by Schreiner, who added that subtypes could specialize and refine the parent-type declared contracts, which does not violate the definitions of subtype and substitution. With respect to behavior contracts, it means the following: the preconditions of the subtype are not stronger than in the parent type, the post conditions of the subtype are not weaker than in the parent type, the invariants of the parent-type-based member variables of the subtype are not weaker than in the parent type, the subtype realizes the history constraints of the parent type [27].

Favor object composition over class inheritance [12]: In consideration of reusability and maintainability the appropriate usage of composition vs. inheritance is a critical issue. The aim is a harmonic class and object structure. Inheritance is the tightest couple between classes, which can be used only in well-defined cases.

Single Responsibility Principle [20]: In the course of determining class behavior, we have to take separated task responsibility into consideration. The determined class behavior should be responsible for "one task", other classes need to be introduced in order to supply other tasks. If we don't take it into consideration, and a single class realizes more tasks, then if one of the task behaviors needs to be changed, it may result in the change of the behavior of the other tasks as a side effect, generating more risk and cost.

The issues of the previously mentioned object-oriented design principles are concerned by the concept of decision merging. Accordingly decision merging (discussed in section 6) can support the issue of the separation of class behaviors into subclasses ("Liskov substitution principle" [12, 19, 27, 28], "Single Responsibility Principle" [20]). Furthermore, decision redundancies and decision merging cases (which eliminate decision redundancies) support the appropriate usage of inheritance and object composition ("Favor object composition over class inheritance" [12]). According to the cases of decision redundancies and the decision merging rules, two object-oriented design principles are created (see Section 3):

- "Using inheritance to dissolve decision redundancy": If one case of decision redundancies induces the usage of decision merging and decision raisings transformations, then the usage of inheritance is confirmed.

- "Avoid decision redundancy": If there are decision redundancies in a source code, then based on the decision merging rules the decision redundancies need to be eliminated.

These new design principles are useful when deciding whether the usage of inheritance or object composition can be confirmed, which is one of the subjects of this paper.

1.4. Object-oriented design metrics

The main aspects of the quality insurance of program developing are maintainability, extendibility, intelligibility, reusability [10] and changeability [15].

Six object-oriented design metrics were specified by Chidamber and Kemerer [8]. These metrics are the following: WMC – "Weighted methods per class", DIT – "Depth of inheritance tree", NOC – "Number of children", CBO – "Coupling between objects", RFC – "Response for a class", LCOM – "Lack of Cohesion in Methods".

One of the first metrics, and at the same time probably the most determinative cohesion metric is the LCOM – "Lack of Cohesion in Methods" metric [8]. The interpretation of this metric is based on dependencies between methods, which can be determined by the sets of the method-used member variables of classes.

Eder et al. introduced the concepts of method cohesion, class cohesion and inheritance cohesion [10]. However, the specified cohesion measuring methods require semantic analysis, which obstructs the industrial usage of them.

Badri et al. analysed the correlation between coupling and cohesion [1], which has not been justified previously. However the correlation between them was mentioned several times. According to these, the high (low) cohesion is associated with the low (high) coupling values [17]. They measured the correlation between their new cohesion metric and coupling metric by empirical analysis. Meeting the requirements they revealed a significant negative correlation between their new cohesion metric and the CBO [7, 8] coupling metric [1].

Several measurements have tried to confirm the correlation between the different metrics and the changeability aspects of programs, in many cases without successfully detecting the correlations between them [15]. Chae and Kwon stated that the existing cohesion metrics will not be good measuring indicators of changeability until such cohesion metrics that can measure the cohesion properties appropriately are realized [6].

Complexity metrics facilitate the detection of complex objects and classes. Implementation, testing and verification efforts are higher in case of classes with high complexity [31]. Some examples of complexity metrics are listed in [31]: "Cyclomatic Complexity" [21], "Depth of Inheritance Tree" (DIT) [8], "Number of Children" (NOC) [8], "Weighted Methods Complexity" (WMC) [8]. These complexity metrics are based on the static aspects of systems (for example class diagrams, source codes), so they are static metrics [31].

Munson and Khoshgoftaar introduced dynamic complexity metrics [23]. They separated the concepts of "Operational complexity" and "Functional complexity" [23]. The "Operational complexity" of objects uses the "Cyclomatic Complexity" metric [21], which is based on the "Control Flow Graph" [31].

The Qi ("Quality Indicator") [2] metric – similarly to the previously mentioned metric – is based on the branches of programs, and with its help, various software attributes such as complexity, cohesion and coupling can be examined together. It uses controlling paths and their probabilities. One curiosity is the appearance of the representations of polymorphic callings with graphs, where the opportunities of dynamic couplings – or method callings – are represented by graph edges. Branches show polymorphic possibilities.

2. Motivation

The appropriate usage of inheritance is the key point of object-oriented programming, the clarifying of this question (inheritance vs. object composition/aggregation) is the aim of numerous design principles [12, 19, 27, 28] and design patterns [12, 16]. At the same time, in spite of these clarifying attempts, it is not obvious, which tool of the object-oriented technology (inheritance, object composition, aggregation) should be used in different cases.

Fowler et al. specified the "Replace Conditional with Polymorphism" refactoring method [11], by which the interpretation of inheritance was extended. It states that replacing is necessary if there are equal conditional statements in a program [11]. Additional details about the necessity of using this refactoring method are not elaborated. The merging method of the concept of "Parallel Inheritance Hierarchies" [11] eliminates the redundancies of the declaration and/or the usage of class hierarchies by defining them as raised decisions. It describes the cases where merging is necessary as follows: The merging of class hierarchies is necessary if the changing of one hierarchy results in the changing of another one [11]. The details of the "Move Embellishment to Decorator" and the "Replace Conditional Dispatcher with Command" refactoring methods [16] don't describe the decision structures that would help to recognize the necessity of the using of the "Decorator" and the "Command" Design Patterns [12].

We must see that the description of the cases of decision redundancies – which realize the necessity of decision raising and/or decision merging – is incomplete. In order to complement this, I specified the cases of decision redundancies [24, 25] (see Section 5), and I clarify it, where the raising and/or the merging of decisions are justified (see Section 6).

In order to achieve better program quality and structure, many object-oriented design principles were formulated that provide quality improvement by increasing cohesion between program units, decreasing coupling and eliminating dependencies. The LSP [12, 19, 27, 28] analyses inheritance quality. Its definition is based on behavior contracts [18, 27], which are the bases of the definitions of decision, decision raising and decision merging. The LSP is the most elaborated design principle, by the help of which the cases when inheritance relation can be used between two types can be determined. At the same time it doesn't help the detection of cases when the introduction of inheritances is confirmed by program structures. The "Favor object composition over class inheritance" [12] design principle is not accurately defined, its theoretical background is not elaborated. The principle tries to give intuitive, practical suggestions in connection with the question of using inheritance and object composition. The "Single Responsibility Principle" [20] is related to cohesion [9, 20]. In the course of defining this principle the use of behavior contracts is suggested, based on which the LSP principle was extended, and furthermore, it is used in this paper as well.

The known design principles are concerned with the critical issue of which object-oriented construction's usage is justified in different programming cases. However, in my opinion, the clarification of this question is necessary, therefore, based on my previously mentioned concept, I define new design principles to answer this issue (see Section 3).

In conformity with an examination, which is based on two metrics Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC) [3], the conclusion is the following: Those classes have lower cohesion, which frequently use inheritances [3]. The experienced inverse dependencies between inheritance based code reusability and cohesion [3] indicate the unclarified status of the usability of appropriate inheritance. Accordingly, based on the Lack of Cohesion in Methods (LCOM) [8], the cohesion specific influences of inheritances are not taken into account [3, 5, 6, 13, 14, 15]. We must note that the aim of inheritances is not reusing, but the extension of the functionality of the classes with specific behaviour. Accordingly the reusing specific application of inheritance can result in the decrease of the optimal structure of the code. In order to promote the appropriate usage of inheritance, there are numerous concepts as I described above. In order to clarify this question I introduce a new concept (decision merging) about the use of inheritance in this paper. Intuitively, we must see that the class-subclass inheritance structures with optimized decision structures – which are resulted based on the eliminations of decision redundancies using decision merging cases – contribute to the decrease of the divisibility of classes, by which the cohesion of classes can be increased (see the empirical validations in Section 8).

Beside the supposed similarity between cohesion metrics and the new measuring method introduced in this paper, I also find the comparison of these new metrics and complexity metrics necessary. The reason for this is there are more complexity metrics which examine the branches of the conditional statements of object-oriented classes. Some of these metrics are the following: "Cyclomatic Complexity" – CC [21, 31], "Weighted Methods Complexity" – WMC [8], "Operational Complexity", "Functional Complexity" [23, 31], "Quality Indicator" – Qi [2]. Furthermore, there are additional C&K metrics [8] which describe the inheritance structure of programs. They are the following: "Depth of Inheritance Tree" (DIT) [8], "Number of Children" (NOC) [8]. These metrics are interesting in the consideration of the appropriate usage of inheritances according to the concepts specified in this paper.

Note that there is no complexity metric which would consider the structurally critical question of whether complexity growing conditional statements and inheritance structures are used appropriately, or whether the structures could be optimized. The method complexity which is measured by the CC metric shows the complexity of tasks, which is realized by the method. The high value of method complexity is not necessarily the sign of wrong code structures, it only shows the complexity of tasks. There is a similar conclusion according to the WMC. The "Operational Complexity" (OCPX) [23, 31] metric is based on dynamic complexity, which takes the CC of running paths into consideration. Therefore, according to the previously mentioned metrics, it can't be used to measure the quality of the decision structures.

I initiate such new object-oriented metrics that give opportunity to determine the rate of decision redundancies in the source code of a program (see Section 7). In order to determine the relationship between the "Metric of decision abstraction" (MDA), the "Ratio of inheritances coming into existence by the elimination of decision redundancies" (RIEDR) metrics and the level of code integrity, I analysed 10-10 states of several open source projects empirically.

To summarize, the following questions have to been answered:

In which cases can we talk about decision redundancies?

This is the most important question from the point of view of my topic. We need to clarify the cases where the use of decision merging is justified. In order to clarify this question the following metrics are introduced: "Metric of decision abstraction" (MDA), "Ratio of equivalent decision cases" (REDC), "Ratio of decision cases with equivalent decision predicates" (RDCEDP). Furthermore, the "Avoid decision redundancy" design principle is defined, by which the elimination of decision redundancies is targeted based on decision merging cases.

In which cases can we use inheritance?

Beyond the previously defined general aspect the aim of the inheritance-specific question is to clarify whether the using cases of inheritances are justified in the source code. We must see that there are overlaps between this and the previously mentioned questions, but in consideration of the prominent role of inheritances it must be specified separately. This question is answered by one of the introduced object-oriented design principles, namely it is the "Using inheritance to dissolve decision redundancy". This principle clarifies the appropriate usage of inheritances based on the decision redundancy cases. In order to determine the scale of appropriate inheritance usage, the "Ratio of inheritances coming into existence by the elimination of decision redundancies" (RIEDR) metric is introduced, by which the polymorph methods are analysed in the inheritances.

In order to clarify these questions, the cases of decision redundancies and decisions merging are defined. In the course of the evaluation of the new metrics I analyse the correlations between decision redundancies and cohesion, complexity and coupling, by which we can notice their relations with the general code quality aspects.

3. Extending the object-oriented design principles

In the following I suggest two new object-oriented design principles, one of which unequivocally tries to highlight those cases, where the use of inheritance is justified ("Using inheritance to dissolve decision redundancy") complementing the "Favor object composition over class inheritance" design principle [12]. Furthermore, I try to determine a general program structure organizing principle, which – beyond the subject of the appropriate usage of inheritance – helps to find optimal structures ("Avoid decision redundancy"). The new design principles contain – as a part of their definitions – the rules of decision merging (see Section 6), by the help of which decision redundancies can be avoided (see Section 5). Based on the cases of decision redundancies according to the decision merging rules, the new object-oriented design principles are the following.

3.1. Using inheritance to dissolve decision redundancy

According to the definitions of decision, decision raising and decision merging, the aim of inheritances is to define decisions in an abstract form, based on which the facility of decision merging can be realized. The use of inheritance is justified if the decision structure based dependencies confirm its usage, that is if one case of decision redundancies which justifies the usage of decision merging is fulfilled. In these cases, the elimination of decision redundancies can be realized by one of the decision merging rules.

3.2. Avoid decision redundancy

If the use of decision merging is confirmed by decision redundancies, then the decision redundancies have to be eliminated based on the decision merging rules. Using this principle, according to the rules of avoiding decision redundancies, a more optimal decision structure can be achieved. This principle determines generally the optimization facilities of decision structures based on decision redundancies and decision merging rules, accordingly it helps determine the using facilities of inheritance. It complement the "Using inheritance to dissolve decision redundancy" principle, which approaches this issue from the appropriate usage of inheritance.

3.3. Comparing the new principles with other ones

The "Using inheritance to dissolve decision redundancy" and the "Avoid decision redundancy" principles specify the cases accurately based on decision redundancies and the decision merging rules, where the use of inheritances is necessary. It complements the "Favor object composition over class inheritance" design principle [12], where the using facility can be decided based on some intuitive concepts. The LSP specifies the relationships between the type and the subtype, but it doesn't mention anything about the initial structures, where the introduction of inheritance is necessary. Based on the new design principles we can detect those structural surroundings, where the inheritances can resolve the decision redundancies. The "Single Responsibility Principle" [20] is the principle of cohesion [9]. Furthermore the eliminations of decision redundancies increase the cohesions (see the empirical evaluation of new metrics in Section 8). Therefore the fulfilment of "Single Responsibility Principle" can be facilitated by using the new principles to reduce decision redundancies.

4. The definitions of decision and decision raising according to behavioral contracts

The decision structure of programs is defined irrespectively from the program implementation. The realization of this structure strongly determines the optimization level of programs. Decision structures can be optimized by different transformations, by which the behavioral aspects of programs are not changed. In order to examine decision structures, transformation methods and optimization cases (when transformations are required) the introduction of the following concepts is necessary.

4.1. Behavior of decision

The behavior of the decision options D_{O_1}, D_{O_2} of decision D can be declared by behavioral contracts $C_{D_{O_1}}$, $C_{D_{O_2}}$ [18, 27]. (The behavior of a decision option is declared by one behavioral contract.) The D_{O_1}, D_{O_2} are the implementations of decision options, which have to realize the declared decision requirements $(C_{D_{O_1}}, C_{D_{O_2}})$. The changing of decision structure implementations does not always result in the altering of behavioral contracts.

The behavioral contracts of decision options declare the pre-conditions of decision options as their decision predicates and the post-conditions of decision options as state transitions. Behavioral contracts define the data structures, on which the state transitions of behavioral contracts are interpreted. The invariants [19] – which narrow the state-space of behavioral contracts – and the history constraints [19] – which define general state-transitions – are handled as parts of the pre- and post-conditions of decision options. The interpretations of these as invariants and history constraints are not important in consideration of the behavioral contracts of decision options.

A decision case is one case of a decision, in the course of which an appropriate decision option is selected based on the evaluation of its decision predicate. According to a selected decision option, a decision option specified state transition is executed, by which the modifications (the modification or/and the extension of the state) are realized based on the data structure of the decision option (the concerning part of the state description).

4.2. Decision raising

It is a transformation, by which decision dependencies can be eliminated. After using this transformation, the behavior and the data structures of decision options are defined by class hierarchies without using "if-then-else" statements. The subclasses of class hierarchies define the decision options, which are integrated by parent classes. The "interface" of a decision is a polymorph method of a parent class, which has to be overridden by its subclasses [24, 25]. After decision raising decisions can be implemented – without "if-then-else" statements – with references which have the same type as the parent class of decision declaration class hierarchies. They refer to the instances of the subclasses of class hierarchies according to the appropriate decision options [24, 25].

Let D_{NR} be a nonraised decision, where it's decision options $D_{NR_{O_1}}$, $D_{NR_{O_2}}$ implement the behavioral contracts $C_{D_{NR_{O_1}}}$, $C_{D_{NR_{O_2}}}$. The decision D_R is the raised decision of the decision D_{NR} if the behavioral contracts $C_{D_{R_{O_1}}}$, $C_{D_{R_{O_2}}}$ of the decision options $D_{R_{O_1}}$, $D_{R_{O_2}}$ of the decision D_R are defined according to the following: $C_{D_{R_{O_1}}} = C_{D_{NR_{O_1}}}$, $C_{D_{R_{O_2}}} = C_{D_{NR_{O_2}}}$. It means that the behavioral contracts of the decision options of nonraised and raised decisions are equivalent.

If the decisions of decision cases have already been raised, there are two types of decision cases: initial decision cases and reusing decision cases. In the course of initial decision cases, a decision option is archived by using a reference. The type of this reference is the parent class of the class hierarchy of a raised decision. This reference points to an instance of the subclass of the selected decision option. In the course of the following decision cases (reusing decision cases), the result of initial decision case is reused based on the calling polymorph method of archiving polymorph reference without the need to re-evaluate the decision. In case of nonraised decisions, the reevaluation of the decisions is necessary, but in case of raised decisions, the archived decisions can be reused (reusing decision cases), so the re-evaluation of the decisions is not necessary.

5. Avoiding decision redundancies

We must see that decision raisings are reasonable if existing or expected decision redundancies can be eliminated. These redundancies result in implementation dependencies that reduce the maintainability and transparency of codes. The conditions of avoiding decision redundancies are the following:

- Decisions should not Recur Rule 1 (DnR Rule 1): Decisions with equivalent decision predicates and equivalent or partly equivalent data structures and behaviors should not recur, so the equivalent or partly equivalent decision should not be realized again in the course of the same running. Therefore, the declarations of the behavioral and the data structure aspects of such decisions should be defined at one place.

- Decisions should not Recur Rule 2 (DnR Rule 2): Decisions with equivalent decision predicates that define diverse data structures and behaviors should not recur. Accordingly such decisions have to be defined in merged forms at one place.

The aim of avoiding decision redundancies is the reduction of decision dependency. Decision dependency can be interpreted as a type of implementation dependency, which is based on the decision structure of programs. If the change of the behavioral contracts of decision options or the introduction of new decision options forces changes in several decision cases, then there is a decision dependency. Using raised and merged decisions (see Section 6) only the initial decision case needs to be changed if the behavior of a decision option is changed or a new decision option is introduced, the changing of other decision cases is not necessary. Inheritance is used rightfully if decision structure dependencies make it reasonable.

6. Decision merging

Decision merging is the tool of eliminating decision redundancies, which can be interpreted as a new refactoring tool. The cases of decision merging are based on the cases of decision redundancies. I use Liskov's subtype-parent type substitution principle [19] based on behavioral contracts [27].

The behavioral contract C' is the strengthening – in my interpretation, the *real refinement* – of the behavioral contract C: $C' \supset C$ (pronounced as: the behavioral contract C is implicated from the behavioral contract C') if the behavioral contract C' realizes the requirements of the behavioral contract C, but it specifies additional statements as well. *Real refinement* means "strengthening" for post-conditions, but it means "weakening" for pre-conditions. In case of the behavioral contracts of decisions pre-conditions as decision predicates cannot be changed.

The behavioral contract C' is the *refinement* of the behavioral contract C: $C' \supseteq C$ (pronounced as: the behavioral contract C is implicated from or is equal to the behavioral contract C') if the behavioral contract C' realizes the requirements of the behavioral contract C, but it specifies additional statements as well, or their behavioral contracts are equal. Regarding the post-conditions, refinement means equivalence (keeping conditions), or "strengthening" (realizing additional conditions). Regarding the pre-conditions, it means equivalence or "weakening". Pre-conditions as decision predicates cannot be changed.

We must see that there may be partial or total equivalence in the behavioral contracts of nonraised and raised decisions, and in such cases in order to eliminate decision redundancies the using of partial or total decision merging is justified. It is important to note that the merging of nonraised decisions can be realized after raising decisions into class-subclass structures. Decision merging may also be necessary in case of raised decisions, which means that there are decisions merging cases where raised decisions need to be merged. In order to determine whether two decisions can be merged, the behavioral contracts of decisions need to be examined, based on which the fulfillment of one case of decision merging rules can be determined.

According to the previously mentioned rules of avoiding decision redundancies, I describe the conditions where the use of decision merging or partial decision merging is justified below.

6.1. The merging/partial merging of fully or partially equivalent decisions

Decision merging/partial merging is necessary if the decision predicates of decisions are equivalent, and decision option declared data structures and behaviors are equivalent or partially equivalent, which means that one of them extends the other or both of them extend a common part. Evidently if there are raised decisions, which complete the conditions of decision merging, the merging must be executed.

6.1.1. Merging of equivalent or extending decisions ("Decision merging")

Two decisions can be merged in the following cases: If the decision options of two decisions realize equivalent behavioral contracts. If the behavioral contract of one decision is refined, strengthened by the behavioral contract of the other decision.

Let there be decisions D_1, D_2 and decision options $D_{1_{O_1}}, D_{1_{O_2}}, D_{2_{O_1}}, D_{2_{O_2}}$, which realize the behavioral contracts $C_{D_{1_{O_1}}}, C_{D_{1_{O_2}}}, C_{D_{2_{O_1}}}, C_{D_{2_{O_2}}}$. The decisions D_1, D_2 can be merged if there are such behavioral contracts $C_{D_{O_1}}, C_{D_{O_2}}$ for which the following are true:

$$C_{D_{O_1}} = C_{D_{1_{O_1}}}, \quad C_{D_{O_1}} \subseteq C_{D_{2_{O_1}}}, \quad C_{D_{O_2}} = C_{D_{1_{O_2}}}, \quad C_{D_{O_2}} \subseteq C_{D_{2_{O_2}}}.$$

Accordingly if the behavioral contracts $(C_{D_{2_{O_1}}}, C_{D_{2_{O_2}}})$ of one of the decisions that are merged match or refine/strengthen the behavioral contracts $(C_{D_{1_{O_1}}}, C_{D_{1_{O_2}}})$ of the other decision, then the behavioral contracts $(C_{D_{O_1}}, C_{D_{O_2}})$ of the merged decision are equivalent with the behavioral contracts of one of the merging decisions, and the behavioral contracts of the other decision are the refinements of the behavioral contracts of the merged decision.

Therefore, we have to examine the equivalence of the data structures and behaviors of decisions, paying attention to the partial equivalence if one is the refinement of the other one. For these reasons decisions can be merged if their data structures and behaviors perform the following:

- Concerning data structure: The sets of available states based on the variables of decisions are equal, or the states based on the data structure of one of the decisions are a subset of the data structure based states of the other decision.
- Concerning behavior: If the pre- and post-conditions as the behaviors of the decision options of decisions are equal, or one of the decisions declares additional post-conditions while pre-conditions are unchanged.

6.1.2. Merging of partially equivalent decisions ("Decision partial merging")

The partial merging of two decisions is possible if the behavioral contracts of decisions have an equal common part, which is extended with additional conditions by both of the merging decisions. These additional conditions are not part of the merging.

Let there be decisions D_1, D_2 and decision options $D_{1_{O_1}}, D_{1_{O_2}}, D_{2_{O_1}}, D_{2_{O_2}}$, which perform the behavioral contracts $C_{D_{1_{O_1}}}, C_{D_{1_{O_2}}}, C_{D_{2_{O_1}}}, C_{D_{2_{O_2}}}$. The decisions D_1, D_2 can be merged partially if the behavioral contracts $C_{D_{O_1}}, C_{D_{O_2}}$ can be described according to the following:

$$C_{D_{O_1}} \subset C_{D_{1_{O_1}}}, \quad C_{D_{O_1}} \subset C_{D_{2_{O_1}}}, \quad C_{D_{O_2}} \subseteq C_{D_{1_{O_2}}}, \quad C_{D_{O_2}} \subseteq C_{D_{2_{O_2}}}.$$

That is, take the separated common parts $(C_{D_{O_1}}, C_{D_{O_2}})$ of the behavioral contracts of the decision options of the decisions D_1 and D_2, which are to be merged. The behavioral contracts of the decision options of decisions D_1, D_2 are the real-refinements or refinements of the behavioral contracts $C_{D_{O_1}}, C_{D_{O_2}}$ of decision D which is the common part of the decisions D_1 and D_2. It means that at least one of the behavioral contracts of the decision options of every merged decision has a real-refinement connection. (If refinement relations were allowed for every merging behavioral contract, then those cases would be interpreted as partial merging where the behavioral contracts of merging decision options are equivalent, or where one of the decisions extends the other decision. However, these are the cases of the "Merging of equivalent or extending decisions".)

It can be stated that a common part is an intersection of the behavioral contracts of merging decisions, so the following must be met:

$$\begin{aligned} C_{D_{1_{O_1}}} &= C_{D_{O_1}} \wedge C_{D_{1_{O_1}}\text{extend}}, \quad & C_{D_{1_{O_2}}} &= C_{D_{O_2}} \wedge C_{D_{1_{O_2}}\text{extend}} \\ C_{D_{2_{O_1}}} &= C_{D_{O_1}} \wedge C_{D_{2_{O_1}}\text{extend}}, \quad & C_{D_{2_{O_2}}} &= C_{D_{O_2}} \wedge C_{D_{2_{O_2}}\text{extend}} \end{aligned}$$

where the behavioral contracts $C_{D_{1_{O_1}}\text{extend}}, C_{D_{1_{O_2}}\text{extend}}, C_{D_{2_{O_1}}\text{extend}}, C_{D_{2_{O_2}}\text{extend}}$ determine the decision specific aspects of merging decisions.

Accordingly the data structures and the behaviors of decisions must be examined in order to determine whether there is a common part. So decisions can be merged if their data structures and behaviors perform the following:

- Concerning data structure: The state sets which are realized based on the data variables of decisions have an intersection. It means that there is a common part, which is extended by the state sets of the examined decisions.
- Concerning behavior: The post-conditions of decision options specify additional conditions in relation to the post-conditions of a common behavior with equivalent pre-conditions.

6.1.3. Demonstration of merging equivalent or extending decisions

Decisions, decision options, decision predicates are indicated according to the following: Decisions: D1, D2. Merged decision: D. The decision predicates of the decisions D1 and D2: D1P, D2P. The decision options of the decisions D1 and D2: D1.D1_O1, D1.D1_O2, D2.D2_O1, D2.D2_O2. The decision options of the merged decision D: D_O1.D_O, D_O2.D_O, D_O1.D1_O, D_O1.D2_O, D_O2.D1_O, D_O2.D2_O.

I show the facilities of the merging of equivalent decisions based on Activity, Class and Sequence UML diagrams [26]. In the course of demonstrating, the "Merging of partially equivalent decisions" case is avoided. The decision structure of equivalent decisions can be represented with an Activity diagram [26] (Figure 1).

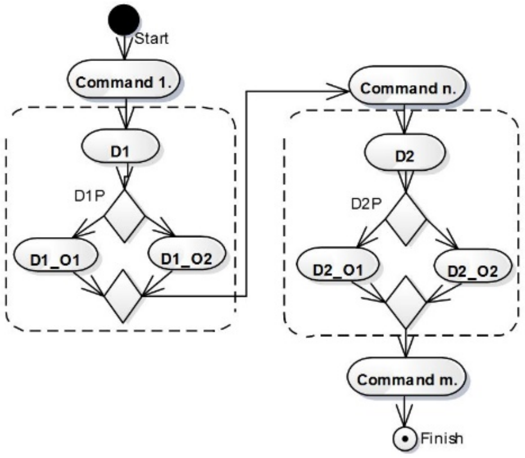

Figure 1: The decision structure of decisions

According to the decision structure, the equivalencies are the following: D1P = D2P, the decision predicates are equivalent. The behavior contracts of the decision options D1_O1 and D2_O1 are equivalent. The behavior contracts of the decision options D1_O2 and D2_O2 are equivalent.

The case of nonmerged decisions can be modeled as follows: One of the possible implementation cases is when separated methods implement the behavior of the decision options of decisions (see Figure 2).

The two cases which are implied from the equivalent decision predicates can be demonstrated with the sequence diagrams in Figure 2. According to these, the similar decision options have to be selected and executed in the course of the same running.

The case of merged decisions can be represented by a class and a sequence diagrams (Figure 3). So the decision option specific operation can be specified by one sequence diagram, on which decision specific behavior is not shown, because it is obscured by polymorph functioning.

The classes and subclasses which represent the merged decisions fulfill the following: The behavior contract of the method D_O1.D_O is equivalent with the behavior contracts of the methods D1.D1_O1, D2.D2_O1, which represent the decision options. The behavior contract of the method D_O2.D_O is equivalent with the behavior contracts of the methods D1.D1_O2, D2.D2_O2, which represent the decision options.

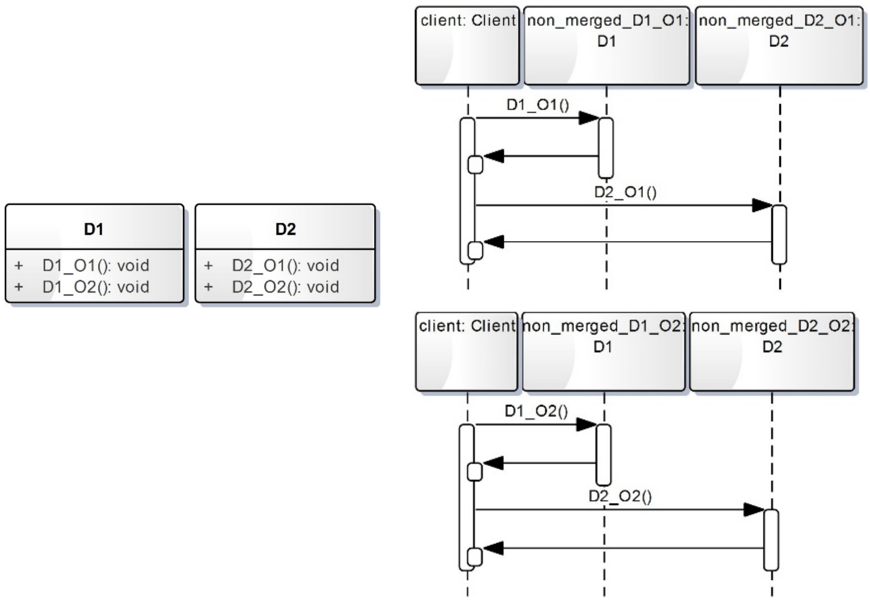

Figure 2: The class diagram of the implementation of the decisions D1 and D2 before decision merging and the sequence diagrams of the decision cases of the decisions D1, D2

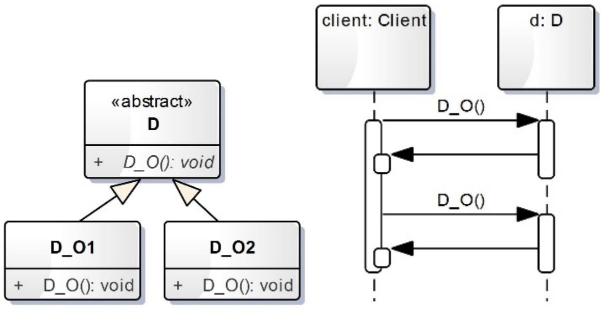

Figure 3: The class diagram representation of the parent class – subclass relationship of merged decisions and the decision cases of merged decisions represented by a sequence diagram

6.2. The merging of decisions with equivalent decision predicates and non-equivalent behavioral contracts

If there are two nonraised or raised decisions, which have equivalent decision predicates, then these decisions can be merged. Accordingly decisions can be merged if their data structures and post conditions are not equivalent from the behavioral aspect, only their decision predicates as preconditions are equivalent.

The decision predicates of the decisions are equivalent: $P_{D_1} = P_{D_2}$ if the decision predicates are equivalent based on the program behavior for every valued-states of state rows/executions. The state of decision predicate is valued if the expression of the decision predicate is evaluated.

Let there be decisions D_1, D_2 and their decision options: $D_{1_{O_1}}$, $D_{1_{O_2}}$, $D_{2_{O_1}}$, $D_{2_{O_2}}$ realize the behavioral contracts $C_{D_{1_{O_1}}}, C_{D_{1_{O_2}}}, C_{D_{2_{O_1}}}, C_{D_{2_{O_2}}}$. The decisions D_1, D_2 can be merged by a decision D with its decision options D_{O_1}, D_{O_2} if there are behavioral contracts $C_{D_{O_1}}, C_{D_{O_2}}$ according to the decision options D_{O_1}, D_{O_2}, which are the disjunctions of the behavior contracts of merged decisions:

$$C_{D_{O_1}} = C_{D_{1_{O_1}}} \vee C_{D_{2_{O_1}}}, \quad C_{D_{O_2}} = C_{D_{1_{O_2}}} \vee C_{D_{2_{O_2}}}.$$

The decision predicates – as the parts of behavior contracts – fulfill the following:

$$P_D = P_{D_1} = P_{D_2},$$

where P_D is the decision predicate of the decision D, furthermore, P_{D_1}, P_{D_2} indicate the decision predicates of the decisions D_1, D_2.

In the following, I show the facilities of decisions with equivalent decision predicates, but different behavior contracts. It is based on Activity, Class and Sequence UML diagrams [26]. The Activity diagram of decisions with equivalent decision predicates equals to the Activity diagram of the decision structure of equivalent decisions. Furthermore, the Class and Sequence diagrams of nonmerged decisions with equivalent decision predicates are equal to the Class and Sequence diagrams of nonmerged equivalent decisions.

According to the decision structure (see Figure 1), the equivalencies are the following: D1P = D2P, the decision predicates are equivalent. The behavior contracts of the decision options D1O1 and D2O1 are NOT equivalent. The behavior contracts of the decision options D1O2 and D2O2 are NOT equivalent.

The case of nonmerged decisions can be modeled according to the following: One of the possible implementation cases is when separated methods implement the behavior of the decision options of decisions. The behavior of methods which represent the decision options is not equivalent (see Figure 2).

The two cases which are implied from the equivalent decision predicates can be demonstrated with the sequence diagrams in Figure 2. According to these – contrary to the previously mentioned decision merging case – the behavior of the executed decision options is not equivalent in the course of the same running. The case of merged decisions can be represented by the class and sequence diagrams of Figure 4.

The class diagram shows how the behavior of two decisions can be defined parallelly in the same subclass, and how the abstract methods represent them in the parent class. The sequence diagram demonstrates the cases, where the different decision options of merged decisions are executed according to the equivalent decision predicates.

The classes and subclasses which represent the merged decisions fulfill the following: The behavior contract of the method D_O1.D1_O / D_O1.D2_O is

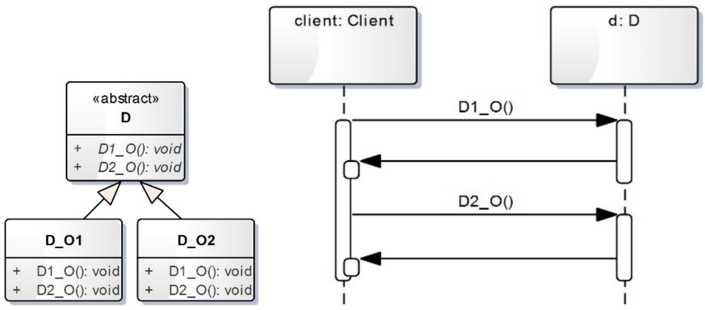

Figure 4: The class diagram representation of the parent class – subclass relationship of merged decisions (with equivalent decision predicates, but different behavior) and the decision cases of merged decisions represented by a sequence diagram

equivalent with the behavior contract of the method D1.D1_O1 / D2.D2_O1. The behavior contract of the method D_O2.D1_O / D_O2.D2_O is equivalent with the behavior contract of the method D1.D1_O2 / D2.D2_O2.

7. The introduction of metrics

The cases of avoiding decision redundancies and the definitions of decision merging specify designing viewpoints that need to be measured and for which measuring methods need to be defined. Accordingly the new metrics which are specified in the following measure the performance of the introduced, new design principles and the decision merging cases which are the theoretical backgrounds of them.

"Metric of decision abstraction": (MDA)

The new metric represents the ratio of polymorph decision cases and the total number of decision cases.

$$\frac{N_{PDC}}{N_{DC}}$$

N_{PDC} – The number of "Polymorph decision cases".
N_{DC} – The total number of "Decision cases".

Under the "Polymorph decision cases" I mean the following: After a decision raising, a decision is realized in a class hierarchy with its classes and subclasses. At the places of use, the callings of polymorph methods represent the decision cases which are called through parent class typed references. Accordingly every polymorph method calling is a decision case. Under the "Decision cases" I mean the conditional statements and polymorph method callings of programs. In conformity with this metric, the decisions represented by conditional statements are not analysed from the point of view of whether the use of decision raising and merging is confirmed or not. It could be fulfilled by analysing the behaviour contracts of decisions. Its value range is 0-1, where the higher value indicates the good structure of systems. According to my presumption the increasing rate of polymorph decision cases decreases complexity, and it promotes the increase of maintainability.

"Ratio of inheritances coming into existence by the elimination of decision redundancies": (RIEDR)

In order to measure the fulfilment of the new "Using inheritance to dissolve decision redundancy" design principle, we have to examine whether inheritances are used for the elimination of decision redundancies. I suppose if a parent-subclass inheritance structure contains polymorph methods (the interfaces of decisions are represented by polymorph methods), then the introduction of that inheritance resulted in the elimination of decision redundancies. Accordingly the fulfilment of this principle can be measured based on the analysis of inheritances, whether they contain polymorph methods which are the interfaces of raised decisions. The fulfilment rate of this principle is better when several inheritances of class structure contain polymorph methods. Determining the ratio:

$$\frac{N_{PI}}{N_I}$$

N_{PI} – The number of inheritances containing polymorph methods.
N_I – The number of inheritances.

Its value range is 0-1, where the higher value indicates the good structure of systems. According to my presumption, a higher metric rate value indicates better code integrity and organizing level. In conformity with this, the increase of this ratio decreases the complexity of codes.

The introduction of additional metrics is suggested according to the previously specified decision redundancies and decision merging cases. These metrics can show the rates of the decision redundancies of programs more sophisticatedly. According to the cases of avoiding decision redundancies, the following new metrics are introduced, which can express the fulfilment rate of the "Avoid decision redundancy" design principle at the same time:

The *"Ratio of equivalent decision cases" (REDC)* metric specifies the ratio of the number of equivalent decision cases (the behaviour contracts are fully or partly equivalent according to the decision cases) and the total number of decision cases. The archived decision cases of raised decisions are not considered as equivalent decision cases.

$$\frac{N_{EDC}}{N_{DC}}$$

N_{EDC} – The number of "Equivalent decision cases".
N_{DC} – The total number of "Decision cases".

Its value range is 0-1, where the higher value indicates the wrong structure of systems.

The *"Ratio of decision cases with equivalent decision predicates" (RDCEDP)* metric specifies the ratio of the number of decision cases the decisions of which have equivalent decision predicates and define diverse behaviours and the total number of decision cases. The archived decision cases of raised decisions are not considered as decision cases with equivalent decision predicates.

$$\frac{N_{DCEP}}{N_{DC}}$$

N_{DCEP} – The number of decision cases with equivalent decision predicates and diverse behaviours.
N_{DC} – The total number of "Decision cases".

Its value range is 0-1, where the higher value indicates the wrong structure of systems.

8. The empirical validations of new metrics

We must see that in order to assess the new metrics REDC and the RDCEDP, the analysis of the behaviour contracts of decision options must be taken into consideration. Using the JML – Java behaviour specification language [18], the behavioural contract based examination of decision options and decision structures is possible [24]. In the future, I intend to analyse the behavioural contract based aspects of decision raising and decisions merging by using the JML specifications of decision structures. According to this I intend to realize the empirical validation of the REDC and RDCEDP metrics based on the JML specific examinations of decision structures.

At the same time, the measurement facilities of the MDA and the RIEDR metrics can be automated easier, therefore the empirical validation of them is easier as well. I analysed the sources of several open source projects from "sourceforge.net"[1] empirically in order to justify the relationship between the decision structure based metrics and code integrity. In the course of these measurements the MDA and the RIEDR metrics were evaluated. The scopes and the sizes of the analysed systems were different, which provide good measurement basis. In the following I described the analysed systems shortly:

ProGuard[2]: It is a free class compressing, optimizing, obfuscator and pre-analyser tool, which can search and eliminate non-used classes, member variables, methods and attributes. The range of 10 analysed versions is 3.0–3.9. The number of examined classes is between 317–391, the number of "useful" lines is between 30,573–39,669.

LWJGL[3]: It supports the development of commercial Java-based games. The

[1]`http://sourceforge.net`
[2]`http://proguard.sourceforge.net`
[3]`http://sourceforge.net/projects/java-game-lib`

range of 10 analysed versions is 2.4.2–2.8.5. The number of examined classes is between 254–416, the number of "useful" lines is between 29,292–42,681.

LaTeXDraw[4]: It is a free, Java-based PSTricks code generator and editing tool. The range of 10 analysed versions is 1.5.0–2.0.6. The number of examined classes is between 69–225, the number of "useful" lines is between 28,368–58,483.

Neuroph[5]: It is a freeware, open source neuron network framework, by which neuron network architectures can be developed. The range of 10 analysed versions is 2.1.0–2.8.0. The number of examined classes is between 69–156, the number of "useful" lines is between 2640–6769.

Finding a properly used outer property as a quality indicator is difficult, furthermore, numerous realized measurements confirmed the correlation between the previously specified metrics (complexity, cohesion and coupling metrics) and outer properties (which are based on maintainability and error-proneness) (Subsection 1.4 and Section 2). In conformity with this, I analysed the relationship between the previously specified complexity, cohesion and coupling metrics and the decision structure based metrics which are specified in this paper. Correlations between them were analysed by the Pearson correlation method, by which linear relationships between independent variables can be detected.

In the course of examinations, I took into consideration the complexity, cohesion and coupling metrics, which were introduced by Chidamber and Kemerer (C&K metrics) [8]. Namely, these metrics are the following: "Weighted methods per class" – WMC, "Coupling between objects" – CBO, "Response for a class" – RFC, "Lack of cohesion in methods" – LCOM. The DIT and the NOC metrics (C&K metrics) were not considered according to the arguments which are listed in the following section. I used the CKJM measurement tool [29] in the course of examinations.

In order to measure the MDA and the RIEDR metrics, which were introduced by this paper, a self-made static code analyzer was used, by which the following parameters of programs can be collected:

- The number of inheritances: The number of inheritance relationships between parent and child classes, including interface implementations as well.
- The number of inheritances, where there is at least one polymorph method.
- The number of branches: Branches are the following conditional statements: "if-then-else", "switch", "while", "for".
- The number of polymorph method callings.

Parameters which can be measured by this tool allow the measuring of the MDA and the RIEDR metrics.

[4]`http://latexdraw.sourceforge.net`
[5]`http://neuroph.sourceforge.net`

8.1. C&K metrics descriptions

The metrics which were specified by Chidember and Kemerer in [8] are described as follows, extended with some intuitive reflections:

"Weighted methods per class" – WMC: It measures the complexity of classes. It has two types. According to the first of them, the weight of methods is 1, therefore the number of methods clearly determines the complexity of classes. According to the second case of this metric, the methods are weighted based on their inner complexity [8]. If the inner complexity of methods is not taken into consideration in the course of their evaluation, then this metric may not work properly, because the change of inner method complexity could compensate for the increasing number of methods.

"Depth of Inheritance Tree" – DIT: It is the maximum depth (the case of multiple inheritances) of a class hierarchy, from the examined class to the root parent class [8]. In case of appropriate inheritance usage, a higher DIT value means more complex decision structures, the decisions of which include each other. It also describes problem complexity, the optimizing of which cannot be realized based on the reduction of the levels of class hierarchies, because the complexity of programs is not changeable. But if the introduction of inheritances is not based on the rules of decision merging, then the elimination of non-properly used inheritances may result in the decrease of DIT metric values, according to the appropriate code structure realizations.

"Number of children" – NOC: It is the number of the subclasses of a class. The high number of subclasses increases the probability of non-proper abstractions. Accordingly if a class has lots of subclasses, then it may be the result of non-proper inheritance usage [8]. The metric is not capable of measuring the number of rightly used inheritances. In several cases, the decrease in the number of child classes by introducing new inheritance levels is not confirmed. Based on decision redundancies, it can be found out that there are decisions that can be "linearised" to a level, namely, their merging can be used to lower the number of subclasses.

"Coupling between objects" – CBO: It determines the number of connections between classes. The exaggerated usage of coupling is detrimental to modularity and it decreases re-usability. So the independency of a class increases re-using capability [8]. In the course of the measure of coupling, inheritances are taken into consideration as one type of coupling, which disfigures the measure of dependencies between coupling and re-usage capability. The unsuitable consideration of inheritances as coupling leads to the incorrect conclusion that NOC metric values are high if classes have high CBO metric values [8]. This conclusion is not good, because inheritances do not necessarily spoil the structural quality of coupling. The aim of inheritance is not class reusing, but the extension of classes with a specific behaviour. This approach was introduced by the "Liskov Substitution Principle" (LSP) [12, 19, 27, 28] and by the inheritance cohesion [10]. Inheritance cohesion is strong if inheritances are used to introduce specialized child classes. Respectively it is weak if the main aim of inheritances is reusing.

"Response for a class" – RFC: The response set of a class consists of those methods that can be executed as an effect of the messages sent by the instances of a given class [8].

"Lack of Cohesion in Methods" – LCOM: The interpretation of this metric is based on the dependencies between the methods, which can be determined by the sets of the member variables of classes used by the method. The lack of cohesion may mean that classes should be split into subclasses [8]. The MDA indicates the increase of the number of polymorph method invocations, accordingly the number of raised and merged decisions is growing as well. This results in the decrease of decision separation based behaviour, which causes low cohesion within a class. At the same time, the LCOM metric has more similarities with the REDC and the RDCEDP metrics, which are based on the similarity and the overlapping of behaviour contracts. The concept of these metrics is more similar to the cohesion theoretical basis, by which the functional separation of classes can be expressed. Based on this idea, my future plan is to investigate whether cohesion can be determined by the examination of the similarities between the behaviour contracts of methods.

From the mentioned C&K metrics DIT and NOC metrics are strongly related to the complexity of inheritance hierarchies. At the same time, these two metrics do not clarify the cases where inheritances can be used rightfully. So it is possible that complex structures signed by DIT and NOC only indicate the complexity of the realized problem, which can be optimal from the point of view of the code structure. Therefore I do not analyse the aspects of these metrics.

8.2. The measuring results of the MDA

For the measurement of the MDA the following ratio must be determined:

$$\frac{N_{PMI}}{N_{CS} + N_{PMI}}$$

N_{PMI} – The number of polymorph method invocations.
N_{CS} – The number of conditional statements.

This ratio can be determined using the results of the developed measuring tool. According to my supposition, N_{PMI} approximately determines N_{PDC}, and N_{DC} can be determined by summing up N_{CS} and N_{PMI}.

I analysed the versions of the previously described ProGuard, LWJGL, LaTeX-Draw, Neuroph projects. The correlations between the MDA and the WMC, CBO,

	WMC	**CBO**	**RFC**	**LCOM**
ProGuard	**−0.893**	**−0.881**	**−0.905**	**−0.773**
LWJGL	**−0.907**	0.821	**−0.678**	**−0.928**
LaTeXDraw	**−0.648**	0.508	**−0.623**	**−0.684**
Neuroph	−0.433	0.067	−0.130	**−0.685**

Table 1: The Pearson product-moment correlation coefficients between the MDA and the WMC, CBO, RFC, LCOM metrics

RFC, LCOM metrics [8] were examined based on the Pearson product-moment correlation coefficient (see Table 1). According to the measurement of the three systems, the correlation between the MDA and the WMC metric is significant, however, there is one system (Neuroph), where the correlation is low, but exists. The correlation measurements between the MDA and the CBO metric are ambiguous, therefore there is no correlation between them. Based on the measurements of three systems, the correlation between the MDA and the RFC metric is significant, however, there is one correlation measurement which indicates no correlation between them (Neuroph system). The measurements confirmed the significant relationship between the MDA and the LCOM metric. In conformity with the measurements, the correlation between these metrics is the most significant.

8.3. The measuring results of the RIEDR metric

For the measurement of the RIEDR metric the following ratio must be determined:

$$\frac{N_{PI}}{N_I}$$

N_{PI} – The number of inheritances containing polymorph methods.
N_I – The number of inheritances.

I analysed the versions of the previously described ProGuard, LWJGL, LaTeX-Draw, Neuroph projects. The correlations between the RIEDR metric and the WMC, CBO, RFC, LCOM metrics [8] were examined based on the Pearson product-moment correlation coefficient (see Table 2).

	WMC	**CBO**	**RFC**	**LCOM**
ProGuard	**−0.830**	**−0.863**	**−0.798**	**−0.925**
LWJGL	**−0.866**	0.739	−0.255	**−0.772**
LaTeXDraw	0.737	0.446	0.745	0.681
Neuroph	−0.278	0.481	0.330	**−0.861**

Table 2: The Pearson product-moment correlation coefficients between the RIEDR and the WMC, CBO, RFC, LCOM metrics

The correlation measurements between the RIEDR metric and the WMC, CBO, RFC metrics are ambiguous. Accordingly, there is no correlation between these metrics. In conformity with the measurements of three systems, the correlation between the RIEDR metric and the LCOM is significant, but there is one system (LaTeXDraw), where the measurement indicates inverse correlation. In order to determine the reason of the inverse correlation in case of the LaTeXDraw system further examinations are needed.

8.4. The summary of empirical validations

In case of the WMC metric [8] the optionally considered inner complexity of methods promote the correlation with the MDA, because inner complexity is related to the quality of decision structures which can be measured by the new metrics.

The relationship between the LCOM metric [8] and the new metrics can be perceived based on behaviour contracts [18, 27], which should be considered in the course of the determination of cohesion. These behaviour contracts specify the basic concepts of decision merging examinations and the introduction of new decision structure quality specific metrics. These supposed relationships were confirmed by empirical validations.

The empirically perceived relationship between the RFC metric [8] and the MDA was not supposed intuitively. To find the cause of the empirical connection between the two metrics requires further examinations.

9. Conclusions

In the course of the paper the definitions of decision, decision raising [24, 25] and the newly introduced decision merging are extended based on the concept of behavioural contract [18, 27].

Using the behaviour contract-based definitions, the behavioural contract specific aspects of the transformations of decision raisings and decision merging can be showed. Using the JML – Java behaviour specification language [18], the behavioural contract-based examination of decision structures is possible [24]. In the future, I intend to analyse the behavioural contract-based aspects of decision raising and decisions merging by using the JML specifications of decision structures.

Based on the described concepts of decision redundancies and the rules of decision merging, I introduced new object-oriented design principles (“Using inheritance to dissolve decision redundancy”, “Avoid decision redundancy”). These principles determine the cases, where the use of inheritance as an object-oriented tool is justified. Several existing object-oriented design principles are engaged in detecting the cases where the use of inheritance vs. object composition is confirmed. I intend to examine the relationship between the existing design principles and the quality of decision structures.

I will deal with the examination of the designing circumstances of design patterns. In the course of the examination of the decision structures of design patterns, I plan to examine high-level optimizing facilities and low-level refactoring methods. One of the new directions could be the examination of the suspected relationship between decision structures and design patterns. According to this relationship the decision structure circumstances of design patterns appear in Use Case models [26]. In conformity with this, a new research direction is to find out how the design patterns appear in Use Case models, or rather, how the decision structures of design patterns reflect on the level of Use Cases.

I initiated new object-oriented metrics that give the opportunity to examine the quality of decision structures. The introduced MDA and RIEDR metrics are examined empirically compared to the previously specified complexity, cohesion and coupling metrics. The correlations between them are analysed by the Pearson correlation method, by which the linear relationship between independent variables can be analysed. According to the measurements, the correlations between the MDA and the WMC, RFC, LCOM [8] are significant, furthermore, there is a significant correlation between the RIEDR metric and the LCOM [8] metric as well. In the cases of the WMC and LCOM metrics [8], the detected relationship can be perceived intuitively, but the empirically confirmed relationship between the RFC metric [8] and the MDA requires additional examinations. The relationship between the LCOM [8] and the decision structure based metrics is based on the dependencies between the functional separation signing capability of cohesion and decision structure anomalies.

References

[1] L. Badri, M. Badri, and B. Gueye. Revisiting class cohesion: An empirical investigation on several system. *Journal of Object Technology*, 7(6):55–75, 2008.

[2] M. Badri, L. Badri, and F. Touré. Empirical analysis of object-oriented design metrics: Towards a new metric using control flow paths and probabilities. *Journal of Object Technology*, 8(6):123–142, 2009.

[3] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. *In Proceedings of the ACM Symposium on Software Reusability (SSR'95)*, pages 259–262, 1995.

[4] G. Booch, R.A. Maksimchuk, M.W. Engel, B.J. Young, J. Conallen, and K.A. Houston. *Object-Oriented Analysis and Design with Applications.* Addison Wesley Longman Publishing Co., Inc., 3rd edition, 2007.

[5] L.C. Briand, J.W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[6] H.S. Chae and Y.R. Kwon. A cohesion measure for classes in object-oriented systems. *In Proceedings of the 5th. International Software Metrics Symposium. Bethesda, MD*, pages 158–166, 1998.

[7] S.R. Chidamber and C.F. Kemerer. Towards a metrics suite for object oriented design. *In Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, pages 197–211, 1991.

[8] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] T. DeMarco. *Structured analysis and system specification.* Yourdon Press, Prentice Hall, Inc., 1979.

[10] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. *Technical Report, University of Klagenfurt, Austria*, pages 1–34, 1994.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the design of existing code.* Addison Wesley Longman Publishing Co., Inc., 1999.

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

[13] B. Henderson-Sellers. *Object-Oriented Metrics. Measures of Complexity.* Prentice Hall, Inc., 1996.

[14] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. *In Proceedings of the International Symposium on Applied Corporate Computing, Monterrey, Mexico*, 50:75–76, 1995.

[15] H. Kabaili, R.K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. *In Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), IEEE, Lisbon, Portugal*, pages 39–46, 2001.

[16] J. Kerievsky. *Refactoring to patterns.* Addison Wesley Longman Publishing Co., Inc., 2004.

[17] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall, Inc., 3rd edition, 2005.

[18] G.T. Leavens and Y. Cheon. Design by contract with JML. *Dept. of Computer Science, Iowa State University, Dept. of Computer Science, University of Texas at El Paso*, pages 1–13, 2006.

[19] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[20] R.C. Martin and M. Micah. *Agile principles, patterns, and practices in C#.* Prentice Hall, Inc., 2006.

[21] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[22] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, 1992.

[23] J.C. Munson and T.M. Khoshgoftaar. *Handbook of Software Reliability Engineering. Chapter 12.: Software Metrics for Reliability Assessment.* IEEE Computer Society Press, McGraw-Hill, 1996.

[24] Sz. Márien. Decision based examination of object-oriented methodology using JML. *Annales Mathematicae et Informaticae*, 35:95–121, 2008.

[25] Sz. Márien. Decision based examination of object-oriented programming and design patterns. *Teaching Mathematics and Computer Science*, 6(1):83–109, 2008.

[26] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual.* Addison Wesley Longman Publishing Co., Inc., 2nd edition, 2004.

[27] W. Schreiner. From types to contracts: Supporting by light-weight specifications the liskov substitution principle. *Technical Report no. 10-22 in RISC Report Series. Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria*, 2010.

[28] R.W. Sebesta. *Concepts of Programming Languages.* Addison Wesley Longman Publishing Co., Inc., 7th edition, 2006.

[29] D. Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, 2005.

[30] Y. Wand and R. Weber. An ontological model of an information system. *IEEE Transactions on Software Engineering*, 16(11):1282–1292, 1990.

[31] S. Yacoub, T. Robinson, and H.H. Ammar. Dynamic metrics for object oriented designs. *In Proceedings of the 6th International Software Metrics Symposium*, pages 50–61, 1999.