



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Manually Mapping Model Elements onto the Modeled Code by Analyzing GitHub Data

Master's thesis in Computer science and engineering

WENLI ZHANG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

MASTER'S THESIS 2023

Manually Mapping Model Elements onto the Modeled Code by Analyzing GitHub Data

WENLI ZHANG



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Manually Mapping Model Elements onto the Modeled Code by Analyzing GitHub
Data
WENLI ZHANG

© WENLI ZHANG, 2023.

Supervisor: Regina Hebig, Computer Science and Engineering
Examiner: Christian Berger, Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2023

WENLI ZHANG

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Context: Class diagrams are one of the most popular UML models and are frequently used in the early stages of software development. The advantage of using class diagrams is that they can reflect design decisions and the system’s implementation structure. Maintainers can use class diagrams to understand the system’s implementation structure. Yet, as the code evolves, the absence of updating class diagrams will cause the code implementation to deviate from the class diagram design. One concern is that such a divergent class diagram does not help maintainers much in the same way during the maintenance stage. As a solution, reverse engineering methods/tools can reverse code into class diagrams. Yet, another concern comes up, in most cases, the reverse-engineered class diagrams are not abstract, and they contain extensive information that will burden the understanding of the system’s implementation structure. This is because the existing reverse engineering methods/tools are imperfect as they do not manage to imitate the human ability to abstract relevant information from the source code. Surprisingly, existing studies on the characteristics of manual abstraction are based on the opinions and experiences of participants but do not study actual cases of models and source code. Also, the methods/technologies used for checking the similarities and differences between the models and source code are purely structural but do not analyze or take the semantics of the model elements into account when mapping classes from models and code. The semantics is closely related to abstraction creation. Thereby, a systematic manual study on the characteristics of manual abstraction is required.

Aim: To fill this gap, this thesis aimed at studying the characteristics of the differences between the class diagram design and the code implementation by manually creating the mappings between the class diagram elements/constituents and the code constructs. Our manual studies can precisely capture the differences between the class diagram design and source code implementation and investigate the causes of these differences.

Method: We employed the methodology of five case studies. The five subjects studied are five Java open-source projects collected from GitHub. They are semi-randomly selected from the Linholmen dataset [1].

Results: For the differences between the class diagram design and code implementation, three causes are summarized: various levels of manual abstraction created in class diagrams, deviations of code implementation from class diagram design,

and common changes between the class diagram elements/constituents and code constructs. We contribute to a sorted list of cases corresponding to these three causes.

Keywords: UML, Models in Open Source Systems, Reverse Engineering, Deviations between Code and Design, Manual Abstraction in Modeling

Acknowledgements

Approaching the end of my master’s degree, I would take this opportunity to express my gratitude to all my lecturers, former colleagues, friends, and family who helped and supported me through these two and a half years.

My thesis supervisor, Dr. Regina Hebig, for your all-around support and trust in my thesis work has been like the warmest sunshine in a dark Swedish winter. Your knowledge, inner drive, and kindness in helping me all around (not limited to academics) fully affected my attitude toward my future studies and life - being loyal and optimistic. Of particularly unforgettable days when I fell into the definition of terminologies, you guided me patiently, even hand in hand, taught and inspired me to find the best solution to accommodate the puzzle I was in at that time. Your valuable and interesting learning experience gained when you were a former student in the past, shared with me, will be an invaluable asset in my life.

My examiner, Dr. Christian Berger, for your valuable input in my thesis proposal and half-time presentation, helping me think of the methodology of this thesis more considerably. Dr. Richard Torkar, for whom I served as a teaching assistant for about six months. Your enthusiasm for research and willingness to help students out has been encouraging me to want to be a person the way you are. To all of my lecturers for instilling me with software engineering-related knowledge and cultivating my problem-solving and independent thinking abilities.

To my former company mentor, Dr. Jianlin Shi, and my director, Yujie Xue, with whom I worked for half a year in 2021. Your full support and care greatly boosted my confidence to pursue what I am into (this thesis work) at the campus.

To my friends, who cheered me up and always believed in me, you are all like true “family members”. We shared our interests and hobbies, patiently taught each other some life tips, and explored new things together. This constructs a major part of my life towards a balance of learning and living. To the most important persons coming into my life - my parents, brother, and sister, who have always influenced me to be brave and supported me to pursue what I have been interested in since I was a child.

My great interest in modeling and software architecture supported me in finalizing this thesis. I am grateful to Dr. Regina Hebig again for providing me with this thesis topic, which helped me to step into my area of interest and establish a framework for learning which can be further extended in the future.

Without all of you, this thesis would not have been possible.

Wenli Zhang, Gothenburg, February 2023

Contents

List of Figures	xvii
List of Tables	xxiii
List of Acronyms	xxiv
1 Introduction	1
1.1 Background	1
1.2 Challenges	1
1.3 Goal and Motivation	3
1.4 Case Study Subjects	3
1.5 Definition of Terminologies	4
1.5.1 Examples of Mappings between mcAM Concepts and voSC . .	5
1.5.2 Ideal Selection of One cSC among Multiple cSCs of a mcAM .	9
1.6 Research Questions	11
1.7 Contributions	12
1.8 Structure of the Paper	13
2 Theory and Related Work	15
2.1 Theory	15
2.1.1 Related Java Knowledge	15
2.1.1.1 Java SE7 Specifications	16
2.1.1.2 Object-Oriented (OO) Paradigm	16
2.1.2 Unified Modeling Language (UML)	17
2.1.2.1 Graphical Notation for Classifiers	17
2.1.2.2 Conversions of BNF	18
2.1.2.3 Textual Notation for Attributes	19
2.1.2.4 Textual Notation for Operations	22
2.1.2.5 Visibility	25
2.1.2.6 Multiplicity	26
2.1.2.7 Graphical Notation for Relationships	27
2.2 Related Work	29
2.2.1 Reverse Engineering	29
2.2.1.1 Manual Abstraction Created over Code	29
2.2.1.2 Solutions and Attempts to Provide Abstraction for Reverse-Engineered Class Diagrams	30
2.2.1.3 Consistency Check(s) between Code and Design . . .	31

2.2.2	Models in Software Development Practice	32
2.2.2.1	Usage of Models	32
2.2.2.2	Dataset/Database of Models	33
3	Methodology	35
3.1	Open Source Repositories Access	36
3.2	Data Selection	36
3.2.1	Definition of Terminologies	37
3.2.2	Ideal Selection of One mcAM from a Project	37
3.2.3	Ideal Selection of One cSC among Multiple cSCs of that mcAM	38
3.2.3.1	Time and Resource Consumption	40
3.3	Automatic Reverse Engineering Tool Selection	41
3.4	Spreadsheet Comparison Template Design	41
3.4.1	Turning mcAM and cSC into Pictures for Differentiating . . .	45
4	Results	49
4.1	Information on the Project Studied	49
4.2	Suspected Cases	49
4.3	Cases of the Differences Caused by MA and disAGTs	51
4.3.1	Cases of MA - Classes	53
4.3.1.1	Case MC-1 - Hierarchical inheritance structure not created in the mcAM is added to the cSC	53
4.3.1.2	Case MC-2 - One class created in the mcAM is di- vided into more than one class in the cSC (related to the specific design patterns)	53
4.3.2	Cases of MA - Attributes	55
4.3.2.1	Case MA-1 - Additional attributes not in the mcAM are added to the cSC	55
4.3.2.2	Case MA-2* - An attribute of classifier A whose type can indicate an aggregation or a composition be- tween classifiers A and B from the cSC, is modeled out by the naming of the association between them in the mcAM	55
4.3.2.3	Case MA-3* - The additional attribute type not in the mcAM is added to the cSC	56
4.3.2.4	Case MA-4 - The additional default value not in the mcAM is added to the cSC	56
4.3.2.5	Case MA-5 - One or more common attributes in one or more subclasses in the mcAM are upshifted to the hierarchical structure in the cSC, i.e., the superclass inherited by these subclasses	57
4.3.2.6	Case MA-6 - One attribute created from the mcAM is divided into more than one attribute in the cSC . .	58
4.3.3	Cases of MA - Operations	59
4.3.3.1	Case MO-1 - Additional operations not in the mcAM are added to the cSC (three subcases)	59

4.3.3.2	Case MO-2 - Additional parameter names not in the mcAM are added to the cSC	61
4.3.3.3	Case MO-3 - Additional parameter types not in the mcAM are added to the cSC	62
4.3.3.4	Case MO-4 - Additional return types not in the mcAM are added to the cSC	62
4.3.3.5	Case MO-5* - Multiplicity (referring to the collection-related interfaces) specified for the return type without its corresponding implementation interfaces/classes specified in the mcAM, yet with them specified in the cSC	62
4.3.3.6	Case MO-6* - The default parameters of the operation (in Android) in the cSC are omitted in the mcAM	63
4.3.3.7	Case MO-7 - One operation created in the mcAM is divided into multiple operations in the cSC	64
4.3.4	Cases of MA - Relationships	64
4.3.4.1	Case MR-1 - Additional relationships not in the mcAM are added to the cSC (two subcases with their six and two corresponding concluded causes, respectively) . .	67
4.3.4.2	Case MR-2 - An aggregation between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (two causes)	79
4.3.4.3	Case MR-3 - A composition between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (one cause)	82
4.3.5	Cases of disAGTs - Classes	84
4.3.5.1	Case DC-1 - Hierarchical inheritance structures in the mcAM are removed in the cSC	84
4.3.6	Cases of disAGTs - Attributes	85
4.3.6.1	Case DA-1 - Attributes in the mcAM are removed in the cSC	85
4.3.6.2	Case DA-2* - Attributes from the mcAM are replaced with additional attributes (that are not in the mcAM added to the cSC) in the cSC	86
4.3.6.3	Case DA-3* - The attribute type in the mcAM and cSC is different (two causes)	86
4.3.6.4	Case DA-4* - Converting variables from the mcAM to constant variables in the cSC	87
4.3.6.5	Case DA-5* - The default value not in the mcAM is added to the cSC	88
4.3.7	Cases of disAGTs - Operations	89
4.3.7.1	Case DO-1 - Operations in the mcAM are removed in the cSC	89

4.3.7.2	Case DO-2 - The parameter names in the mcAM are removed in the cSC	89
4.3.7.3	Case DO-3 - The parameter types in the mcAM are removed in the cSC	89
4.3.7.4	Case DO-4* - The parameter type in the mcAM and cSC is different (one cause)	89
4.3.7.5	Case DO-5 - The return types in the mcAM are removed and as void in the cSC	90
4.3.7.6	Case DO-6* - One or more operations from classifier A in the mcAM are moved to classifier B in the cSC (related to the particular architectural patterns) . . .	91
4.3.8	Cases of disAGTs - Relationships	92
4.3.8.1	Case DR-1 - Relationships between A and B from the mcAM are removed in the cSC (one cause) . . .	93
4.3.8.2	Case DR-2 - A composition between A (whole) and B (part) from the mcAM is changed into an aggregation in the cSC (one cause)	94
4.3.8.3	Case DR-3* - Relationships between A and B from the mcAM are replaced with new relationships in the cSC (two causes)	95
4.4	Cases of the Differences Caused by CC	98
4.4.1	Cases of CC - Classes	100
4.4.1.1	Case CC-1 - Naming of classes in the mcAM and cSC is different (three causes)	100
4.4.2	Cases of CC - Attributes	101
4.4.2.1	Case CA-1 - Naming of attributes in the mcAM and cSC is different (two causes)	101
4.4.2.2	Case CA-2 - Naming of attribute types is different (two causes)	102
4.4.3	Cases of CC - Operations	103
4.4.3.1	Case CO-1 - Naming of operations in the mcAM and cSC is different (four causes)	103
4.4.3.2	Case CO-2 - Naming of parameters in the mcAM and cSC is different (two causes)	104
4.4.3.3	Case CO-3 - Naming of parameter types in the mcAM and cSC is different (one cause)	105
4.4.3.4	Case CO-4 - Naming of return types in the mcAM and cSC is different (one cause)	105
4.4.4	Ratios of Cases with Related Projects	106
4.4.5	Typical/Common Cases with Related Projects	107
4.4.6	Aggregated Results of the Cases for the Project	108
5	Discussion	111
5.1	Reflection on Data Selection	111
5.2	Limitations of Reverse Engineering Tools	112
5.3	Advantages of Manual Mappings	113

5.4	Guideline Enlightened by Cases	113
5.5	Related Technologies for Healing the Cases	115
5.6	Threats to Validity	116
5.6.1	Threats to Construction Validity	117
5.6.2	Threats to External Validity	117
5.6.3	Threats to Conclusion Validity	118
6	Conclusion and Future Work	119
6.1	Conclusion	119
6.2	Future Work	120
6.2.1	Covering Additional OOP Projects	120
6.2.2	Quantitative Analysis	120
	Bibliography	121
A	Corresponding mcAMs in the Five Projects Studied	I

List of Figures

1.1	Three challenges involved in the staged SDLC are likely to cause the implementation of the source code to deviate from the design of the class diagram; thereby, differences between the class diagram and the source code are introduced.	2
1.2	In the mcAM, for the concept, e.g., <i>Dog</i> , there is a second concept <i>Pet</i> is described by the superclass <i>Pet</i> of the class <i>Dog</i> that describes the concept <i>Dog</i>	5
1.3	For the superclass <i>Pet</i> of the class <i>Dog</i> that describes a second concept <i>Pet</i> of a concept <i>Dog</i> in the mcAM, the class <i>Pet</i> in this voSC is considered to be mapped to that superclass <i>Pet</i> of the class <i>Dog</i> in the mcAM.	5
1.4	A class <i>SinglePlayModel</i> describes a concept <i>SinglePlayerModel</i> in the mcAM, yet the naming of the class <i>SinglePlayModel</i> with a misspelling.	6
1.5	The class <i>SinglePlayerModel</i> in this voSC has highly similar attributes and operations with the class <i>SinglePlayModel</i> in the mcAM and thus, the class <i>SinglePlayModel</i> is considered to be mapped to the class <i>SinglePlayerModel</i> in this voSC.	6
1.6	In the mcAM, a concept, i.e., <i>decorators</i> is described by a class <i>ItemDecorator</i> , which might imply the decorator design pattern would be applied in the voSC.	7
1.7	In this voSC, the concept <i>decorators</i> in the mcAM is described by the superclass <i>Slot</i> , and the subclasses <i>LeftUpgradeSlot</i> , and <i>RightUpgradeSlot</i> derived from the superclass <i>Slot</i> . These three classes are the extension of the interface <i>Icon</i> embedded in the Java library. This interface <i>Icon</i> is invoked in a class related to View, which is responsible for communicating with the subclass <i>Upgrade</i> derived from the superclass <i>Item</i> in the mcAM.	7
1.8	In the mcAM, for the concept, e.g., <i>Dog</i> a second concept <i>Pet</i> is described by the superclass <i>Pet</i> of the class <i>Dog</i> that describes the concept <i>Dog</i>	8
1.9	This voSC only includes one class <i>Pet</i> that can be mapped to the superclass <i>Pet</i> of the class, e.g., <i>Dog</i> in the mcAM.	8
1.10	A concept, i.e., <i>InputFunction</i> is described by a class <i>InputFunction</i> in the mcAM.	9
1.11	The concept <i>InputFunction</i> in the mcAM is described by a superclass <i>InputFunction</i> that not in the mcAM but added to the voSC.	9

1.12	In the mcAM, eight concepts are described by eight classes, i.e., <i>BaseUI</i> , <i>BaseController</i> , <i>IncomeRegisterUI</i> , <i>RegisterIncomeController</i> , <i>Income</i> , <i>IncomeTyperRepository</i> , <i>CheckingAccount</i> , and <i>IncomeRepository</i> , respectively.	10
1.13	A cSC of the mcAM covers all eight concepts in the mcAM.	11
1.14	Compared with the cSC illustrated in Figure 1.13, this cSC of the mcAM covers four more operations.	11
2.1	Three abstraction levels of graphical class notation: suppressed (on the top left corner), analysis (on the right), and implementation (on the bottom left corner) [21, p. 50]. Compared with the suppressed level with only the specified name of a class, another two levels present a more comprehensive layout of a class with more or less) textual notation for attributes and operations embedded.	18
2.2	Specification of Backus-Naur Form (BNF) conversions [21, pp. 16–17].	19
2.3	Textual notation for attributes [21, pp. 129–130].	19
2.4	Textual notation for operations [21, pp. 107–108].	22
2.5	Syntax for a multiplicity string [21, p. 98].	26
2.6	Graphical notation for seven types of relationships.	28
2.7	Four abstraction levels specified for four relationships - <i>dependency</i> , <i>association</i> , <i>aggregation</i> , and <i>composition</i> , respectively.	29
3.1	The overall process of the methodology.	35
3.2	The selection process of one ideal cSC among multiple cSCs of the selected mcAM.	39
3.3	Record information about the project.	42
3.4	Record information of the mcAM.	43
3.5	Record information of the cSC.	44
3.6	Record the differences between the mcAM and the cSC of that mcAM.	44
3.7	Record additional notes.	45
3.8	For the class <i>CheckingAccount</i> , the attributes, e.g., <i>income: Income</i> is modeled in the mcAM.	47
3.9	For the class <i>CheckingAccount</i> , that attribute <i>income: Income</i> from the mcAM is replaced with fully new attributes <i>incomeRepo: IncomeRepository</i> in the cSC.	47
3.10	We recorded the difference, i.e., attributes from the mcAM are replaced with new attributes not in the mcAM yet added to the cSC, differentiated into disAGTs in our designed comparison template. . .	47
4.1	The subclasses, e.g., <i>Max</i> implemented in the cSC are hidden in the mcAM. Also, the superclass <i>InputFunction</i> inherited by these subclasses in the cSC is modeled as a class in the mcAM.	53
4.2	A hierarchical inheritance structure with the corresponding additional two subclasses not in the mcAM is added to the cSC.	53
4.3	In the mcAM, a concept, i.e., <i>decorators</i> is described by a class <i>ItemDecorator</i> , which might imply the decorator design pattern will be applied in the cSC.	54

4.4	In the cSC, the concept <i>decorators</i> in the mcAM is described by the superclass <i>Slot</i> , and the subclasses <i>LeftUpgradeSlot</i> , and <i>RightUpgradeSlot</i> derived from the superclass <i>Slot</i> . These three classes are the extension of the interface <i>Icon</i> embedded in the Java library. This interface <i>Icon</i> is invoked within a <i>View</i> class, which is responsible for communicating with the subclass <i>Upgrade</i> derived from the superclass <i>Item</i> in the mcAM.	54
4.5	The naming of the association between the classes <i>LaborBilling</i> and <i>PhaseLabor</i> is specified with the attribute name <i>lb</i> in the cSC.	56
4.6	In the cSC, the attribute name <i>lb</i> indeed exists in the class <i>PhaseLabor</i>	56
4.7	In the mcAM, for the attributes, e.g., <i>email</i> , a default value is not assigned.	57
4.8	A default value “ ” not in the mcAM is added to the attribute <i>email</i>	57
4.9	In the mcAM, for the subclasses <i>Food</i> and <i>Upgrade</i> , there is a common attribute <i>image</i>	58
4.10	The common attribute <i>image</i> of the subclasses <i>Food</i> and <i>Upgrade</i> in the mcAM is upshifted to their superclass <i>Item</i> in the cSC.	58
4.11	Three subcases correspond to this case with their corresponding causes.	61
4.12	Case MR-1 with the corresponding subcases and causes.	65
4.13	Cases MR-2 and MR-3, with the corresponding causes.	66
4.14	In the mcAM, no association is modeled between the classes <i>LaborBilling</i> and <i>Pattern</i>	68
4.15	In the cSC, an operation <i>getId()</i> of <i>LaborBilling</i> is invoked in an operation <i>getPhaseLabor(LaborBilling, Phase): PhaseLabor</i> of <i>Pattern</i> . Yet, the parameter types of the latter operation, e.g., <i>LaborBilling</i> , are not modeled out in the mcAM (as shown in Figure 4.14).	68
4.16	In the mcAM, no relationship is modeled between the classes <i>BaseController</i> and <i>CheckingAccount</i> . Plus, the return type <i>CheckingAccount</i> modeled in <i>BaseController</i> in the mcAM can partially imply an association to exist in the cSC.	69
4.17	In the cSC, an instance of <i>CheckingAccount</i> is created within the operation <i>buildCheckingAccount(): CheckingAccount</i> of <i>BaseController</i> , further being returned to this operation.	69
4.18	In the mcAM, no relationship is modeled between the classes <i>ItemVisitor</i> and <i>Food</i> . Yet, an instance of <i>Food</i> , i.e., is specified as a parameter type of the operation named <i>visit</i> of <i>ItemVisitor</i> in the mcAM.	70
4.19	In the cSC, for <i>Food</i> , its operation <i>getPrice(): int</i> inherited from the superclass <i>Item</i> is indeed invoked within <i>visit(Food): void</i> of <i>ItemVisitor</i> in the cSC.	71
4.20	In the mcAM, no relationship is modeled between the classes <i>SinglePlayer</i> and <i>TitlePage</i>	72
4.21	In the cSC, an instance of <i>TitlePage</i> is created within an operation <i>pausedMainMenu(View view): void</i> of <i>SinglePlayer</i>	72
4.22	In the mcAM, no relationship is modeled between the classes <i>UserBuilder</i> and <i>Control</i>	73

- 4.23 In the cSC, an instance named *userBuilder* of *UserBuilder* is created within the operation *newUser(int, String, String, String)* of *Control*. Plus, an operation of *userBuilder*, e.g., *setEmail(): String*, is further invoked with the same operation of *Control* (in the mcAM)/*RaiseMeUp (in the cSC)*. 73
- 4.24 In the mcAM, no relationship is modeled between the classes *Food* and *Control*. 74
- 4.25 In the cSC, an instance of *Food* (named *f*) is specified as a parameter type of an additional operation *removeFood(Food): boolean* of *Control/RaiseMeUp*. This instance of *Food* is used in another operation *delFood(f): boolean* of another class *Dao*. Plus, in *Dao*, an operation of *Food*, e.g., *getName()*, is further invoked with that *delFood(Food): boolean*. 75
- 4.26 In the mcAM, no relationship is modeled between the classes *Neuron* and *NeuralNetwork*. 76
- 4.27 In the cSC, an additional attribute *inputNeurons: NeuorphArrayList<Neuron>* not in the mcAM is added to the class *NeuralNetwork*. The operation *size()* of *Neuron* is further invoked within the operation *getInputsCount(): int* of *NeuralNetwork*. Plus, the instances of *Neuron* are contained by not only *NeuralNetwork* but also by *Layer* in the cSC. 77
- 4.28 In the mcAM, no relationship is modeled between the classes *Job* and *Dao*. 78
- 4.29 In the cSC, an additional attribute *jobs: Map<Integer, Food>* not in the mcAM is added to the class *DAO* and it is further created in *Dao* body. An operation *put()* (an API) is further invoked in the operation *getJob(): Map<Integer, Job>* of *Dao* with the instance *jobs*. However, the instances of *Job* are contained by not only *DAO* but also by another class *Pet* in the cSC. 78
- 4.30 The aggregation between *Employee* and *Schedule* from the cSC is modeled as an association in the cSC. Plus, the naming of this association in the mcAM is specified by an attribute name (whose related type is specified by the instances of *Schedule*) of *Employee* from the cSC. 80
- 4.31 In the cSC, an attribute type *ArrayList<Schedule>* contained in the class *Employee* means a group of instances of the class *Schedule* declared in *Employee*. These instances of *Schedule* are further created within the operation *Employee(Integer, String, String)* of *Employee* in the cSC. However, the instances of the class *Schedule* are contained not only by the instances of *Employee* but also by the instances of another class *MainClass* in the cSC. 80
- 4.32 An aggregation between *Income* and *IncomeRepository* from the cSC is partially indicated by the attribute type (instances of *Income*) of *IncomeRepository*, i.e., *List<Income>*, in the mcAM. 82

4.33	In the cSC, the type of the attribute named <i>listIncome</i> is specified by a collection of instances of <i>Income</i> , i.e., <i>ArrayList<Income></i> in <i>IncomeRepository</i> . This collection of instances of <i>Income</i> is further created within an operation <i>IncomeRepository()</i> of <i>IncomeRepository</i> . Plus, the instances of <i>Income</i> are invoked within an operation <i>save(Income): void</i> of <i>IncomeRepository</i> in the cSC. Yet, the instances of <i>Income</i> are contained by not only <i>IncomeRepository</i> but also another class <i>RegisterIncomeController</i> in the cSC.	82
4.34	A composition between the classes <i>SinglePlayer</i> and <i>SinglePlayModel</i> from the cSC is modeled as an association in the mcAM.	84
4.35	In the cSC, an instance <i>model</i> of the class <i>SinglePlayerModel</i> that is modeled out as an attribute type in the class <i>SinglePlayer</i> in the mcAM is indeed declared as an attribute type of the class <i>SinglePlayer</i> in the cSC. This created instance <i>model</i> of <i>SinglePlayerModel</i> is further invoked within an operation of <i>SinglePlayerModel</i> , e.g., <i>onKeyDown(int, KeyEvent): boolean</i> . Plus, in the cSC, this instance <i>model</i> is exclusive to the corresponding instances of <i>SinglePlayer</i> . . .	84
4.36	A hierarchical inheritance structure in the mcAM.	85
4.37	The hierarchical inheritance structure in the mcAM is removed in the cSC. Solely the superclass <i>Pet</i> in the mcAM is remained yet changed into a class in the cSC.	85
4.38	For the class <i>CheckingAccount</i> , the attributes <i>income: Income</i> and <i>amount: BigDecimal</i> are modeled in the mcAM.	86
4.39	For the class <i>CheckingAccount</i> , all its attributes in the mcAM are replaced with fully new attributes <i>incomeRepo: IncomeRepository</i> and <i>expenseRepo: ExpenseRepository</i> in the cSC.	86
4.40	Two causes for case DA-3*.	87
4.41	In the mcAM, for the attribute named <i>owner</i> , <i>User</i> (a non-primitive data type) is specified. Also, for its setter operation named <i>setOwner()</i> , a parameter type <i>User</i> is specified.	90
4.42	In the cSC, for that attribute named <i>owner</i> its specified type <i>User</i> in the mcAM changed to <i>int</i> instead. Accordingly, for the setter operation named <i>setOwner()</i> of that attribute, its parameter type changed from <i>User</i> to <i>int</i>	90
4.43	In the mcAM, for the class <i>DAO</i> , the operations, e.g., <i>listUser(): Map</i> is created.	92
4.44	In the cSC, the operation, e.g., <i>listUser(): Map</i> created in the class <i>DAO</i> from the mcAM is moved to another class <i>RaiseMeUp/Controll</i> as <i>listUsers(): Map<Integer, User></i> . Then it is used for getting a list of user data from the Model-related class <i>Dao</i> in the cSC.	92
4.45	Cases DR-1, DR-2, and DR-3*, with the corresponding causes.	93
4.46	In the mcAM, a composition is created between the classes <i>Pet</i> and <i>User</i>	94

4.47	In the cSC, the attribute type whose type is specified by an instance of <i>User</i> from the mcAM is converted to a primitive data type <i>int</i> . This leads to the corresponding composition between <i>Pet</i> and <i>User</i> from the mcAM being removed in the cSC.	94
4.48	In the mcAM, a composition is modeled between the classes <i>Layer</i> and <i>Neuron</i>	95
4.49	In the cSC, the instances of <i>Neuron</i> are contained by not only <i>Layer</i> but also <i>NeuralNetWork</i>	95
4.50	In the mcAM, an association is created between <i>CheckingAccount</i> and <i>IncomeRepository</i>	96
4.51	For <i>CheckingAccount</i> , the attributes from the mcAM, e.g., <i>income: Income</i> , are replaced with fully new attributes, e.g., <i>incomeRepo: IncomeRepository</i> in the cSC. The specified attribute type, an instance of <i>IncomeRepository</i> , is created within an operation <i>CheckingAccount()</i> of <i>CheckingAccount</i> in the cSC. This instance is further invoked within another operation <i>add(Income): void</i> of <i>CheckingAccount</i> in the cSC. Furthermore, the instances of <i>IncomeRepository</i> are contained not only by <i>CheckingAccount</i> but also by another class <i>ValuesCalculator</i> in the cSC.	96
4.52	In the mcAMA, an association is created between <i>Pet</i> (origin) <i>PetObserver</i> (target).	98
4.53	<i>RaseiMeUp</i> (target) is linked with <i>Pet</i> (origin); <i>RaseiMeUp</i> (origin) is linked with <i>PetObserver</i> (target). Thus, an indirect association between <i>Pet</i> (origin) <i>PetObserver</i> (target) is built up.	98
4.54	Causes for the cases caused by CC.	99
A.1	Selected mcAM included in project 1.	I
A.2	Selected mcAM included in project 2.	II
A.3	Selected mcAM included in project 3.	III
A.4	Selected mcAM included in project 4.	IV
A.5	Selected mcAM included in project 5.	V

List of Tables

4.1	The project background (\sim = around).	49
4.2	Suspected cases caused by MA and disAGTs potentially exist in other voSC(s)/cSC(s) not selected by us previously.	51
4.3	Cases for the differences caused by MA and disAGTs (* = own case, <> = opposite).	52
4.4	The corresponding example of the case MA-1.	55
4.5	The corresponding example of the case MA-3*.	56
4.6	The corresponding example of the case MA-6.	59
4.7	Corresponding examples of the causes for the three subcases of case MO-1.	61
4.8	Six corresponding examples of the cases titled MO-2, MO-3, MO-4, MO-5*, MO-6*, and MO-7.	64
4.9	The corresponding example of the case DA-1.	86
4.10	Two corresponding examples of the causes for case DA-3*.	87
4.11	The corresponding example of the cases DA-4* and DA-5*.	88
4.12	Four corresponding examples of the cases DO-1, DO-2, DO-3 and DO-5.	91
4.13	Seven cases of the differences caused by CC.	98
4.14	Three corresponding examples of the causes for case CC-1.	101
4.15	Two corresponding examples of the causes for case CA-1.	101
4.16	Two corresponding examples of the causes for case CA-2.	102
4.17	Four corresponding examples of the causes for case CO-1.	104
4.18	Two corresponding examples of the causes for case CO-2.	105
4.19	The corresponding example of the cause for case CO-3.	105
4.20	Two corresponding examples of the cause for case CO-4 (* = particular interest).	106
4.21	Ratios of the cases with involved projects (* = own case).	107
4.22	Typical/common cases with related projects.	108
4.23	Project 1 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).	108
4.24	Project 2 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).	109
4.25	Project 3 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).	109
4.26	Project 4 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).	110
4.27	Project 5 - Respective differentiated cases of MA, disAGTs, and CC.	110

List of Acronyms

Uncommon acronyms

mcAMs	Manually created architectural models
voSC	Version(s) of source code
cSC	Conformable source code
MA	Manual abstraction
disAGTs	Disagreements
CC	Common changes

General acronyms

SDLC	Software Development Life Cycle
OOP	Object-Oriented Programming
UML	Unified Modeling Language
FOSS	Free/Open Source Software
API	Application Programming Interface
EA	Enterprise Architect
IDEA	IntelliJ IDEA
AST	Abstract Syntax Tree
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
OCL	Object Constraint Language
MDE	Model-Driven Engineering

1

Introduction

1.1 Background

The software development life cycle (SDLC) is defined by Stocia *et al.* [2] as “an environment that describes the activities performed in each stage of the software development process.” In the early stages of the SDLC, Unified Modeling Language (UML) is widely used to model and visualize system artifacts [3]. Among the various models of UML, the class diagram is the most commonly used and important diagram in object-oriented system modeling [4]. The advantage of using class diagrams in practice is that it can help software engineers—both developers and maintainers—to understand systems architectures, behaviors, design choices, and implementations [5]. Thus, it is easier for developers and maintainers to understand the structure of the system by looking at the class diagram rather than reading through the code in detail.

1.2 Challenges

Class diagrams model the information on the domain of interest in terms of objects organized in classes and relationships between them [6]. They are intensively used in the early stages of the SDLC to present the system’s structure. Maintainers benefit from using class diagrams to understand the system’s structure, and thus the places required to be modified can be located [7]. However, there are three challenges revealed by other authors that are likely to cause the implementation of the source code to deviate from the design of the class diagram. There is a concern that such divergent class diagrams cannot help developers and maintainers to understand the structure of the system in the same way.

These three challenges affect the activities performed by software engineers in different stages of the SDLC. Note that the stages of the SDLC vary depending on the source [2, 8, 9]. Figure 1.1 illustrates five of the stages: analysis, design, implementation, testing, and maintenance. This thesis focuses on three of them which involve the three challenges: design, implementation, and maintenance, respectively (as shown in Figure 1.1, stages filled in blue). Architects and their teams, developers, and maintainers are involved in each of these three stages. A detailed description of the three challenges depicted in Figure 1.1 is as follows:

- Challenge 1 - Creating various levels of manual abstraction on the elements of

the class diagram during the design stage: Osman [p. 45, 10] proposed that class diagrams with a low level of detail are used to show a high-level abstraction of the structure of the system. However, little is known about which level of abstraction the architects and their teams create during the design phase.

- Challenge 2 - Missing or unfollowed parts of the class diagram during the implementation stage: Guéhéneu [5] proposed that class diagrams produced during the design stage are often forgotten during the implementation stage, under time pressure usually. Truong *et al.* [11] investigated that many created designs are only partially followed during implementation. Thus, missing or unfollowed parts of the class diagram will cause the implementation of the source code to deviate from the design of the class diagram.
- Challenge 3 - Missing updates of the class diagram during the maintenance stage as the code evolves: Osman *et al.* [12] proposed that the frequency of updating UML models is low, and a new feature of the system is introduced in a new version/release, which should result in an update of the class diagram. We agree with the assertion by Osman *et al.* [13] that keeping diagrams up to date with code evolution is often a challenge. Figure 1.1 illustrates that code evolves over time from implementation to maintenance. However, if the class diagram remains static (without update) as the code evolves, it does not reflect the new features introduced to the system.

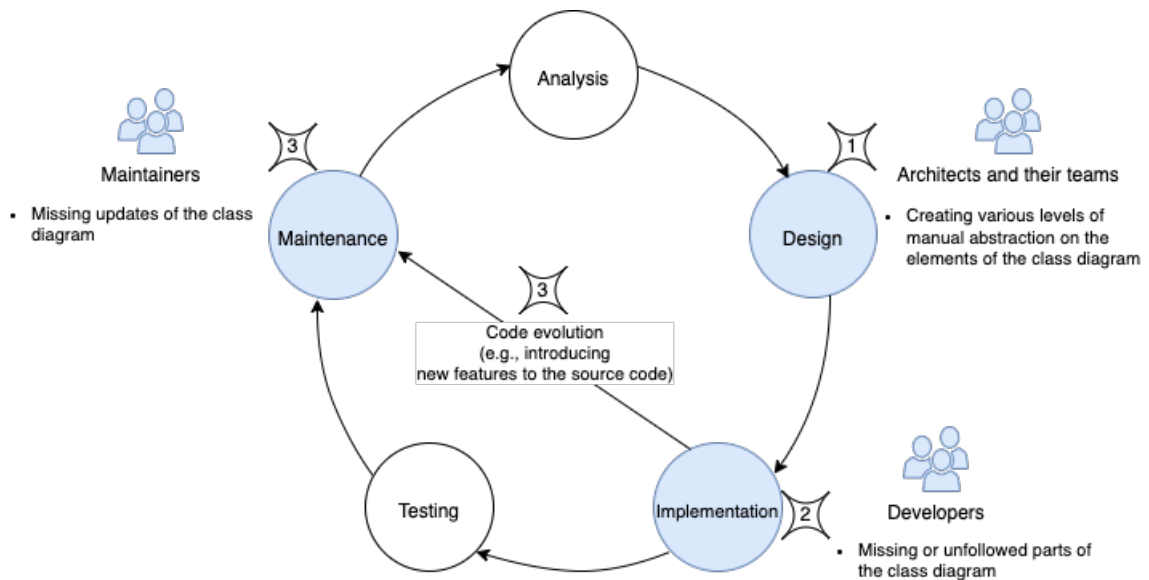


Figure 1.1: Three challenges involved in the staged SDLC are likely to cause the implementation of the source code to deviate from the design of the class diagram; thereby, differences between the class diagram and the source code are introduced.

1.3 Goal and Motivation

As aforementioned, the divergent class diagrams cannot be used in the same way by software engineers for understanding the system's implementation structure. As a solution, reverse engineering methods/tools can reverse code into class diagrams. Yet, the reverse-engineered class diagrams, in most cases, are not abstract and with extensive information which will burden the software engineers' understanding of the system's implementation structure. This is due to the inability to provide input of manual abstraction characteristics for reverse engineering tools/methods. Thus, the tools/methods cannot manage to imitate humans to abstract relevant information only. Yet, the characteristics of manual abstraction can only be achieved by flexible manual studies based on the fact humans can jointly interpret the semantics conveyed by different model elements based on a full understanding of the relevant code implementation. Thus, the goal of this thesis aims at manually discovering the characteristics of manual abstraction created in the model elements.

The goal of this thesis is motivated by the following holds:

No actual case of the models and source code is studied in terms of manual abstraction characteristics: So far as we know, the existing studies on the characteristics of manual abstraction are based on the opinions and experiences of the participants, yet do not study an actual case of models and source code. The consistency checks of the differences and similarities between the design and code are purely structural and do not take semantics conveyed by model elements into account. Yet, semantics conveyed by model elements is key for studying the manual abstraction characteristics. Given that different systems have their own specified implementation structures, the desired functionalities require code structures that are interrelated and cooperated while also taking into account the application of the specific architecture and design patterns. These factors need to be considered jointly, which can only be achieved by flexible manual studies of models and source code.

1.4 Case Study Subjects

In order to investigate the characteristics of the differences between the source code and class diagram, we employed the methodology of five case studies. It is necessary to access a dataset that includes a set of projects with class diagrams and corresponding source code. However, such a dataset is rare and difficult to access since the industrial models and source code is often not accessible for research. To address this issue, we decided to make use of models used in Free/Open Source Software (FOSS) projects. Thus, we used Lindholmen Dataset [1] created by Hebig *et al.* to do this thesis work.

The Lindholmen Dataset [1] holds 3 295 open source projects of GitHub, which include together 21 316 UML models. The model files are in two formats (images and

standard files). We first selected 5 projects in Java programming languages from that dataset as our study subjects. Then the selected class diagrams introduced to the project are limited to image format only, and we referred to them as **manually created architectural models (mcAMs)**. The model elements we studied are classes, attributes, operations, and relationships (i.e., dependencies, usages, associations, aggregations, compositions, inheritances, and realizations).

1.5 Definition of Terminologies

We defined the following terminologies or used definitions given by others, enabling the reader to understand the research questions (RQs) formulated in section 1.6.

As defined in the source [14], **commit** is a snapshot of changes made to the staging area, where holds the files to be included in the next commit.

Version(s) of source code (voSC) is represented by a collection of source code files found in the repository after a commit (before next commit).

Concepts are described by classes created in the mcAM. To be specific, a concept can be described by an abstract (super-) class of the class or a non-abstract (normal) class of the class.

Note that regarding the definition of **concepts**, one can argue that concepts can be described by both classes and relationships between these classes from the mcAM. However, we argue that concepts are described only by classes from the mcAM. This is because the differences in attributes and operations from the classes between the mcAM and the source code would lead to differences in relationships. These correlated differences we aim to study.

Map to a voSC refers to mapping one or more concepts described in the mcAM to one or more classes of that voSC. Note that human judgments are involved in the mapping since the naming of classes in the mcAM and source code might be different.

A voSC v is conformable to the mcAM if for every concept described in the mcAM the following holds: for the concept a there is a Map to the voSC v , or there is a second concept b in the mcAM described by the superclass of the class that describes the concept a and for this second concept b there is a Map to that voSC v .

The latter case can be illustrated by the following example:

Example: This example is taken from the repository of one of our study projects, i.e., ZooTypers [15] on GitHub. As seen in Figure 1.2, there are five concepts in the mcAM: *Pet*, *Dog*, *Cat*, *Fish*, and *Penguin*, which are described by the superclass *Pet*, and four subclasses *Dog*, *Cat*, *Fish*, and *Penguin* that are derived from that superclass *Pet*, respectively. Figure 1.3 illustrates that a voSC that only includes

a class *Pet* that can be mapped to the superclass *Pet* of the class, e.g., *Dog* in the mcAM is conformable to the mcAM, since for the concept, e.g., *Dog*, there is a second concept *Pet* described by the superclass *Pet* of the class *Dog* that describes the concept *Dog*, and for this second concept *Pet*, there is a Map to that voSC.

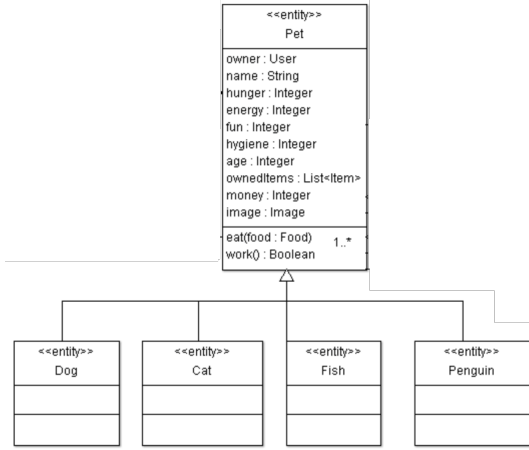


Figure 1.2: In the mcAM, for the concept, e.g., *Dog*, there is a second concept *Pet* is described by the superclass *Pet* of the class *Dog* that describes the concept *Dog*.

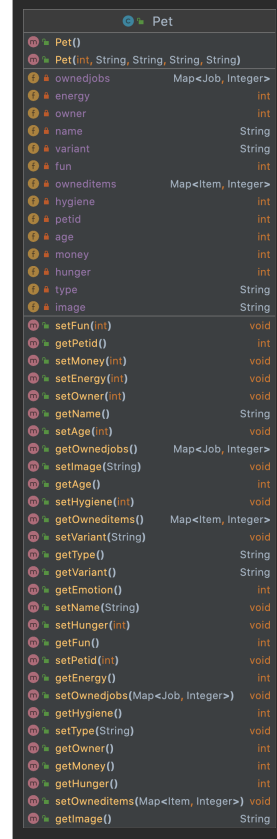


Figure 1.3: For the superclass *Pet* of the class *Dog* that describes a second concept *Pet* of a concept *Dog* in the mcAM, the class *Pet* in this voSC is considered to be mapped to that superclass *Pet* of the class *Dog* in the mcAM.

Conformable source code (cSC) refers to a voSC that is conformable to the mcAM.

1.5.1 Examples of Mappings between mcAM Concepts and voSC

To illustrate how mappings are created between mcAM concepts and voSC, the following examples are brought out:

Note that regarding the following examples, for a concept *a*, the naming of the class *b* that describes the concept *a* in the mcAM and the naming of the mapped class *c* of

the class b in the voSC v might be different. However, class b and class c are ontologically identical since they describe the same concept with highly similar attributes and operations and thereby, class b in the mcAM is considered to be mapped to class c in the voSC v .

Case 1 - A class that describes a concept in the mcAM can be mapped to one class in the voSC.

Example of Case 1: This example is taken from the repository of one of our study projects, i.e., ZooTypers [15] on GitHub. As observed in the comparison between Figure 1.4 and Figure 1.5, the naming of the class, i.e., *SinglePlayModel* in the mcAM differs from the naming of the class i.e., *SinglePlayerModel* in the voSC. This might be due to a misspelling of the name of the class *SinglePlayModel* in the mcAM. However, the class *SinglePlayModel* in the mcAM and the class *SinglePlayerModel* in the voSC are ontologically identical since they both have highly similar attributes and operations and thereby these two classes are considered to describe the same concept *SinglePlayerModel*. Therefore, the class *SinglePlayModel* in the mcAM is considered to be mapped to the class *SinglePlayerModel* in the voSC.

SinglePlayModel
-currWordIndex: int [1] //(index of current word in words displayed)
-currLetterIndex: int [1] //(index of current letter in current word)
-wordsDisplayed: int [1] [1] //array of 5 represent indices in words list
-wordsList: String [1] [1] //array = [apple, bear, cat, dog, elephant, ...]
-nextWordIndex: int [1] //(index of next word in words list)
-score: int [1] //(current user score, # of letters typed)
-numWordsDisplayed: int [1] //final value
-am: AssetManager [1]
-currFirstLetters: Set<Character> [1]
+SinglePlayModel (diff: States.difficulty, am: AssetManager, wordsDis int)
-getWordsList (diff: States.difficulty)
+populateDisplayedList()
+typedLetter (letter: Char) //takes in the letter typed
-updateWordsDisplayed() //notify view to put words on screen

Figure 1.4: A class *SinglePlayModel* describes a concept *SinglePlayerModel* in the mcAM, yet the naming of the class *SinglePlayModel* with a misspelling.

SinglePlayerModel
+ SinglePlayerModel(difficulty, AssetManager, int)
+ wordsList String[]
+ currWordIndex int
+ score int
+ wordsDisplayed int[]
+ nextWordIndex int
+ currFirstLetters Set<Character>
+ am AssetManager
+ numWordsDisplayed int
+ currLetterIndex int
+ typedLetter(char) void
+ populateDisplayedList() void
+ getCurrWordIndex() int
+ getScore() int
+ getWordsList(difficulty) void
+ getCurrWord() String?
+ getCurrLetterIndex() int
+ updateWordsDisplayed() void

Figure 1.5: The class *SinglePlayerModel* in this voSC has highly similar attributes and operations with the class *SinglePlayModel* in the mcAM and thus, the class *SinglePlayModel* is considered to be mapped to the class *SinglePlayerModel* in this voSC.

Case 2 - A class that describes a concept in the mcAM can be mapped to more than one class in the voSC.

Example of Case 2: This example is taken from the repository of one of our study projects, i.e., RaiseMeUp [16] on GitHub. Figure 1.6 illustrates that a concept *decorators* is described by a class *ItemDecorator* in the mcAM. Decorators are part of the decorator design pattern [17]. Thereby, the naming of the class *ItemDecorator*

Example of Case 3: This example is taken from the repository of one of our study projects, i.e., RaiseMeUp [16] on GitHub. Figure 1.8 illustrates that in the mcAM, for the concept, e.g., *Dog*, a second concept *Pet* is described by the superclass *Pet* of the class that describes the concept *Dog*. For the second concept *Pet*, there is a Map to this voSC, i.e., mapping that superclass *Pet* in the mcAM to one class *Pet* in the voSC (see Figure 1.9).

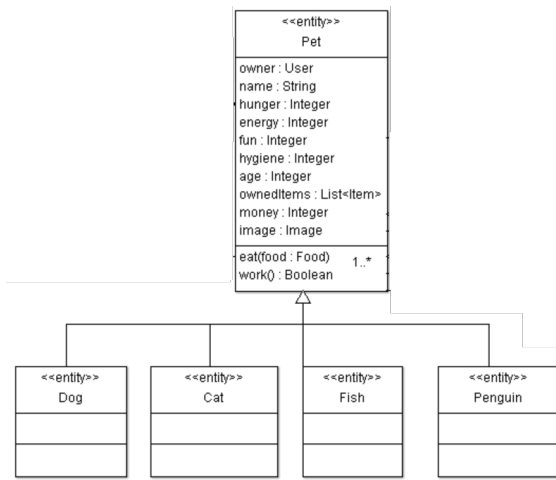


Figure 1.8: In the mcAM, for the concept, e.g., *Dog* a second concept *Pet* is described by the superclass *Pet* of the class *Dog* that describes the concept *Dog*.

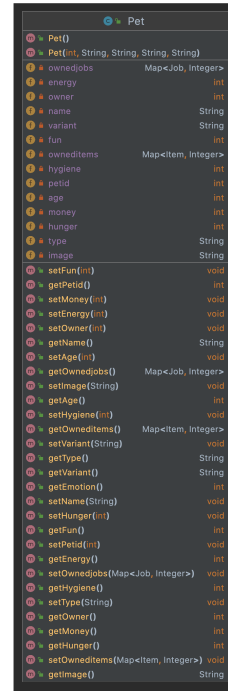


Figure 1.9: This voSC only includes one class *Pet* that can be mapped to the superclass *Pet* of the class, e.g., *Dog* in the mcAM.

Case 4 - A class that describes a concept *a* in the mcAM can be mapped to an additional superclass (not in the mcAM, but added to the voSC) that describes a concept *a* in the voSC (ignoring that one or more additional subclasses derived from that superclass (not in the mcAM, but added to the voSC as well) describe one or more additional concepts (not in the mcAM, but added to the voSC)).

Example of Case 4: This example is taken from the repository of one of our study projects, i.e., Neuroph [18] on GitHub. Figure 1.10 illustrates that a concept, i.e., *InputFunction* is described by a class *InputFunction* in the mcAM. Figure 1.11 illustrates that this concept *InputFunction* remains in the voSC and is described by one additional superclass *InputFunction* (not in the mcAM, but added to the cSC). Therefore, the class *InputFunction* in the mcAM is considered to be mapped to that superclass *InputFunction* in the voSC.

Observed from the comparison between Figure 1.10 and Figure 1.11, an additional concept, e.g., *Max* not in the mcAM is added to the voSC and is described by an additional subclass *Max* (not in the mcAM is added to the voSC). For this additional concept *Max*, there is a second concept *InputFunction* described by the superclass *InputFunction* of the class *Max* that describes the additional concept *Max* in the voSC and for this second *InputFunction*, the superclass *InputFunction* is considered to be mapped to the class *InputFunction* in the mcAM.

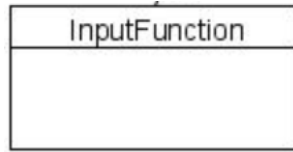


Figure 1.10: A concept, i.e., *InputFunction* is described by a class *InputFunction* in the mcAM.

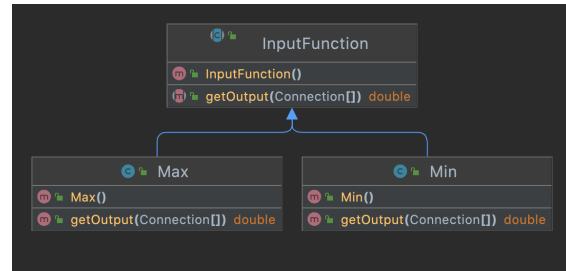


Figure 1.11: The concept *InputFunction* in the mcAM is described by a superclass *InputFunction* that not in the mcAM but added to the voSC.

1.5.2 Ideal Selection of One cSC among Multiple cSCs of a mcAM

In a project's GitHub repository, commits are throughout the SDLC as the code evolves. Thus, a project has different voSC as the code evolves. A mcAM may have one or more cSCs, i.e., one or more voSCs that are conformable to the mcAM. However, considering the time constraints and we want to study more projects, we decided to select only one cSC of them. Compared with other cSCs of the mcAM, this cSC should ideally cover the most attributes and operations associated with the concepts in the mcAM. However, this cannot be guaranteed since it is not possible for us to check the voSC one by one (referring to the detailed methodology employed illustrated in section 3.2). This will lead to a threat, which will be illustrated in section 5.6.

To illustrate how one cSC among multiple cSCs for a mcAM is selected, the following example is given:

This example is taken from the repository of one of our study projects, i.e., EAPLI_PL_2NB [19] on GitHub. Figure 1.12 illustrates in the mcAM there are eight concepts that are described by eight classes, i.e., *BaseUI*, *BaseController*, *IncomeRegisterUI*, *RegisterIncomeController*, *Income*, *IncomeTyperRepository*, *CheckingAccount*, and *IncomeRepository*, respectively.

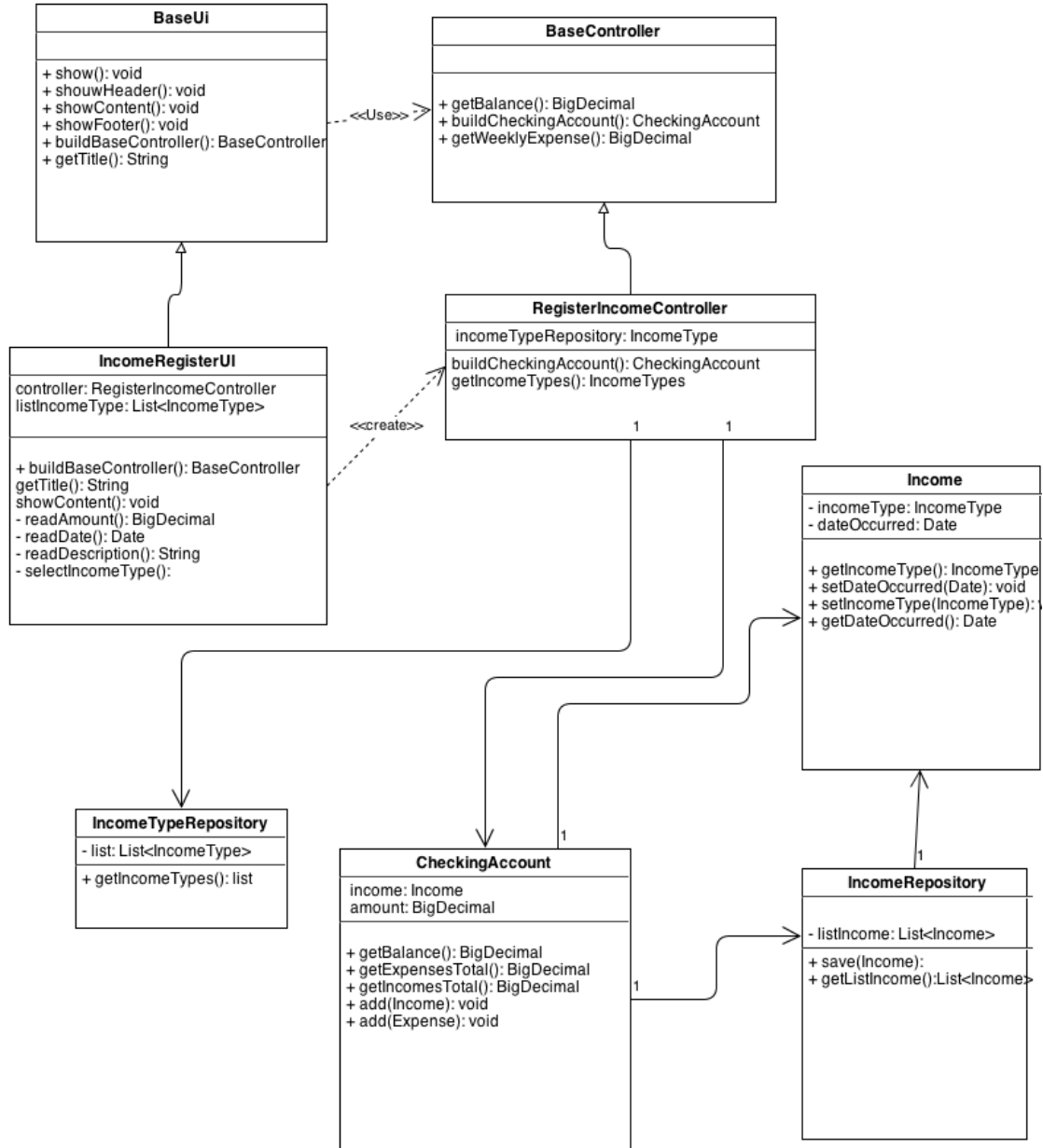


Figure 1.12: In the mcAM, eight concepts are described by eight classes, i.e., *BaseUI*, *BaseController*, *IncomeRegisterUI*, *RegisterIncomeController*, *Income*, *IncomeTypeRepository*, *CheckingAccount*, and *IncomeRepository*, respectively.

attributes and operations) planned out in that mcAM. This will cause the differences between the cSC of that mcAM and that mcAM.

RQ2: *What causes the differences between the cSC of the mcAM and that mcAM?* With the answer to RQ2, we can get a list of cases that cause the differences between the cSC of the mcAM and that mcAM. Could these cases be categorized into some common causes?

RQ3: *What are the differences between the cSC of the mcAM and that mcAM?* With the answer to RQ3, we can conclude some common causes that cause the differences between the cSC of the mcAM and that mcAM.

1.7 Contributions

The results of this thesis provide a sorted list of cases that cause the cSC to deviate from the mcAM and a sorted list of suspected cases inferred by these observed cases. These suspected cases are considered to exist possibly and would also lead to the differences between the mcAM and the cSC. In accordance, this thesis will have the following implications for the reverse engineering and modeling community:

1. To provide input for future improvement of reverse engineering methods/tools in terms of abstractness: So far as we know, reverse engineering is imperfect as it does not manage to imitate the human ability to abstract relevant information from the source code. Guéhéneu [5] proposed that no existing mainstream reverse engineering tool produces abstract yet precise class diagrams. The concluded cases of creating various manual abstraction on the model elements can be used as such input.

2. To provide input for developing mapping rules which can be used for the consistency check(s) between the model design and code implementation: Existing methods/technologies developed for the consistency checks between code and design are purely structural and do not take the semantics conveyed by the model elements into account. Yet, the semantics is closely related to the manual abstraction characteristics. Thereby, the sorted list of cases of the differences between the code and design, which was yielded from manually studying five Java projects based on interpreting the semantics of the model elements, can provide such input.

3. To provide a guideline for designing model elements to avoid over-abstraction and over-specification: Given the abstract nature of the model, the model elements can be modeled at various levels depending on the design decisions made by architects. When it comes to design decisions for creating different elements of the class diagram during the design stage, little is known about which design decisions are inclined to be acceptable and unacceptable by developers in the code implementation. This can result from over-specifying the model elements yet losing the abstractness. Thereby, the developers disagree with these design decisions made by architects, and they make different decisions in the code implementation. On

the other hand, this can also result from over-abstracting the model elements. This leads to vague design decisions that cannot be accepted by developers in the code implementation, given that the developers need to settle these vague design decisions down and further specify the detailed implementation for these decisions. Then the deviations of the code from the design come up. Thus, the concluded cases of the differences caused by developers' deviations from the architects' design decisions allow us to create this guideline.

1.8 Structure of the Paper

This thesis presents a systematic manual study of the characteristics of the differences between the mcAM and one cSC of that mcAM by analyzing five open-source Java projects on GitHub. The structure of this thesis is outlined as follows: In Chapter 2, the relevant theoretical knowledge and early research done by others are described. Chapter 3 details the methodology of the five case studies employed. The results of this thesis work are illustrated in Chapter 4. The threats to the validity of this thesis are described and discussed in Chapter 5. This thesis work is concluded, and the future work of this thesis is suggested in Chapter 6.

2

Theory and Related Work

In this chapter, in order to help the reader understand the work of this thesis, the relevant theoretical knowledge of Java and UML is first described. This lays the ground for understanding the constituents of a mcAM and thereby every constituent in that mcAM can be mapped to the corresponding constructs of one cSC of that mcAM. After that, related early work done by other authors on reverse engineering and models in open-source systems is presented and discussed. The former work provided the inspiration for this thesis and from which this thesis originated. The study subjects of this thesis relied on the outcomes of the latter work.

2.1 Theory

In order to detect the differences between the mcAM and the cSC of that mcAM, for each model constituent in the mcAM, there must be a map to the corresponding construct(s) of the cSC of that mcAM. Only with knowledge of Java and UML can one understand how to map every element in a mcAM to the corresponding construct(s) of a cSC of that mcAM.

As Java and UML evolve, multiple versions of their specifications exist at different times. On the other hand, the mcAM and the first voSC found in the repository of each project were created at different times. For the five projects studied, in order to ensure that these mcAMs and voSCs included in the projects matched the appropriate versions of Java and UML specifications, respectively, it is critical to identify when the earliest mcAM and voSC were created in the repository. This is because the creation date of the Java specifications and UML specifications used should be ideally as close as possible to the creation date of the earliest created mcAM and voSC, and in turn, later versions have new updates that may not adapt to these mcAMs and voSCs. After checking, the earliest mcAM and voSC were created on July 8, 2011, and August 24, 2011, respectively. In consequence, Java SE7 specifications (released in July 2011) [20] and UML v2.4.1 superstructure specifications (released in July 2011) [21], respectively, were selected as the basis for this study.

2.1.1 Related Java Knowledge

Relevant Java knowledge needs to be understood, including Java syntax/specifications and the OO paradigm. Of particular interest part Java specifications, along with several related concepts in the OO paradigm, which can help to understand how

abstraction is able to be created over the cSC.

2.1.1.1 Java SE7 Specifications

Of particular interest part of Java SE7 specifications is illustrated in the following:

Framework is a set of classes and interfaces which provide a ready-made architecture [22].

Collection framework provides the ready-made classes and interfaces needed to represent a group of objects (also called instances) as a single entity in Java [22]. For example, the *Map* interface with the corresponding classes, e.g., *HashMap*, is used to present a group of instances.

2.1.1.2 Object-Oriented (OO) Paradigm

Several related concepts of the OO paradigm are described in the following, according to the source [20, 23, 24, 25, 26, 27].

Object are often referred as an instance or an array of a class in Java and all objects created belong to a certain class [20, 25]. Objects are an encapsulation of information and behavior relative to some entity of the application domain under consideration [25]. In real systems many objects with similar information (data) and behavior (functionality) can be found [25].

Class captures those objects with similar information (data) and behavior (functionality) and classes can be viewed as an abstract data type [25]. Class is defined as including at least two types of features: **attributes** (also called **variables**, **fields** or **data members**), which stand for the stored information and **methods** (also called **operations** or **function members**), which represent the behavior [25].

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces [26]. An encapsulated module can only be accessed by clients (that is, other modules that make use of this module) via this interface [27]. Implementation details are “hidden” within the module. The primary reason for requiring encapsulation is to make it possible to change (improve) the implementation of a module without having to change (and/or recompile) the module’s clients [27].

Take Java as an example. For the encapsulation of attributes included in a class, all attributes about that class should be set to private unless they are specifically declared public [25]. The public setter and getter operations set for the attributes of a class are called its interfaces and should only be the “tip of the iceberg” with the hidden part that is called the implementation [25]. Those interfaces of a class allow the supplier class to render the values of those attributes to the customer class [25].

Inheritance allows the subclass that extends the superclass to be arranged in a hierarchical structure [24, p. 451], and thereby a subclass to take on the general attributes and operations of that superclass in the inheritance chain so that attributes and operations then form part of the definition of the subclass for code reuse [23, p. 63].

2.1.2 Unified Modeling Language (UML)

UML is a modularly structured language that can provide specific components of primary interests for a specific domain or application [21, p. 1]. UML is a de facto standard formalism for software design and analysis [6]. With some existing case tools such as Enterprise Architect [28] and IntelliJ IDEA [29], specific constituents of UML can be handily visualized to accommodate the specific requirements. Of particular interest are class diagrams used for modeling the information on the domain of interest in terms of objects (instances) organized in classes and relationships between them [6]. Thus, the specific constituents of UML that are most likely to be required in most cases for constructing a class diagram are classifiers (classes), classifiers' (classes') embedded text notation for attributes and operations, and relationships between classifiers (classes). As mentioned above, they can all be visualized with a case tool; these constituents are detailed separately in this section.

2.1.2.1 Graphical Notation for Classifiers

Classifier refers to a classification of instances describing a set of instances that have features in common [21, p. 51], in which the textual notation for attributes and operations is embedded.

Figure 2.1 presents examples of graphical notation for a class *Window* at three different levels of abstraction: suppressed (on the top left corner), analysis (on the right), and implementation (on the bottom left corner) [21, p. 50]. However, for different systems, at which level or in the fluctuations between these levels a class is constructed actually depends on different design decisions made by different architects during the design stage. In some cases, they may determine to model a specific part of the system that is of primary interest at a low level (e.g., an implementation level) to give developers more insight into that part of the system during the implementation stage. On the contrary, for the part of a system that is of less interest, they may determine to model that part of the system at a higher level relative to the level of implementation (e.g., a suppressed or an analysis level).

As seen in Figure 2.1, if the class *Window* is constructed at an analysis or implementation level, details of its embedded textual notation for attributes and operations are (more or less) laid out. However, the meta-textual notation defined for them in [21] is far more comprehensive than any of the three illustrated in Figure 2.1. Take Figure 2.1 as an example, aiming at leaving the reader with an impression of what a class is possibly like at various abstraction levels. That means in general, the layout of a graphical class is composed of three primary sections. These three sections are elaborated in the following with the aid of a given example shown on the right of

Figure 2.1.

- **Upper section (mandatory):** Contains the name of the class, e.g., *Window* [30].
- **Middle section (optional):** Contains one or more attributes of the class *Window*, and they are used to describe the qualities of *Window* [30], e.g., the attribute *size: Area* is used to describe the size of an instance of the class *Window*. Noteworthy, this section is only required when describing a specific instance of a class [30].
- **Bottom section (optional):** Includes class operations displayed in list format, each operation, e.g., *hide()* takes up its own line [30]. The operations describe how a class interacts with data [30], e.g., for the operation *attachX(xWIN: XWindow)*, the class *Window* references class *XWindow* as a parameter data type, and thereby an interaction between the class *XWindow* and the class *Window* comes out.

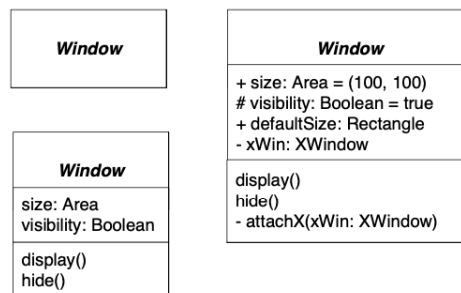


Figure 2.1: Three abstraction levels of graphical class notation: suppressed (on the top left corner), analysis (on the right), and implementation (on the bottom left corner) [21, p. 50]. Compared with the suppressed level with only the specified name of a class, another two levels present a more comprehensive layout of a class with more or less) textual notation for attributes and operations embedded.

2.1.2.2 Conversions of BNF

To standardize the textual notation for attributes and operations embedded in the classifier, legal formats are first specified, i.e., the Backus-Naur Form (BNF) conversions (as depicted in Figure 2.2). The legal formats make the textual notation for attributes and operations more easily interpreted.

Note that the specification of BNF conversions applies to both earlier serial UML v1.0 specifications series and the latest serial v2.0 specifications series.

- All non-terminals are in italics and enclosed between angle brackets (e.g., *<non-terminal>*).
- All terminals (keywords, strings, etc.), are enclosed between single quotes (e.g., 'or').
- Non-terminal production rule definitions are signified with the '::<=' operator.
- Repetition of an item is signified by an asterisk placed after that item: '*'.
- Alternative choices in a production are separated by the '|' symbol (e.g., *<alternative-A> | <alternative-B>*).
- Items that are optional are enclosed in square brackets (e.g., [*<item-x>*]).
- Where items need to be grouped they are enclosed in simple parenthesis; for example:

*(<item-1> | <item-2>)**

signifies a sequence of one or more items, each of which is *<item-1>* or *<item-2>*.

Figure 2.2: Specification of Backus-Naur Form (BNF) conversions [21, pp. 16–17].

2.1.2.3 Textual Notation for Attributes

Reference to [21, pp. 129–130], the notation for attributes defined is depicted in Figure 2.3.

Note that attributes are a legacy terminology of the earlier UML v1.0 specifications and are referred to as properties in the UML v2.4.1 superstructure specifications. Property (denoted in Figure 2.3) and attribute are ontologically identical. Terminology attributes are used in this thesis.

<property> ::= [<visibility>*] [*'/'*] *<name>* [*':'* *<prop-type>*] [*'['* *<multiplicity>* *']'*] [*'='* *<default>*]
['{' *<prop-modifier>* *','* *<prop-modifier>* *']** *'}'*]*

where:

- *<visibility>* is the visibility of the property. (See "VisibilityKind (from Kernel)" on page 141.)
- *<visibility>* ::= '+' | '-' | '#' | '~'
- *'/'* signifies that the property is derived.
- *<name>* is the name of the property.
- *<prop-type>* is the name of the Classifier that is the type of the property.
- *<multiplicity>* is the multiplicity of the property. If this term is omitted, it implies a multiplicity of 1 (exactly one). (See "MultiplicityElement (from Kernel)" on page 95.)
- *<default>* is an expression that evaluates to the default value or values of the property.
- *<prop-modifier>* indicates a modifier that applies to the property.

*<prop-modifier> ::= 'readOnly' | 'union' | 'subsets' <property-name> |
'redefines' <property-name> | 'ordered' | 'unique' | 'nonunique' | 'id' | <prop-constraint>*

where:

- *readOnly* means that the property is read only.
- *union* means that the property is a derived union of its subsets.
- *subsets <property-name>* means that the property is a proper subset of the property identified by *<property-name>*.
- *redefines <property-name>* means that the property redefines an inherited property identified by *<property-name>*.
- *ordered* means that the property is ordered.
- *unique* means that there are no duplicates in a multi-valued property.
- *id* means that the property is part of the identifier for the class.
- *<prop-constraint>* is an expression that specifies a constraint that applies to the property.

Figure 2.3: Textual notation for attributes [21, pp. 129–130].

This thesis work focuses on five constituents of the attributes' textual notation: *name*, *prop-type*, *multiplicity*, *default* and *visibility* [21, p. 129] (as depicted in Figure 2.3, underlined in pink). These five constituents are referred to as *name*, *attribute type*, *multiplicity*, *default value*, and *visibility*, respectively, in this thesis. The definitions of these constituents used in this thesis and the reasons why they are of interest are detailed in the following.

Note that although their corresponding definitions are depicted in Figure 2.3, some of them may still need to refer to earlier UML v1.5 specifications [31]. In this way, these two versions of definitions are able to complement each other. This will enhance the comprehensibility of those constituents.

- Reference to the definition of name in the UML 1.5 specifications, Chapter 3, Part 5, Section 3.25 “Attribute”, **name** is an identifier string, usually a simple word, to represent an attribute [31, p. 42].

Names are mandatory for attributes' textual notation. Names are not mere identifiers for attributes; in particular, they carry relevant semantics related to the static data structure of the classifier [32]. Semantics preservation is a main objective of the refinement of design into code [32]. Thereby, attributes names are expected to enhance our comprehensibility on mappings between the attributes in the mcAM and cSC. That means based on the semantics related to the attributes in the mcAM, their corresponding attributes in the cSC that represent similar semantics are able to be identified.

Note that considering attributes in the mcAM can be referred to as either variables or constant variables in the cSC. In accordance, an attribute in the mcAM can be modeled as a variable or a constant variable that will be implemented in the cSC. Referring to Java naming conventions, the naming of variables should be in camel case [33] and the naming of the declared class constant variables should be all in uppercase letters with words separated by underscores (“_”) [34]. These naming conversions are represented in the same way as they are in the mcAM. Thus, by observing the naming conversions of attributes in the mcAM, one can infer whether an attribute in the mcAM can be mapped to a variable or a constant variable in the cSC.

- Reference to UML v1.5 specifications, Chapter 3, Part 5, Section 3.25 “Attribute” [31, p. 42], **attribute type** refers to either name of the classifier or a language-dependent string that maps into a primitive data type in Java.

There are two reasons for us to study the differences in attribute types. The first reason is that the attribute types might be related to relationships between classifiers (also instances). For example, the relationships of *association*, *aggregation*, and *composition* between classifier A and B should first satisfy that classifier A references classifier B as the type of an attribute included in classifier A. Then, to confirm exactly the relationship between classifiers A and

B, based on the definitions of relationships in this section, it demands us to check the detailed implementation of the cSC.

Although the attribute type (name of the classifier) is critical for understanding the relationships between classifiers, the attribute type is an option (as illustrated in Figure 2.3). In essence, the possibility of omitting the attribute types by architects when designing the mcAM cannot be excluded. This is the second reason. The omission of the attribute types is a kind of abstraction. Opposite to this case, there would be two other cases caused by developers' disagreements in the implementation of the cSC with respect to the abstraction created by architects in the mcAM. These two cases are **1.** The attribute type in the mcAM is removed in the cSC. **2.** The attribute type in the mcAM and cSC is specified differently.

- **Multiplicity** is specified in the textual notation for both attributes and operations. Thereby, its definition and the reason why it is a focus in this thesis are detailed separately in subsection 2.1.2.6.
- **Default value** is an expression that evaluates to the default value or values of the attribute [21, p. 113]. Referring to the notation for attributes depicted in Figure 2.3, the default value is an option. Thus, assume a possibility that when designing attributes/variables in the mcAM, some architects may intend to omit the specification of default values and choose to leave them out for developers to initialize the variables in the cSC. The reason for this is that these architects may have taken into account the need to cater to new requirements in the future, and thus the values of variables will be updated by developers one or more times during the implementation of the cSC. This will cause the default value not in the mcAM to be added to the cSC.

Opposite to the omission of the default values of the attributes (variables), some architects may over-specify the default values of attributes (variables) in the mcAM. However, developers may disagree with this, and they choose to remove the specified values in favor of other approaches, such as adopting public setter operations of these attributes (variables) to initialize the attributes (variables) and update their values one or more times in the cSC. Note that the prerequisite for using public setter operations is that the attributes (variables) declared in the classifier should be set to private.

On the other hand, for the design of attributes (constant variables) in the mcAM, considering that a constant variable should be assigned a value once across the life-cycle of the program, a query is whether the architects intended to specify a default value for a constant variable in the mcAM or not.

As mentioned previously, there would be a case - the differences in attributes caused by the conversion of variables and constant variables between the mcAM and cSC. Thus, another query as to whether this case has an impact on

the specification of default values in the mcAM and cSC. As we all know, the variables are likely to be updated one or more times in the cSC. In particular, assume that variables in the mcAM, which may not hold default values, are converted to constant variables with default values assigned in the cSC. This will lead to another case - the default value not in the mcAM is added to cSC. However, differing from the former case, this case is considered to be caused by the deviations between the implementation of the cSC and the design of the mcAM.

- **Visibility** is specified in the textual notation for both attributes and operations. Thereby, its definition and the reason why it is a focus in this thesis are detailed separately in subsection 2.1.2.5.

Except for the five constituents of attributes' textual notation illustrated above, the other two constituents of `/` and `prop-modifier(attr-modifier)` are not considered to be the focus due to the absence of the relevant cases in the five projects studied.

2.1.2.4 Textual Notation for Operations

Reference to [21, pp. 107–108], the textual notation for operations defined is depicted in Figure 2.4.

```
[<visibility>] <name> '(' [<parameter-list> ')' [<return-type>] [<multiplicity> '']
['{' <oper-property> ',' <oper-property>* '}]
```

where:

- `<visibility>` is the visibility of the operation (See “VisibilityKind (from Kernel)” on page 141).

```
<visibility> ::= '+' | '-' | '#' | '~'
```

- `<name>` is the name of the operation.
- `<return-type>` is the type of the return result parameter if the operation has one defined.
- `<multiplicity>` is the multiplicity of the return type. (See “MultiplicityElement (from Kernel)” on page 95).
- `<oper-property>` indicates the properties of the operation.

```
<oper-property> ::= 'redefines' <oper-name> | 'query' | 'ordered' | 'unique' | <oper-constraint>
```

where:

- `redefines <oper-name>` means that the operation redefines an inherited operation identified by `<oper-name>`.
- `query` means that the operation does not change the state of the system.
- `ordered` means that the values of the return parameter are ordered.
- `unique` means that the values returned by the parameter have no duplicates.
- `<oper-constraint>` is a constraint that applies to the operation.
- `<parameter-list>` is a list of parameters of the operation in the following format:

```
<parameter-list> ::= <parameter> ['<parameter>']*
<parameter> ::= [<direction>] <parameter-name> ':' <type-expression>
['{' <multiplicity> '}' ['<default>] ['{' <parm-property> ',' <parm-property>* '}]
```

where:

- `<direction>` ::= `'in'` | `'out'` | `'inout'` (defaults to `'in'` if omitted).
- `<parameter-name>` is the name of the parameter.
- `<type-expression>` is an expression that specifies the type of the parameter.
- `<multiplicity>` is the multiplicity of the parameter. (See “MultiplicityElement (from Kernel)” on page 95).
- `<default>` is an expression that defines the value specification for the default value of the parameter.
- `<parm-property>` indicates additional property values that apply to the parameter.

Figure 2.4: Textual notation for operations [21, pp. 107–108].

This thesis work focuses on four constituents of the operations' textual notation:

name, *parameter-list*, *return-type*, and *visibility* [21, p. 107] (as shown in Figure 2.4, underlined in pink). They are referred to *name*, *parameter list*, and *return type*, and *visibility*, respectively, in this thesis. The definitions of these constituents used in this thesis and the reasons why they are of interest are detailed in the following.

Note that although their corresponding definitions are depicted in Figure 2.4, some of them may still need to refer to earlier UML v1.5 specifications [31]. In this way, these two versions of definitions are able to complement each other. This will enhance the comprehensibility of those constituents.

- Reference to the definition of name in the UML 1.5 specifications, Chapter 3, Part 5, Section 3.26 “Operation”, **name** is defined as an identifier string to represent an operation [31, p. 44].

Names are mandatory for the operations’ textual notation. Names are not mere identifiers for operations; in particular, they carry relevant semantics related to the behavioral status of the classifier [32]. Semantics preservation is a main objective of the refinement of design into code [32]. Thereby, operations names are expected to enhance our comprehensibility on mappings between the operations in the mcAM and cSC. That means based on the semantics related to the operations in the mcAM, we are able to identify their corresponding operations in the cSC that represent similar semantics. Sometimes, the semantics of the operations in the mcAM might remain in the cSC. However, for the operations, their implementation in the cSC might deviate from their design in the mcAM. This is because developers disagree with the design decisions of operations made by architects in the mcAM; rather, they make different decisions of operations in the cSC. The characteristics of such deviations in operations between the mcAM and cSC are what this thesis aims to study.

- **Parameter list** is defined as a list of parameters of the operation [21, p. 108] (as depicted in Figure 2.4).

Considering Java syntax, i.e., the construct of an operation, e.g., a parameter cannot be specified with a default value. Thereby, the *default value* will be excluded in this thesis work. For the parameter, none of the five projects studied have relevant cases with respect to the constituents (i.e., *parm-property* and *direction*). Thus, these two constituents will also be excluded. To this end, only three constituents (i.e., *parameter-name* and *type-expression*, and *multiplicity*) of the *parameter* are the focuses of this thesis work. They are referred to *parameter name* and *parameter type*, and *multiplicity*, respectively, in this thesis.

- Reference to the definition of parameter name in the UML 1.5 specifications, Chapter 3, Part 5, Section 3.26 “Operation”, **parameter name** is defined as an identifier string to represent a parameter [31, p. 45].

Parameter names are mandatory for the specification of operations. However, compared with the names specified for attributes and operations, here the parameter names are considered more related to identifiers for operations parameters. The reason for this is that the semantics of the names specified for operations can help us already to do mappings between the operations in the mcAM and cSC.

- **Parameter type** is defined as an expression that specifies the type of the parameter [21, p. 108] (as depicted in Figure 2.4). The parameter type can be either primitive data type or nonprimitive data type that is represented by *name of the classifier*.

For the specification of parameters, compared with the parameter names, parameter types should be taken more concern, since the parameter type(s) of an operation is(are) related to the relationships between classifiers. For example, classifier A references classifier B as a parameter type. This implies a dependency between classifiers A and B (i.e., classifier A depends on classifier B for its implementation). However, if the parameter type specified in the mcAM is changed or removed in the cSC, which will further lead to changes in the relationships between these classifiers (or involving other classifiers). Such differences in relationships between classifiers in the mcAM and cSC are what this thesis aims to study.

- **Multiplicity** (referring to the subsection 2.1.2.6)
- Reference to UML v1.5 specifications, Chapter 3, Part 5, Section 3.25 “Operation”, **return type** is defined as a language-dependent specification of the implementation type (i.e., the primitive data type in Java), or types of the value returned by the operation (i.e., the nonprimitive data type in Java) [31, p. 45].

Note that the colon and the return type are omitted if the operation does not return a value (as for Java *void*) [31, p. 45]. Thereby, if no return value is specified for an operation in the mcAM, yet *void* is shown as a return type in the cSC, we do not regard this change as a difference in return types between the mcAM and cSC.

The reason for us to study the differences in return types is that the differences in return types might lead to the changes in relationships between classifiers in the mcAM and cSC. For example, classifier A references classifier B as a return type. This implies a dependency between classifiers A and B (i.e., classifier A depends on classifier B). If the return type from classifier B is changed or removed as a *void*, this will further lead to changes in the relationship between these classifiers (or involving other classifiers). Such differences in relationships between classifiers in the mcAM and cSC are what this thesis aims to

study.

- **Visibility** (referring to the subsubsection 2.1.2.5)

Except for the four constituents of operations' textual notation illustrated above, the other constituent *oper-property* is not the focus of this thesis, since there is no relevant case associated with the five projects studied.

2.1.2.5 Visibility

Reference to UML 1.5 specifications, Chapter 2, Part 2, Section 2.5 “Core” [31, pp. 35–36], the definitions of **feature** and **visibility** are given in the following:

Feature is defined as an **attribute** or **operation**, which is encapsulated within a classifier [31, p. 35].

Visibility is defined as specifying whether Feature can be seen and referenced by other classifiers [31, p. 36].

Four types of visibility and their denotations are illustrated as below:

- Public (denoted by the symbol ‘+’) - Any outside classifiers with visibility to classifier A can use the Feature of classifier A [31, p. 36].
- Protected (denoted by the symbol ‘#’) - Any descendent of the classifier A can use the Feature of classifier A [31, p. 36].
- Private (denoted by the symbol ‘-’) - Only the classifier A itself can use the Feature itself, or nested classifier B within classifier A can use the Feature of classifier A [31, p. 36].
- Package (denoted by symbol ‘~’) - Any classifier declared in the same package (or a nested subpackage, to any level) as the owner of the Feature can use the Feature [31, p. 36].

Osman *et. al* [35] proposed that software engineers prefer to leave *Private Operations* and *Protected Operations* out, to make a class diagram simplified. However, it has not been validated in a case, and to what extent they are left out is unknown. Thus, this thesis indeed wants to fill this gap. Considering the encapsulation of attributes in the OO paradigm, one inquiry is for simplifying a class diagram, whether the *public* set for the setter and getter operations of those private attributes are preferred to be excluded or even *public* for those operations and *private* for those encapsulated attributes are all excluded.

In another source [36], Osman *et. al* investigated that “counting the number of public operations” is the most important metric for indicating the importance of a class. This is from a reverse direction to analyze how to recover an important

class for a reverse-engineered class diagram; rather, from the forward direction, for (both important and secondary important) classes, whether the exclusion of the *public* set for operations relates to the types of operations is a question (based on the three types of operations defined in this thesis: constructors, setter or getter operations, and operations (besides setter and getter operations, which is detailed in section 4.3). Thus, those inquiries motivate this thesis to study the visibility in both attributes and operations.

2.1.2.6 Multiplicity

Multiplicity is defined as an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound [21, p. 95].

The textual notation for multiplicity specified by BNF is depicted in Figure 2.5. Only the multiplicity range is the focus since the other constituents of multiplicity are absent in the five mcAMs studied. Considering the Java specifications, the multiplicity in the mcAM can be either an array or a collection, and then the multiplicity range should be $[0, +\infty]$ [7].

Multiplicity is specified in the attributes and operations for three constituents that are related to data types. These three constituents are *attribute type*, *parameter type*, and *return type*. There are two reasons for focusing on the multiplicity of these three constituents. The first reason is that there might be differences in the interface types provided in the collection, e.g., *Set<name of the classifier>* in the mcAM changed into *List<name of the classifier>* in the cSC. The second reason is that the multiplicity not in the mcAM might be added to the cSC, e.g., *name of the classifier* in the mcAM is changed into *List<name of the classifier>* in the cSC. Multiplicity is a property of these three constituents and represents the number of instances of the classifier, so multiplicity along with the constituents, should be considered in parallel. On the other hand, the multiplicity and the relationships of *aggregation* and *composition* are correlated (referring to their corresponding definitions and semantics in subsubsection 2.1.2.7).

Note that the multiplicity placed at the end of an association is not the focus of this thesis, as it is not easy to manually check how many invocation sites of the instances in the cSC.

```

<multiplicity> ::= <multiplicity-range>
                [ [ '{' <order-designator> [ ',' <uniqueness-designator> ] '}' ] |
                  [ '{' <uniqueness-designator> [ ',' <order-designator> ] '}' ] ]
<multiplicity-range> ::= [ <lower> '..' ] <upper>
<lower> ::= <integer> | <value-specification>
<upper> ::= '*' | <value-specification>
<order-designator> ::= 'ordered' | 'unordered'
<uniqueness-designator> ::= 'unique' | 'nonunique'

```

Figure 2.5: Syntax for a multiplicity string [21, p. 98].

2.1.2.7 Graphical Notation for Relationships

The aim of this thesis is to study seven types of relationships of the mcAM: dependencies, usages, associations, aggregations, compositions, inheritances (also known as generalizations), and realizations. Reference to UML v1.5 specifications, Chapter 3, Part 5 [31, pp. 34–93], and UML v2.4.1 specifications [21], their definitions are given as follows (note that due to the unavailability of clear definitions of some relationships, their definitions should be better detailed with their semantics in order to enhance their comprehensibility):

Dependency is defined as a relationship that relates the model elements (constituents) themselves and does not require a set of instances for its meaning [31, p. 90]. Dependency signifies that a class requires another class for its specification or implementation [21, p. 61], so dependency indicates a situation in which a change to the target element may require a change to the source element in the dependency [31, p. 90].

Usage is defined as a relationship where one class requires another class for its full implementation or operation [21, p. 139].

Note that in the metamodel, Usage is a Dependency in which the client requires the presence of the supplier.

A **binary association** refers to an **association** among exactly two classes (including the possibility of an association from a class to itself) [31, p. 68].

Aggregation refers to a type of whole/part of a binary association relationship.

Composition refers to a strong form of aggregation that requires a part instance to be included in at most one composite at a time [21, p. 38]. If a composite is deleted, all of its parts are normally deleted with it [21, p. 38].

Inheritance refers to a taxonomic relationship between a more general superclass and a more specific subclass [21, p. 38]. Each instance of the subclass is also an indirect instance of the superclass [21, p. 70]. Thus, the subclass inherits the features of the superclass [21, p. 38].

Realization refers to a relationship between a class and an interface implying that the class supports the set of features owned by the interface and any of its parent interfaces [21, p. 89].

The graphical notation for the seven types of relationships defined above is depicted in Figure 2.6.

Of particular note, according to the definitions of the relationships of dependency, association, aggregation, and composition, four corresponding levels (from 1 to 4) of abstraction defined are depicted in Figure 2.7. The higher the level of abstraction, the lower the level.

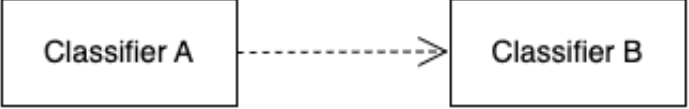
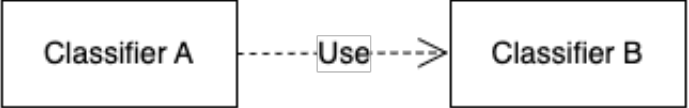
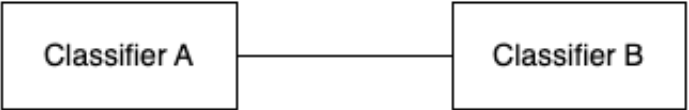
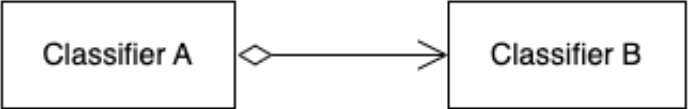
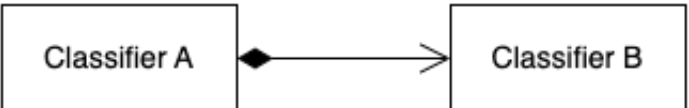

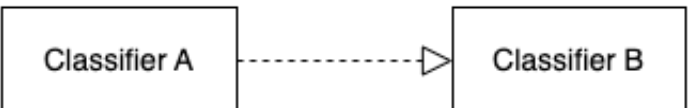
Relationship types	Graphical notation for relationships
Dependency	 <pre> graph LR A[Classifier A] -.-> B[Classifier B] </pre>
Usage	 <pre> graph LR A[Classifier A] -.-> Use B[Classifier B] </pre>
Association	 <pre> graph LR A[Classifier A] --- B[Classifier B] </pre>
Aggregation	 <pre> graph LR A[Classifier A] o--> B[Classifier B] </pre>
Composition	 <pre> graph LR A[Classifier A] *--> B[Classifier B] </pre>
Inheritance	 <pre> graph LR A[Classifier A] --> Extends B[Classifier B] </pre>
Realiazation	 <pre> graph LR A[Classifier A] -.-> > B[Classifier B] </pre>

Figure 2.6: Graphical notation for seven types of relationships.

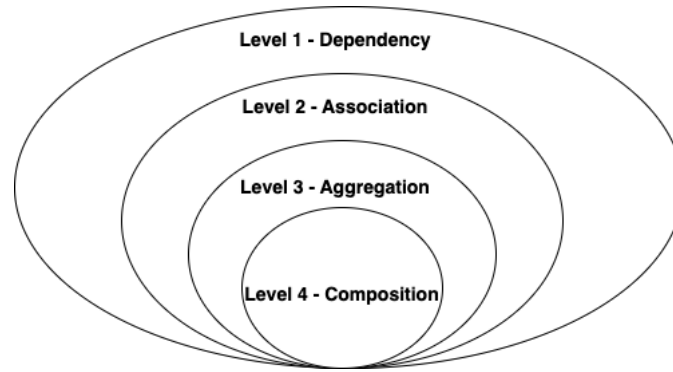


Figure 2.7: Four abstraction levels specified for four relationships - *dependency*, *association*, *aggregation*, and *composition*, respectively.

2.2 Related Work

This section consists of two parts. The first part is related to the currently existing reverse engineering methods/tools. The second part describes the usage of models in the software development practice.

2.2.1 Reverse Engineering

Müller *et al.* [37] refined the definition of reverse engineering in [38] as “a process of analyzing a subject system to identify its current components and their interrelationships and to extract and create system abstractions and design information.” Reverse engineering methods/tools play a key role in legacy systems based on the absence of a design. In particular, class diagrams are often poorly to be updated during development and maintenance [39, 13]. There is a concern that such a divergent class diagram cannot help software maintainers in the same way understand the system’s architecture during maintenance later on. Thereby, as a solution, reverse engineering methods/tools can be used to automatically generate reverse-engineered class diagrams that are extracted from the current code. Such a reverse-engineered class diagram can represent the up-to-date system’s architecture.

2.2.1.1 Manual Abstraction Created over Code

The class diagrams produced during the design and implementation phases of the SDLC can be referenced by software maintainers during the maintenance phase to understand the system’s architecture. However, in some cases, class diagrams may contain volumes of information [35]. This makes it hard for software maintainers to understand the system’s architecture [35]. Thereby understanding how abstraction is manually created by software engineers and thus condensing/simplifying class diagrams is essential. For this purpose, Osman *et al.* [35] conducted a survey to investigate how manual abstraction is created over code. This survey involves 32 software developers, with 75% of the participants having more than 5 years of experience with class diagrams [35]. As a result, they found the important elements in a class diagram are class relationships, meaningful class names, and class properties

[35]. Also, the information that should be excluded in a simplified class diagram is *GUI-related information*, *Private and Protected operations*, *Helper classes*, and *Library classes* [35]. However, these findings are needed to be validated in a case. The five case studies employed in this thesis can help validate these findings to some extent.

2.2.1.2 Solutions and Attempts to Provide Abstraction for Reverse-Engineered Class Diagrams

To make reverse-engineered diagrams both abstract and precise is a primary goal in the reverse engineering community. Thereby, concepts related to diagrams condensation/simplification or diagrams abstractness are first proposed along with several technologies developed later on aiming at archiving this goal.

Guéhéneux [5] proposed that both abstract and precise reverse engineering tools do not yet exist on the market. Guéhéneux [7] started by developing a tool named PTIDEJ aiming to produce precise reverse-engineered class diagrams, in particular, to infer *use*, *association*, *aggregation*, and *composition* relationships based on the consideration of lacking clear definitions of those relationships. PTIDEJ [7] performs even more accurately than class diagrams manually created by humans. Substantially, Guéhéneux [5] argued that the lack of abstraction with respect to current existing reverse engineering tools is because of the lack of clear definitions of class diagrams' constituents. Thereby, Guéhéneux systematically studied constituents of the class diagrams in reference to UML 1.5 specifications and refined their definitions. Guéhéneux [5] then exemplified the study with PTIDEJ to reverse Java programs as UML diagrams abstractly and precisely.

We agree with the assertion in [5] proposed by Guéhéneux, i.e., the definitions of some constituents of class diagrams are vague and verbose. Thus, the refinements of definitions in [5, 40] by Guéhéneux helped this thesis work a lot in regard to mappings between mcAM constituents and cSC constructs.

The lack of abstraction with respect to reverse engineering tools/methods is proposed by other authors as well, according to the source [35, 41, 39]. The resultant diagram generated by reverse-engineering methods/tools is often very cluttered [41]. This is of little help to software engineers in understanding the system's architecture since it is hard for them to locate the key places of primary interest.

Regarding condensation of reverse-engineered class diagrams to enhance their comprehensibility, Osman *et al.* [42] proposed an approach by using a supervised classification algorithm where design metrics (e.g., number of operations, number of attributes, etc.) as the input. Yet, an elemental question left out is which elements of the system's architecture should be selected for accommodating various levels of abstraction [39]. An extension of Osman *et al.*'s work is conducted by Thung *et al.* They used design metrics [42] created by Osman *et al.* and further added additional network metrics (e.g., betweenness centrality, closeness centrality, etc.) in their approach [41]. As a result, it reaches a 9% improvement compared to the approach

developed by Osman *et al.* Considering there is one concern of Thung *et al.*'s work, i.e., the generalization to other projects, Mersam and Peter [39] used semantic web technologies to improve the condensation process. Two main contributions of their work are V-OntModel, and a condensation architecture [39]. The former aims to help overcome the lack of missed information on software evolution, such as classes in a class diagram [39]. The latter aims to reach further a high level of abstraction of the class diagrams and gain comprehension [39]. The condensation architecture in which classes are ranked according to a decision list, then a final decision can be made on the candidate classes to be excluded from a reverse-engineered class diagram [39].

The findings in our thesis on the characteristics of the differences caused by manually creating various abstractions can provide input for improving the abstractness of future reverse engineering methods/tools. The reason for this is this thesis took the semantics information revealed behind the class diagrams into account during the mappings between mcAM constituents and cSC constructs.

2.2.1.3 Consistency Check(s) between Code and Design

Reverse-engineered diagrams are generated by extracting information from code based on the absence of a design that can be referenced. Rather, the consistency check(s) between code and design is based on the existence of an existing design. We agree with the assertion proposed by Antoniol *et al.* [32]. If the design exists, evolving it and then mapping it to code is a preference since the existing design includes context and high-level semantic information. However, a bias exists in reverse-engineered designs, multiple semantics are candidate explanations for the same piece of code [32], e.g., in Java, classifier A references B as an attribute type can represent either an aggregation or a composition between classifiers A and B. Thereby, evolving designs are considered of higher quality provided that traceability with code is maintained [32].

Various methods and models exist for checking the consistency between code and design. Still, Antoniol *et al.* [32] compared different design-code traceability methods based on different class properties (e.g., class names) and property combinations (e.g., the names of class operations and attributes prefixed with class names). They found that the methods based on *class names* perform best [32]. Dennis *et al.* [43] defined a matching between classifiers based on three approaches, i.e., *matching based on names*, *metric profile*, and *package information*, and their combination. They found that “matching based on names” performs best. It is not hard to find that Antoniol *et al.* and Dennis *et al.* all investigated that class names play the most important role in mappings between entities in design and constructs in code. However, for those classes that have not been successfully matched, there is a lack of systematic study on why deviations are between them (or their names). This thesis work will fill this gap by investigating the differences in class names and summarizing the reasons for these differences. In consequence, providing input for improving future methods/tools of consistency check(s) between code and design.

There is a lack of manual and systematic study into defining a set of mapping rules between code and design. The researchers mentioned above are inclined to use models and methods (sometimes leverage specific metrics) to directly investigate which class properties or combinations of properties are given more weight in the contribution to the consistency check between code and design. However, the tools and methods they used did not help them yield great gains in discovering the similarities and differences in class properties' interrelationships. Differences in the internal linkage of class properties are considered to potentially imply abstraction. Thus, the cases of the differences between the mcAM and cSC provided in this thesis can provide input for refining future mapping rules.

Differing from focusing on similarities and differences between the code and design, Shatnawi and Alzu'bi [44] focused on the deviations in three quality factors reusability, extendibility, and understandability. Thereby, they proposed a quality model named QMOOD [44] that uses object-oriented (OO) metrics to measure internal properties of the software (e.g., encapsulation, inheritance, and coupling) and external quality attributes (e.g., reusability, and understandability), aiming to find the correspondence between the design and implementation. As a result, inheritance, polymorphism, abstraction, composition, understandability, and extendibility are found to have the highest correspondence between code and design [44].

This thesis also argues that the internal properties mentioned above are noteworthy since they are built upon abstraction. The characteristics of abstraction are what this thesis aims to study.

2.2.2 Models in Software Development Practice

Existing studies on the usage of models in software development will be presented first. This will give the reader a context for the use of models. After that, some related work in creating the dataset/database that stores models will be presented. The created dataset/database can be used to answer a range of questions on model usage in FOSS. Considering that the models and code from the industry are often not accessible, the five studied subjects of this thesis are yielded from that dataset.

2.2.2.1 Usage of Models

Surprisingly, most studies so far on UML usage are based on the opinions and experiences of the participants rather than studying the actual models. Baltes and Diehl [45] conducted an online survey with 394 participants involved to investigate the use of sketches and diagrams in software engineering practice. They found that the majority of participants related their sketches to methods, classes, or packages but not to source code artifacts with a lower level of abstraction [45]. Yatani *et al.* [46] interviewed 9 Ubuntu contributors to investigate how and why they used diagrams. The study shows that the interviews used diagrams yet did not use them consistently [46]. The use and practices of diagramming are influential in FOSS development, yet one concern is that it is far from simple at times in regard to diagramming [46]. Chung *et al.* [47] questioned 230 contributors from 40 various FOSS projects and

interviewed eight participants in regard to sketching and drawing diagrams. They found the usage of sketching and drawings in the FOSS is infrequent.

We are convinced that the characteristics of manual abstraction provided by us would contribute to enabling software developers to gain insight into how to create a class diagram at an appropriate abstraction level to accommodate the required system's implementation structure, i.e., to avoid over-specification and under-specification in model elements. If a class diagram is created at an appropriate manual abstraction level, it will play a key role in helping software engineers understand and avoid misunderstanding of the system's implementation structure.

2.2.2.2 Dataset/Database of Models

It is hard to access industrial projects that include class diagrams and their corresponding source code. Thus, we decided to make use of models in FOSS. Karasneh *et al.* [48] adopted a crawling approach to acquire so far more than 700 UML class diagrams, and they converted them into xmi. stored in a searchable database afterward. Hebig *et al.* [1] mentioned that the lack of available data is the reason why so far, no answers could be given to several basic questions on the amount of UML files in open-source projects, such as whether the UML models are static or updated. Thus, they adopted a semi-automated approach to collect UML stored in images (.jpeg, .png, .gif, .svg, and .bmp) and standard formats (.xmi and .uml files) by randomly scanning ten percent of all GitHub projects (1.24 million) [1]. As a result, they gained a list of 3 295 open source projects, including 21 316 UML models. The Lindholmen dataset [1] is the first corpus established in the modeling community.

Compared to Hebig *et al.*'s work, in Karasneh *et al.*'s work, one concern is that the number of collected class diagrams is small. Yet, another concern is that no model context except for model artifacts is provided. This thesis aims to study the actual cases of models and code, so we need to have the corresponding source code of the model. The code information can be accessed via the links to the project provided in the Lindholmen dataset [1]. Thus, we decided to make use of the Lindholmen dataset [1] to obtain the eligible data.

3

Methodology

This chapter describes the methodology of multiple case studies employed in this thesis. Case study allows research to be conducted in a specific, natural setting and with low obtrusiveness to study a particular software engineering phenomenon [49]. Thereby, the case study is suitable for studying the characteristics of the differences between the mcAM and the cSC of that mcAM since no human obtrusiveness is involved. The overall process of the methodology is illustrated in Figure 3.1.

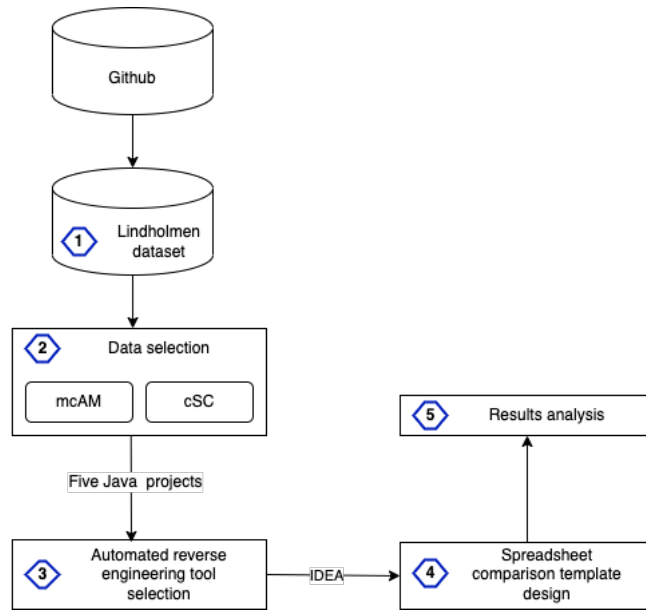


Figure 3.1: The overall process of the methodology.

First, we accessed Lindholmen dataset [1] (Step 1). This dataset is a collection of open-source projects from Github [1]. These projects include UML files with their corresponding source code. The information of Lindholmen dataset [1] is detailed in section 3.1.

We selected five Java non-academic projects from that dataset as the study subjects (Step 2). For each project, there might be one or more mcAMs included in the project, yet only one of them will be selected as the study subject. Also, there might be multiple cSCs of a selected mcAM in the project's repository. Still, only one cSC of that mcAM will be selected. The ideal selection of one mcAM and one cSC of that mcAM from a project is illustrated in section 3.2.

In Step 3, we selected IntelliJ IDEA (IDEA) as our reverse engineering tool for reversing the source code into a class diagram. The reason we selected this tool is detailed in 3.3.

A spreadsheet comparison template is designed (Step 4). This template is used to record the information on the project, the information of the selected mcAM and the cSC of that mcAM, as well as the differences between the mcAM and the cSC of that mcAM. The meaning of each column that forms this comparison template is detailed in section 3.4, followed by how we turned mcAM and the cSC of that mcAM into pictures for differentiating.

With the support of IDEA (selected from Step 3) and the comparison template (designed in Step 4), it allows differences between the mcAM and the cSC of that mcAM to be easily identified and thus be recorded in the comparison template. Afterward, those recorded differences are analyzed to answer the RQs (Step 5).

3.1 Open Source Repositories Access

The Lindholmen dataset [1] used in this thesis is a large modeling corpus, and it was derived from mining GitHub’s open-source repositories. Hebig et al. [1] collected UML stored in images formats (.jpeg, .png, .gif, .svg and .bmp) and standard formats (.xmi, and .uml files) by randomly scanning ten percent of all GitHub projects (1.24 million). After scanning and collecting data, they got this dataset, a list of 3 295 open source projects which include together 21 316 UML models [1]. The collecting process is without specific programming language restrictions and domain type restrictions. Thus, this dataset can support this thesis work for selecting five Java projects with generalization in a highly natural setting. That means those selected projects can represent the practices in the real world. Furthermore, the Lindholmen dataset [1] comprises two csv files, i.e., UMLFiles_List and Project_FileTypes. Only the former is accessed because this file includes a list of links of UML files to all projects [1]. These links allow the UML files to be accessed directly.

3.2 Data Selection

Due to the time constraints of this thesis work, a pragmatic scope is necessary. Thus, five Java projects would be feasible for us to study. As aforementioned, these five projects are selected from the Lindholmen dataset [1]. The criteria for choosing these five projects are: First, they should not come from academia, as they are not real practices in the industry, which is a threat to the validity that will be illustrated in section 5.6. Second, the five mcAMs selected from those five projects should be in image format with the existence of their corresponding cSC(s). Selecting class diagrams in image format can save us time and allow us to acquire information on the class diagrams directly. This means that an additional step is required compared to other .xmi and .uml files, which is to use an automatic reverse engineering tool

to process the files and generate the corresponding class diagrams in image format.

3.2.1 Definition of Terminologies

The following terminologies are defined by us or with reference to the definitions given by other authors, which allows the reader to understand the ideal selection of one mcAM from a project and the ideal selection of one cSC among multiple cSCs of that mcAM in this section.

Reference to the source [50], **manual abstraction (MA)** is defined as using high-level elements in the mcAM containing fewer details than the requirement for the implementation to represent the implementation structure of the system.

Disagreements (disAGTs) refer to developers' deviations from the design decisions made by architects in the mcAM.

Common changes (CC) refer to the differences that are not caused by disAGTs and MA between the mcAM and the cSC of that mcAM.

Note that for the definitions of the three terminologies given above, a sorted list of cases corresponding to them is provided in Chapter 4. These cases will enhance the reader's comprehension of those definitions.

3.2.2 Ideal Selection of One mcAM from a Project

As aforementioned in section 3.2, the selected mcAM should first be in image format. Second, the mcAM should have the corresponding cSC that can be found in the repository. If only one mcAM is included in a project, we select that mcAM. If there are multiple mcAMs that are included in a project, considering the time constraints and the fact that we want to study more projects, we would ideally want to select one of them to study. The following two cases need to consider:

Case 1 - A mcAM with one or more updates: As the code evolves, a mcAM is sometimes updated one or more times. However, we would ideally want to select the latest update of the mcAM as the study subject. We cannot guarantee that compared with other legacy mcAMs, the latest updated mcAM is the most accurate. This is only about a decision that needs to make in the case that the mcAM with one or more updates, and we expect to select one only to study. This is because we do not expect to study a legacy mcAM that cannot reflect the up-to-date implementation structure of the system. One concern is that the voSC that includes the latest mcAM might not be fully implemented, so accordingly, there is no map for some mcAM concepts. Nevertheless, this concern can be dealt with by selecting one ideal cSC of the latest updated mcAM (proposed in section 3.2.3).

Case 2 - Multiple mcAMs are created to represent a system, and these mcAMs might represent different parts of the system: If it is the case, we would ideally want

to select one of them to study randomly. It is difficult to set criteria for an ideal mcAM we need to select since the number of concepts or attributes and operations associated with the concepts is independent of the complexity of the system, i.e. the more concepts or more attributes and operations created in a mcAM does not imply that the system it represents is more complicated. Thus, randomly selecting one mcAM will not introduce any bias.

3.2.3 Ideal Selection of One cSC among Multiple cSCs of that mcAM

As mentioned in section 1.5.2, if multiple cSCs correspond to one mcAM, only one cSC of them will be selected. Also, this cSC should ideally cover most attributes and operations of the mcAM compared to other cSCs; otherwise, the missed attributes and operations will result in overestimating the differences caused by MA and disAGTs in the results. This is a threat to the validity, which will be described in section 5.6. Considering time constraints, assuming that a project has hundreds or thousands of voSCs, it is not feasible for us to check all voSCs one by one to know which of them are cSCs of the mcAM and, further which cSC is the best selection. Thus, it cannot be guaranteed that the finally selected cSC is without any missed attributes and operations which are implemented in other cSCs. However, we tried to select a cSC that is approximately close to an ideal cSC. In order to shorten the time spent on selecting an ideal cSC and study more projects, a process is proposed (see Figure 3.2). Note that regardless of when the selected mcAM was created in the SDLC, the voSC that includes the selected mcAM is always the first selection. This voSC is considered likely to be a cSC of the selected mcAM since they were created at the same time.

Figure 3.2 illustrates a case that the selected mcAM is created based on the existing source code as the code evolves. We first check every concept in the mcAM to know whether there is a map to that voSC. If so, this voSC is a cSC of the mcAM. However, considering that a mcAM may have multiple cSCs, to prevent the overestimation of the differences caused by MA and disAGTs, we would want to map the voSC before and after that voSC to the mcAM. As long as an ideal cSC that covers most attributes and operations is found, we will stop the mapping. Note that if there are several ideal cSCs, i.e., cSCs that cover the most attributes and operations in the mcAM compared to other cSCs, we will select one of them randomly.

If for one or more concepts in the mcAM, there is no map to that voSC, that means that voSC is not a cSC of the mcAM. This situation has not been encountered in our studies. Nevertheless, we proposed a process in the dashed line (as shown in Figure 3.2) to accommodate this situation. This process aims to help find an ideal cSC. We also proposed the following two reasons, with a respective case assumed as well, to explain this situation.

Reason 1: For the voSC before that voSC, there might be one or more voSC that are the cSC(s) of the mcAM, yet as the code evolves, the cSC(s) has been changed

several times, which leads to the fact that voSC is no longer a cSC of the mcAM.

Reason 2: For the voSC after that voSC, there might be one or more voSC that are the cSC(s) of the mcAM. That implies some development work may not be done yet, so as the code evolves, one or more concepts in the mcAM will be implemented in the following voSC.

It is difficult to distinguish between both cases by looking at one voSC. However, this issue can be dealt with by comparing the number of classes in the voSC with the number of concepts in the mcAM. If the number of classes in that voSC is much more than the number of concepts in the mcAM, yet for some of the concepts in the mcAM, there is no map to that voSC. This case will be assumed to be caused by reason 1. If the concepts that have a map in that voSC are less than the concepts in the mcAM, i.e., too little source code information about the classes that we can map to, for the concepts in that voSC, this case will be assumed to be caused by reason 2. For both cases, it requires us to continuously check the voSC before that voSC in reverse order and continuously check the voSC after that voSC, respectively, until an ideal cSC is selected. That means the selected cSC reaches a peak where it covers most attributes and operations (before the next cSC covers fewer attributes and operations compared to it, or the next voSC is not a cSC, i.e., for some concepts in the mcA, there is no map to that voSC).

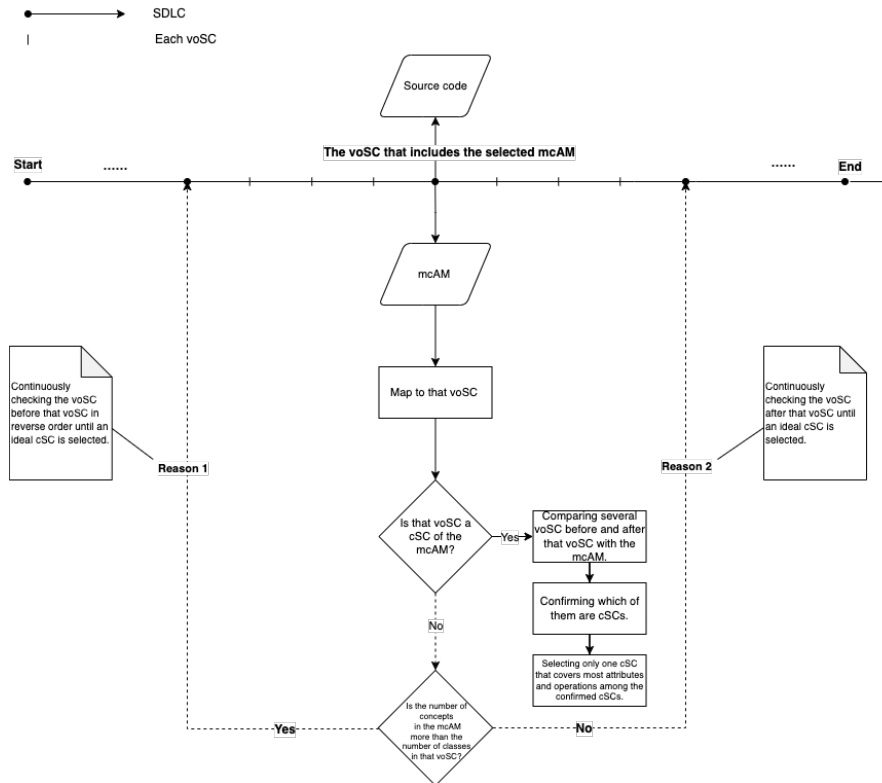


Figure 3.2: The selection process of one ideal cSC among multiple cSCs of the selected mcAM.

Besides the case illustrated in Figure 3.2, another two cases we need to consider: **1.** The selected mcAM may have been created at the start of the project with or without the corresponding source code. For this case, we need to continuously check the voSC after that voSC until an ideal cSC is selected. **2.** The selected mcAM may have been created at the end of the project. For this case, we need to continuously check the voSC before that voSC in reverse order until an ideal cSC is selected. The continuous check is the same way as in the two cases mentioned above.

3.2.3.1 Time and Resource Consumption

One question that might arise in the ideal selection of one cSC among multiple cSCs of the selected mcAM is the time and resource consumption. For each of the five projects studied, the voSC that includes the selected mcAM is all a cSC of that selected mcAM. In regard to that voSC/cSC of a project, the time taken highly depends on the coverage of attributes and operations in it. If fewer attributes and operations are covered by it, the more left uncovered attributes and operations that we need to check for creating the mappings between the voSCs and mcAM and vice versa.

As observed from the five projects studied, for three of them, the finally selected ideal cSC is the voSC/cSC that includes the selected mcAM. Thus, the time for selecting that ideal cSC would be possibly around half a day. For the other two projects, the time span between the voSC that includes the selected and the finally selected cSC in terms of creating is less than ten days. The time would be taken around one day. In regard to the “continuous” checking process, there might be some voSCs that are irrelevant to the mcAM. Thus, they can be quick to be skipped. The voSC(s) related to the mcAM are the concerns. Once the changes in a voSC that is related to mcAM were found, it was unnecessary for us to download those voSCs from the repository; rather, we online checked the related classes’ source code files in the repository to locate the changes, respectively. Thus, the storage resource is not consumed much, and only some cache will be consumed. Yet, it is essential to take related digital or hands notes in regard to the “continuous” checking process, e.g., commit A covers one more attribute *a* than commit B and with recording their corresponding creation time. Then, the timeline at least would be clearly known by us. Note that if the timeline is constrained to very few days, yet there is still a possibility that a huge amount of voSCs involved in it. It is hard to know how many of them are related to the mcAM since some of them might be used to describe the other parts of the system not modeled by the selected mcAM. Thus, we can only illustrate the issue of time consumption at a descriptive level.

Note that we did not encounter a situation where the voSC that includes the selected mcAM is not a cSC for that mcAM. Thus, the approximate time expected to be taken in that situation is unknown.

3.3 Automatic Reverse Engineering Tool Selection

To simplify the cSC selection process, as an aid, the automatic reverse engineering tool can mimic humans via automatically abstracting classes. Thereby the classes can be abstracted over the detailed code implementation and laid out in the reverse-engineered class diagram. Then the mappings between the classes from mcAM and voSC are easily created, and the cSC(s) can be confirmed afterward. Thereby, the only requirement for selecting an automatic reverse engineering tool is high preciseness in capturing the classes.

We compared Enterprise Architect (EA) and IntelliJ IDEA (IDEA). They can both capture the classes precisely. Considering our proficiency in using the tool, IDEA was chosen as our automatic reverse engineering tool. Notably, we always used the source code as a benchmark to identify the differences between mcAM and cSC, especially considering the fact that existing tools cannot fully and correctly identify the relationships in the source code. Another note is that the selection of an ideal cSC is semi-manual based on the fact that we only selected the related classes' source files to be visualized by the selected tool.

3.4 Spreadsheet Comparison Template Design

The scope of this thesis is to investigate five projects, so the results of these five projects are needed to be aggregated. Thereby, a spreadsheet comparison template is needed to be created. The comparison template designed consists of 29 columns that are used to record 5 types of information in order: (i) Information about the project. (ii) Information on the mcAM. (iii) Information on the cSC. (iv) The differences between the mcAM and the cSC of that mcAM. (v) Additional notes. Considering the limited space on the page, the comparison template is designed to be too long to be clearly visible. Thus, it will be divided into five parts according to those five types of information. Each part will be explained separately, with corresponding explanations.

Figure 3.3 shows that the columns from A to G (with a grey background) are used to record information (i). This allows us to have a general understanding of the background of the project. An explanation of each column is as follows:

- **A - Repo. Name and Link:** The name of the project's repository and the link to the repository. This link helps us to easily access the repository as soon as we need to get some information about the project.
- **B - SDLC:** The time span is from the first commit creation time in the repository to the last commit creation time.
- **C - No. voSC:** The number of voSC found in the repository.

- **D - No. Contributors:** The number of contributors who contributed to the repository.
- **E - Lang.:** Lang. refers to languages.

Note that this thesis focuses only on Java programming projects.

- **F - Arch. Pattern:** Arch. pattern refers to the architectural pattern used in the project.
- **G - Type:** There are two types of projects: industrial and uncertain. If the information available in the repository cannot help us to confirm whether a project is from the industry, then it will be defined as an uncertain project.

Note that here the available information that can be looked up refers to the following:

- Profile of the contributors, e.g., work information, which can imply whether a contributor is an employer at a company or an organization or a student when they build that repository.
- The documents found in the repository document the information containing the relevant information with respect to the types of the project.

A	B	C	D	E	F	G
Repo. Name and Link	SDLC	No. voSC	No. Contributors	Lang.	Arch. Pattern	Type

Figure 3.3: Record information about the project.

Figure 3.4 shows that the columns from H to N (with a green background) are used to record information (ii). An explanation of each column is given below:

- **H - ID of the voSC that Includes the mcAM:** ID refers to a unique 7-length identifier of a commit, which can be found in the commit history.
- **I - mcAM Created Date:** The creation date of the mcAM can be found in the repository, and it should be in DD-MM-YYYY format.
- **J - mcAM Link:** The link to the mcAM. That allows us to track the mcAM easily via this link.

- **K - No. Concepts:** The number of concepts created in the mcAM.
- **L - No. Attrib.:** The number of attributes that are associated with the concepts in the mcAM.
- **M - No. Ops.:** The number of attributes that are associated with the concepts in the mcAM.
- **N - No. Relatsh:** The number of relationships between the concepts in the mcAM.

H	I	J	K	L	M	N
ID of the voSC that includes the mcAM	mcAM Created Date	mcAM Link	No. Concepts	No. Attrib.	No. Ops.	No. Relatsh.

Figure 3.4: Record information of the mcAM.

Figure 3.5 shows that the columns from O to X (with a blue background) are used to record information (iii). An explanation of each column is given below:

- **O - cSC ID:** ID refers to a unique 7-length identifier of a commit, which can be found in the commit history.
- **P - cSC Created Date:** The creation date of the cSC that can be found in the repository, and it should be in DD-MM-YYYY format.
- **Q - cSC Link:** The link to the cSC. That allows us to track the source code easily via this link.

R - No. Cls.: The number of classes in the cSC.

Note that classes here refer to classes associated with mappings from one or more concepts in the mcAM to the cSC.

- **S - No. Attrib.:** The number of attributes that are associated with the classes (counted in column R) in the cSC.
- **T - No. Ops.:** The number of operations that are associated with the classes (counted in column R) in the cSC.

- **U - No. Relatsh.:** The number of relationships that are associated with the classes (counted in column R) in the cSC.
- **V - All Attrib. Covered? (0/1):** To check whether all attributes created in the mcAM are covered by the cSC (0 = No, 1 = Yes).
- **W - All Ops. Covered? (0/1):** To check whether all operations created in the mcAM are covered by the cSC (0 = No, 1 = Yes).
- **X - All Relatsh. Covered? (0/1):** To check whether all relationships created in the mcAM are covered by the cSC (0 = No, 1 = Yes).

O	P	Q	R	S	T	U	V	W	X
cSC ID	cSC Created Date	cSC Link	No. Cls	No. Attrib.	No. Ops.	No. Relatsh.	All Attrib. Covered? (0/1)	All Ops Covered? (0/1)	All Relatsh. Covered? (0/1)

Figure 3.5: Record information of the cSC.

Figure 3.6 shows that the columns from Y to AA (with a dark green background) are used to record information (iv). An explanation of each column is given below:

- **Y - Diff. Caused by MA:** To record the cases of the differences caused by MA.
- **Z - Diff. Caused by disAGTs:** To record the cases of the differences caused by disAGTs.
- **AA - Diff. Caused by CC:** To record the cases of the differences caused by CC.

Y	Z	AA
Diff. Caused by MA	Diff. Caused by disAGTs	Diff. Caused by CC

Figure 3.6: Record the differences between the mcAM and the cSC of that mcAM.

Figure 3.7 shows that column AB is used to record information (v). This column helps us to record additional information that cannot be filled into the columns from

A to AA.

AB
Notes(Optional)

Figure 3.7: Record additional notes.

3.4.1 Turning mcAM and cSC into Pictures for Differentiating

The comparison template designed in section 3.4 can assist us in recording the differences in texts between the mcAM elements and cSC constructs. Yet, for some of the specific differences, we will use the related partial pictures turned by the mcAM and reverse-engineered class diagrams generated by the automatic reverse engineering tool (IDEA) to illustrate. This can ease the reader's understanding of these differences and makes it easier for us to give examples to illustrate them. This is because the huge deviations in the cSC implementation from the mcAM design cause some differences between them. The semantics conveyed by mcAM elements are thereby implemented by complicated cSC constructs. Some of the complicated cSC implementations are better illustrated by using the related partially generated reverse-engineered class diagrams with detailed explanations.

For detecting the differences between the mcAM and cSC, we initially intended to use an automatic reverse engineering tool to help us achieve that. Yet, after trying to use the tool for a long time, we found that the tool's usage (not limited to IDEA we adopted yet for another tool EA we tried out) is very limited. The tools can only help us detect the structural differences, and any other non-structural differences that need to take the semantics into account cannot be revealed by a reverse-engineered class diagram generated by the tool. In consequence, we decided to always use the source code as a baseline for detecting the differences between the mcAM and cSC. Thus, the selected tool IDEA is only considered an aid for us to give partial screenshots (pictures) from the generated reverse-engineered class diagrams to illustrate the examples of the given differences.

In the comparison, we first manually selected the related classes' source code files to be visualized by IDEA. That means the mapping from classes between the mcAM and cSC is fully completed by humans. Then we manually create one-to-one mappings from the mcAM elements to the cSC constructs in terms of attributes and operations. For any non-conformance between the mcAM and cSC, in terms of attributes and operations, or any of them from the mcAM found to be missed in the

reverse-engineered class diagrams, we will check the relevant detailed cSC implementation to fully understand the roles of related classes, functionalities of attributes, and operations associated with these classes. Then the one-to-many and many-to-many manual mappings from the mcAM to the cSC can hereby be created by jointly taking the semantics of the mcAM attributes, operations, and relationships into account. Finally, the differences related to semantics can thereby be detected. For studying the differences between the relationships, given the fact that relationships are highly dependent on the design of the attributes and operations, understanding the detailed cSC implementation is a must here. We never used the tool IDEA as an aid for helping us to detect the differences in relationships since IDEA cannot even correctly recognize the relationships that exist in the cSC implementation and then visualize them therein.

To illustrate how we turn mcAM and the reverse-engineered class diagrams generated by IDEA into the corresponding partial screenshots/pictures to differentiate them, we give the following example, taken from one of our study subjects, i.e., EAPLI_PL_2NB [19] on GitHub:

As mentioned previously, for every class in the mcAM, in terms of its attributes and operations, we first manually create respective one-to-one mappings from the mcAM to the cSC. Thus, in this example, for the class *CheckingAccount*, we checked its included elements, e.g., attributes, one by one, e.g., *income: Income* (see Figure 3.8). It is observed that this attribute from the mcAM is missed in the reverse-engineered class diagrams generated by IDEA (see Figure 3.9). Note that we purely named such a reverse-engineered class diagram generated by IDEA over the detailed cSC implementation as cSC in our illustration. Considering that the linkage between the classes *CheckingAccount* and *IncomeRepository* from the mcAM remains in the cSC, built on the fact that an attribute type of an instance of *IncomeRepository* is specified in *CheckingAccount*. We can indeed infer that missed attribute is replaced with new attribute *incomeRepo: IncomeRepository* (not in the mcAM yet added to the cSC).

Note that we are inclined to turn the pictures from the mcAM and cSC with respect to the two classes *CheckingAccount* and *IncomeRepository* rather than only one class *CheckingAccount* to illustrate the fact that the attribute *income: Income* of *CheckingAccount* is replaced with new attribute *incomeRepo: IncomeRepository* in the cSC. This is because we jointly took the semantics conveyed by attribute *income: Income* in *CheckingAccount* and the linkage between *CheckingAccount* and *IncomeRepository* in the mcAM into account.

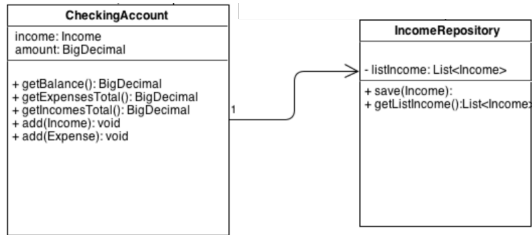


Figure 3.8: For the class *CheckingAccount*, the attributes, e.g., *income: Income* is modeled in the mcAM.

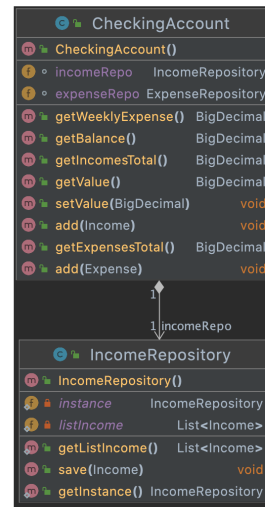


Figure 3.9: For the class *CheckingAccount*, that attribute *income: Income* from the mcAM is replaced with fully new attributes *incomeRepo: IncomeRepository* in the cSC.

To enable the reader to know how we recorded such a difference illustrated above in our design comparison template, we gave the following Figure 3.10 to illustrate. It is observed that we first recorded the class name which we were checking, i.e., *CheckingAccount*. Then we recorded the attribute *income: Income* from the mcAM is missed in the cSC, followed by that it is replaced by a new attribute *incomeRepo: IncomeRepository* in the cSC.

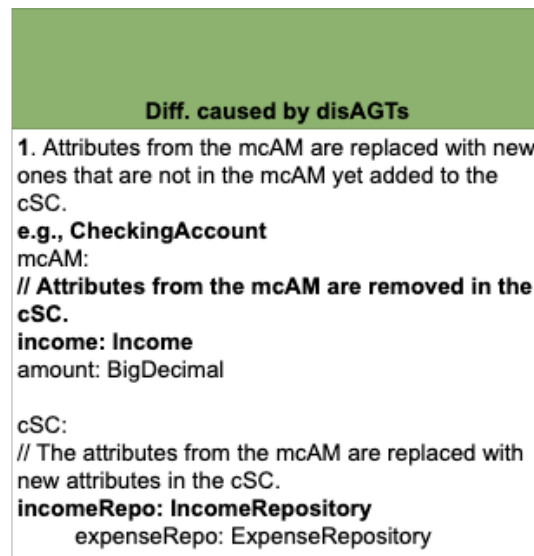


Figure 3.10: We recorded the difference, i.e., attributes from the mcAM are replaced with new attributes not in the mcAM yet added to the cSC, differentiated into disAGTs in our designed comparison template.

4

Results

The results of this thesis work are presented in this chapter, aiming to answer the RQs in section 1.6. Information about the project manually studied is presented first, allowing the reader to understand the project’s background. After that, three primary causes that cause the differences between the mcAM and cSC, namely MA, disAGTs, and CC, are concluded. A sorted list of their cases is presented as well. Regarding the cases of the differences caused by MA and disAGTs, not only do they have their own cases, but some of them are opposed to each other. An explanation for each of these cases is given, followed by a corresponding example to illustrate.

4.1 Information on the Project Studied

With the help of the template designed in section 3.4, the information on the five projects studied is presented in Table 4.1. Note that columns from “Repo. name and link” to “Type” have the same meaning as the corresponding columns from A to G in that template explained in section 3.4.

Table 4.1: The project background (\sim = around).

ID	Repo. name and link	SDLC	No. cSC	No. contributors	Lang.	Arch. pattern	Type
1	ZooTypers [15]	\sim 1.5 months	745	5	Java	MVVM	Uncertain
2	RaiseMeUp [16]	\sim 1.5 months	24	2	Java	MVC	Uncertain
3	EAPLI_PL_2NB [19]	\sim 2 months	483	9	Java	MVVM	Industrial
4	FreeDaysIntern [51]	\sim 30.5 months	449	3	Java	Uncertain	Uncertain
5	NeurophChanges [18]	\sim 28 months	534	7	Java	Spring MVC	Industrial

4.2 Suspected Cases

Table 4.2 illustrates the cases that we suspected might exist in other voSC(s)/cSC(s), caused by MA and disAGTs. This is based on the fact that we only selected one cSC among multiple cSC(s) for a project, and those suspected cases might be hidden in other voSC(s)/cSC(s) not selected by us previously. On the other hand, considering some cases of MA and disAGTs are opposed to each other since the design decisions made on the mcAM design are unfollowed and different decisions are taken in the cSC implementation. We can infer the opposite from the case we observed. Those cases are able to be covered in our future work or might provide some inspiration

for others who want to study those cases.

Referring to Table 4.3, case MC-2, considering the specificity of applying design patterns in projects, a design pattern is not limited to only one way that can be adopted for its implementation. Therefore, it is possible that the over-specification of the classes related to design patterns in a mcAM is not followed; instead, another way is taken to address the same concern that the design pattern has in the cSC implementation. Those over-specified classes from the mcAM are possibly condensed in the cSC implementation. Thus, case **SDC-1** can hereby be inferred.

Considering the possibility of converting the constant variables from the mcAM to variables in the cSC, the default values assigned to the constant variables from the mcAM might be removed accordingly in the cSC. To be specific, assigning default values to variables might be less common during the mcAM design, as they are very likely to be updated frequently in the cSC implementation to cater to the new requirements involved. However, if a constant variable is modeled out with a specified default value in the mcAM, it is possible that the architects want to emphasize the importance of this value. Thereby, case **SDA-1** is inferred by us.

Out of the consideration of the code reuse and *inheritance* concept of the OO paradigm, in reference to Table 4.3, case MA-5 was observed by us in the past. Yet, we still cannot exclude the possibility that one or more attributes in the super-class are downshifted to its subclasses in the hierarchical chain (i.e., case **SDA-2**). It is considered not to be an ideal decision made in the code implementation, though.

With reference to Table 4.3, case MA-6, the opposite to it might be because the attributes are over-specified in the mcAM. Yet, those over-specified attributes are redesigned as one condensed attribute during the cSC implementation (i.e., case **SDA-3**). The same as to case **SDO-1**, the over-specified operations from the mcAM are redesigned into one operation in the cSC.

Referring to Table 4.3 cases MR-2 and MR-3, an aggregation or a composition from the cSC can be modeled as an association in the mcAM to show the linkage between different classifiers (or their instances). However, compared with an association, there is a constrain on an aggregation and a composition - that must be built on the fact that B (part) can be accessed through a field of A (whole) [40], i.e., an attribute specified in A, where instances of B are referenced. Thereby, a more explicit restriction on the relationships is a deviation of the cSC from the mcAM. This can lead to cases **SDR-1** and **SDR-2**.

Considering the abstraction level of an association is higher than an aggregation and a composition, as a basis, we can infer case **SMR-1** that might exist in other voSCs/cSC(s), i.e., a composition from the cSC is modeled as an aggregation. This case indeed covers manual abstraction.

Table 4.2: Suspected cases caused by MA and disAGTs potentially exist in other voSC(s)/cSC(s) not selected by us previously.

Case ID	Suspected cases for MA	Case ID	Suspected cases for disAGTs
Classes		SDC-1	Multiple classes are created in the mcAM, yet only one class is implemented in the cSC.
Attributes		SDA-1	The default value in the mcAM is removed in the cSC.
		SDA-2	One or more attributes in the hierarchical structure in the mcAM, i.e., the superclass, are downshifted to one or more subclasses inherited from this superclass in the cSC.
		SDA-3	Multiple attributes are created in the mcAM, yet only one attribute is implemented in the cSC.
Operations		SDO-1	Multiple operations are created in the mcAM, yet only one operation is implemented in the cSC.
Relationships (between classifiers A and B)		SDR-1	An aggregation between A (whole) and B (part) from the mcAM is changed into an association between A (origin) and B (target) in the cSC.
		SDR-2	A composition between A (whole) and B (part) from the mcAM is changed into an association between A (origin) and B (target) in the cSC.
	SMR-1		A composition between A (whole) and B (part) from the cSC is modeled as an aggregation in the mcAM.

4.3 Cases of the Differences Caused by MA and disAGTs

Table 4.3 presents a sorted list of cases caused by MA and disAGTs. Here the model elements of classes, attributes, operations, and relationships are the focus. Some of the cases caused by MA and disAGTs are opposite to each other. The reason for this is that the architects might sometimes over-specify some parts of the system, yet the developers disagree with those design decisions, and they make different decisions when developing the system. An example of each of these cases is also given (note that a case might exist in one or more projects and might have one or more examples in a project, yet we only select one example from a project as the representation to illustrate that case).

Another note is that the naming of the classes, attributes, and operations between the mcAM and cSC might be different. This might be involved in the presented examples. However, such naming differences do not affect the fact that they are the same class, attribute, and operation. This thesis considers the naming differences to be caused by CC, which will be elaborated in section 4.4.

4. Results

Table 4.3: Cases for the differences caused by MA and disAGTs (* = own case, <> = opposite).

	Case ID	MA		Case ID	disAGTs
Classes	MC-1	Hierarchical inheritance structures not created in the mcAM are added to the cSC	<>	DC-1	Hierarchical inheritance structures in the mcAM are removed in the cSC
	MC-2	One class created in the mcAM is divided into more than one class in the cSC (related to the specific design patterns)	<>		Referring to Table 4.2, case SDC-1
Attributes	MA-1	Additional attributes not in the mcAM are added to the cSC	<>	DA-1	Attributes in the mcAM are removed in the cSC
				DA-2*	Attributes from the mcAM are replaced by additional attributes (that are not in the mcAM added to the cSC) in the cSC
	MA-2*	An attribute of classifier A whose type can indicate an aggregation or a composition between classifiers A and B from the cSC is modeled out by the naming of the association between them in the mcAM			
	MA-3*	The additional attribute type not in the mcAM is added to the cSC			
				DA-3*	The attribute type in the mcAM and cSC is different (two causes, see Figure 4.40)
	MA-4	The additional default value not in the mcAM is added to the cSC	<>		Referring to Table 4.2, case SDA-1
				DA-4*	Converting variables from the mcAM to constant variables in the cSC
				DA-5*	The default value not in the mcAM is added to the cSC (caused by case DA-4*)
	MA-5	One or more common attributes in one or more subclasses in the mcAM are upshifted to the hierarchical structure in the cSC, i.e., the superclass inherited by these subclasses	<>		Referring to Table 4.2, case SDA-2
	MA-6	One attribute created from the mcAM is divided into more than one attribute in the cSC	<>		Referring to Table 4.2, case SDA-3
Operations	MO-1	Additional operations not in the mcAM are added to the cSC (three subcases, see Figure 4.7)	<>	DO-1	Operations in the mcAM are removed in the cSC
	MO-2	Additional parameter names not in the mcAM are added to the cSC	<>	DO-2	The parameter names in the mcAM are removed in the cSC
	MO-3	Additional parameter types not in the mcAM are added to the cSC	<>	DO-3	The parameter types in the mcAM are removed in the cSC
				DO-4*	The parameter type in the mcAM and cSC is different (caused by cause 2 of DA-3*)
	MO-4	Additional return types not in the mcAM are added to the cSC	<>	DO-5	The return types in the mcAM are removed and as void in the cSC
	MO-5*	Multiplicity (referring to the collection-related interfaces) specified for the return type without its corresponding implementation interfaces/classes specified in the mcAM, yet with them specified in the cSC			
	MO-6*	The default parameters of the operation (in Android) in the cSC are omitted in the mcAM			
	MO-7	One operation created in the mcAM is divided into multiple operations in the cSC	<>		Referring to Table 4.2, case SDO-1
Relationships (between classifiers A and B)				DO-6*	One or more operations from classifier A in the mcAM are moved to classifier B in the cSC (related to the particular architectural patterns)
	MR-1	Additional relationships not in the mcAM are added to the cSC (two subcases with their six and two corresponding concluded causes, respectively, see Figure 4.12)	<>	DR-1	Relationships between A and B from the mcAM are removed in the cSC (one cause, see Figure 4.45)
	MR-2	An aggregation between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (two causes, see Figure 4.13)	<>		Referring to Table 4.2, case SDR-1
	MR-3	A composition between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (one cause, see Figure 4.13)	<>		Referring to Table 4.2, case SDR-2
		Referring to Table 4.2, case SMR-1	<>	DR-2	A composition between A (whole) and B (part) from the mcAM is changed into an aggregation in the cSC (one cause, see Figure 4.45)
				DR-3*	Relationships between A and B from the mcAM are replaced by new relationships in the cSC (two causes, see Figure 4.45)

4.3.1 Cases of MA - Classes

4.3.1.1 Case MC-1 - Hierarchical inheritance structure not created in the mcAM is added to the cSC

Explanation of case MC-1: This case covers manual abstraction that hides one or more subclasses derived from one superclass implemented in the cSC in the mcAM. This also means the superclass inherited by these subclasses implemented in the cSC can be modeled as a class in the mcAM.

Example of case MC-1 (selected from project 5): As observed in the comparison between Figures 4.1 and 4.2, the subclasses, e.g., *Max* derived from the superclass *inputFunction* implemented in the cSC are hidden in the mcAM.

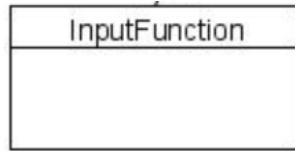


Figure 4.1: The subclasses, e.g., *Max* implemented in the cSC are hidden in the mcAM. Also, the superclass *InputFunction* inherited by these subclasses in the cSC is modeled as a class in the mcAM.

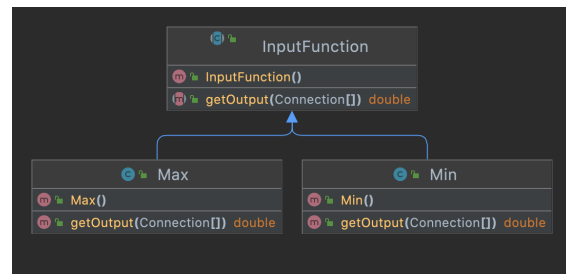


Figure 4.2: A hierarchical inheritance structure with the corresponding additional two subclasses not in the mcAM is added to the cSC.

4.3.1.2 Case MC-2 - One class created in the mcAM is divided into more than one class in the cSC (related to the specific design patterns)

Explanation of case MC-2: In the mcAM, a particular design pattern can be implied by the naming of a class and the relationships between that class and another one or more classes. Yet, one concern is that the detailed implementation of that design pattern from the cSC is more or less condensed in the mcAM. On the other hand, a design pattern can be implemented in different ways in the cSC, depending on the different decisions made by developers. Thus, there might be a case that the design pattern-related class in the mcAM is divided into multiple classes to be implemented in the cSC. These classes in the cSC are intended to implement the design pattern modeled by that class in the mcAM.

Example of case MC-2 (selected from project 2): Figure 4.3 illustrates that a concept *decorators* is described by a class *ItemDecorator* in the mcAM. Decorators are part of the decorator design pattern [17]. Thereby, the naming of the class *ItemDecorator* and the relationships between *ItemDecorator* and the subclass *Upgrade* derived from the superclass *Item* possibly imply the decorator design pattern

is applied in this project. One or more *decorators* of the decorator design pattern might be further planned to decorate the subclass *Upgrade* derived from the superclass *Item* in the mcAM. To confirm whether the decorator design pattern is applied in the cSC and the concept *decorators* created in the mcAM remains in the cSC, we checked the detailed implementation of the cSC illustrated in Figure 4.4. Then we can know that the classes *Slot*, *LeftUpgradeSlot*, and *RightUpgradeSlot* are indeed the classes related to decorators of the decorator design pattern. Also, these classes are the extension of the interface *Icon* embedded in the Java library. Note that a MVC architectural pattern is adopted in the cSC. With this as a basis, in the cSC, the subclass *Upgrade* derived from the superclass *Item* is a Model-related class, and the interface *Icon* is invoked in a View-related class that is responsible for communicating with that subclass *Upgrade*. Therefore, these three classes in the cSC are considered to be related to decorators and are considered to be mapped to that class *ItemDecorator* in the mcAM since the concept *decorators* indeed remains in the cSC, and thereby described by those three classes.

Notably, the class *ItemDecorator* from the mcAM is divided into the three corresponding classes, *Slot*, *LeftUpgradeSlot*, and *RightUpgradeSlot*, for representing the decorator design pattern. The relationships of *realization* and *composition* related to *ItemDecorator* to be linked with *Upgrade* are thereby removed simultaneously.

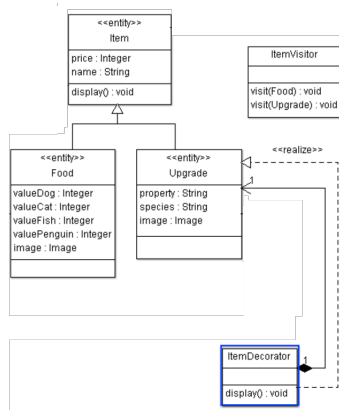


Figure 4.3: In the mcAM, a concept, i.e., *decorators* is described by a class *ItemDecorator*, which might imply the decorator design pattern will be applied in the cSC.

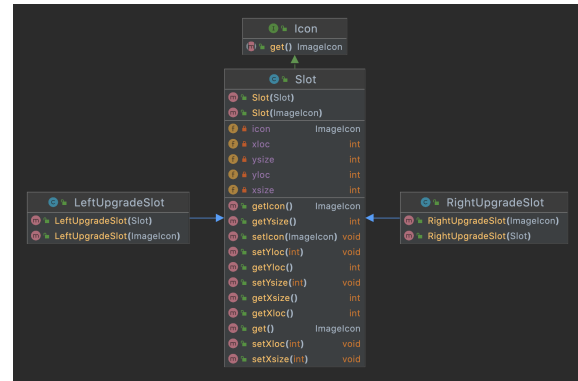


Figure 4.4: In the cSC, the concept *decorators* in the mcAM is described by the superclass *Slot*, and the subclasses *LeftUpgradeSlot*, and *RightUpgradeSlot* derived from the superclass *Slot*. These three classes are the extension of the interface *Icon* embedded in the Java library. This interface *Icon* is invoked within a View class, which is responsible for communicating with the subclass *Upgrade* derived from the superclass *Item* in the mcAM.

4.3.2 Cases of MA - Attributes

4.3.2.1 Case MA-1 - Additional attributes not in the mcAM are added to the cSC

Explanation of case MA-1: This case covers manual abstraction that hides attributes implemented in the cSC from mcAM. The level of abstraction created for the attributes depends on the architects' different design decision-making according to the required implementation structure of the system.

Note that a limitation exists here: this thesis did not investigate the project's context. Project's context, to some extent, relates to the implementation requirements of the system, e.g., some project description documents in the repository. Therefore, it is unknown why some attributes not planned out in the mcAM are added to the cSC.

Example of case MA-1 (selected from project 3): Referring to Table 4.4, for the class *IncomeRepository*, an additional attribute, i.e., *instance: IncomeRepository*, not in the mcAM is added to the cSC.

Table 4.4: The corresponding example of the case MA-1.

Class	mcAM	cSC
IncomeRepository		instance: IncomeRepository

4.3.2.2 Case MA-2* - An attribute of classifier A whose type can indicate an aggregation or a composition between classifiers A and B from the cSC, is modeled out by the naming of the association between them in the mcAM

Explanation of case MA-2*: The attribute type (represented by the name of the classifier) is essential for understanding exactly how different classifiers are linked. For example, if classifier A references classifier B as an attribute type, this indicates an aggregation or a composition (depending on the detailed implementation of the cSC) between classifiers A and B. However, to emphasize the links between the classifiers, the architects sometimes choose to name the relationships in the mcAM with the names of these attributes (whose specified types can indicate relationships between classifiers) in the cSC rather than specifying these attributes directly in the mcAM.

Example of case MA-2* (selected from project 4): As observed in the comparison between Figures 4.5 and 4.6, the naming of the association between the classes *LaborBilling* and *PhaseLabor* in the mcAM is specified with the name of the attribute *lb: LaborBilling*, i.e., *lb* in the cSC. *lb* indicates the links between the classes *LaborBilling* and *PhaseLabor*.

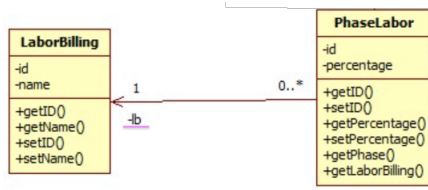


Figure 4.5: The naming of the association between the classes *LaborBilling* and *PhaseLabor* is specified with the attribute name *lb* in the cSC.

```

public class PhaseLabor {
    1 usage
    private String id;
    2 usages
    private LaborBilling lb;
    2 usages
    private Phase ph;
    2 usages
    private Float percentage;
}

```

Figure 4.6: In the cSC, the attribute name *lb* indeed exists in the class *PhaseLabor*.

4.3.2.3 Case MA-3* - The additional attribute type not in the mcAM is added to the cSC

Explanation of case MA-3*: This case covers manual abstraction that hides the attribute types in the cSC from mcAM. This is because the architects have different decisions in the design of mcAM attributes.

Note that attribute types are cSC constructs. It is impossible not to specify them in the cSC. Thereby, no opposite case caused by disAGTs exists.

Example of case MA-3* (selected from project 4): As observed in Table 4.5, the attribute named *id* is not specified with a type in the mcAM yet is specified with a type *String* in the cSC.

Table 4.5: The corresponding example of the case MA-3*.

Class	mcAM	cSC
Project	id	id: String

4.3.2.4 Case MA-4 - The additional default value not in the mcAM is added to the cSC

Explanation of case MA-4: The default value of an attribute in the mcAM is optional. There are three ways to initialize a member variable (an attribute): **1.** Assigning a default value to the member variable directly. **2.** The initialization of a member variable can be done within one or more constructors of the classifier for different instances of different classifiers at runtime. **3.** The initialization of a member variable can be done within the setter method of that variable. The latter two ways aim for the encapsulation of the attributes. Note that due to the syntax of the class diagram, the detailed implementation behind ways 2 and 3 cannot be shown in the mcAM, even though the architect might intend to initialize a variable via these two ways.

Although the above three ways can initialize a member variable, only architects and developers have a consensus that adopting way 1 can lead to this case. To be specific, some architects may not intend to assign the default values for some specific variables in the mcAM; however, the developers can either adopt way 1 to initialize a member variable or, based on the adoption of way 1, further adopt way 2 or/and way 3 to enable the default value of that attribute to be updated one or more times. Conversely, if the architects intend to assign a default value to a member variable by adopting way 2 or way 3; rather the developers adopt way 1 to initialize that member variable in the cSC. This will lead to another case DA-5* - The default value not in the mcAM is added to the cSC. This case is considered to be caused by disAGTs and is elaborated in subsubsection 4.3.6.5.

Example of case MA-4 (selected from project 2): Comparing Figures 4.7 and 4.8, for the attributes, e.g., *email* of the class *UserBuilder*, a default value “ ” not in the mcAM is added to that attribute in the cSC. Also, the public setter method *setEmail(email: String): UserBuilder* of that attribute provided in the cSC can allow the introduced default value “ ” to be updated one or more times for different instances of different classifiers.

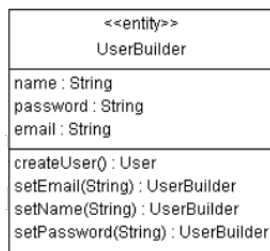


Figure 4.7: In the mcAM, for the attributes, e.g., *email*, a default value is not assigned.

```

public class UserBuilder {
    2 usages
    private int id=0;
    2 usages
    private String email=" ";
    2 usages
    private String username=" ";
    2 usages
    private String password=" ";

    4 usages
    public UserBuilder() {}

    public UserBuilder setId(int id) {...}

    2 usages
    public UserBuilder setEmail(String email) {
        this.email = email;
        return this;
    }
}
  
```

Figure 4.8: A default value “ ” not in the mcAM is added to the attribute *email*.

4.3.2.5 Case MA-5 - One or more common attributes in one or more subclasses in the mcAM are upshifted to the hierarchical structure in the cSC, i.e., the superclass inherited by these subclasses

Explanation of case MA-5: The architects may sometimes over-specify the common attributes possessed by one or more subclasses derived from one superclass in the mcAM. However, considering the *inheritance* concept of the OO paradigm, the developers might first abstract those common attributes of the subclasses in the

mcAM and then upshift these attributes to the hierarchical structure, i.e., the superclass inherited by these subclasses in the cSC. By doing so, those common attributes can be hidden, and only one or more of its derived subclasses in the cSC can access them. Code reuse is thus achieved. This means those upshifted attributes in the superclass in the cSC can be inherited by its subclasses and form part of its subclasses.

Example of case MA-5 (selected from project 2): Comparing Figures 4.9 and 4.10, the common attribute *image* of the subclasses *Food* and *Upgrade* in the mcAM is upshifted to the hierarchical structure, i.e., the superclass *Item* they derive from in the cSC. As a result, in the cSC, that upshifted attribute *image* of the superclass *Item* can be inherited by its two subclasses *Food* and *Upgrade* for sharing and reusing. Note that for the attribute *image*, its specified attribute type in the mcAM and cSC is different, i.e., *Image* and *String*. The difference in attribute types is considered to be caused by disAGTs, which is illustrated by another case DA-3* (in subsection 4.40).

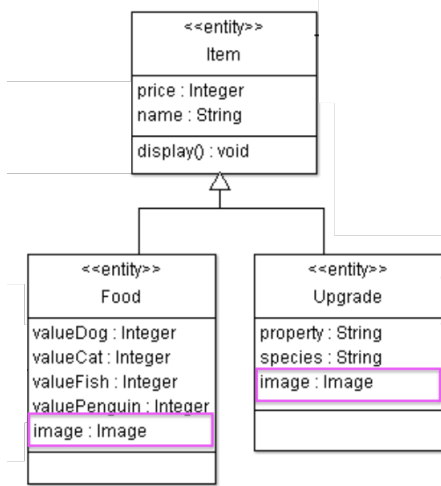


Figure 4.9: In the mcAM, for the subclasses *Food* and *Upgrade*, there is a common attribute *image*.

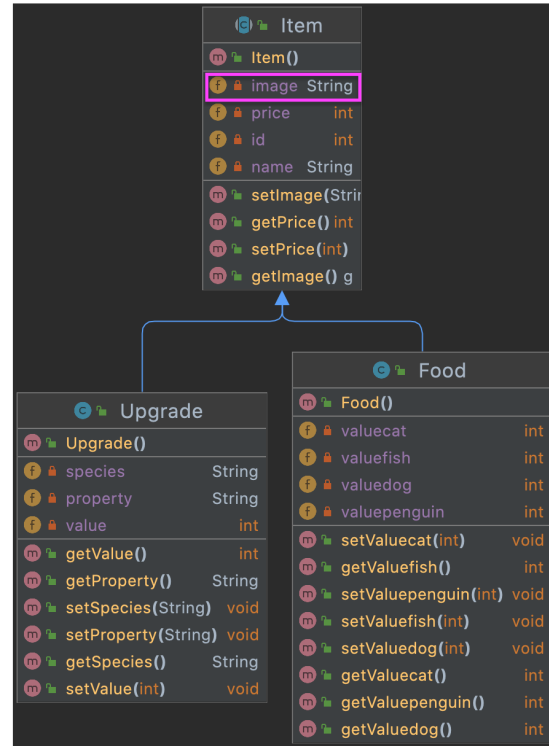


Figure 4.10: The common attribute *image* of the subclasses *Food* and *Upgrade* in the mcAM is upshifted to their superclass *Item* in the cSC.

4.3.2.6 Case MA-6 - One attribute created from the mcAM is divided into more than one attribute in the cSC

Explanation of case MA-6: This case covers manual abstraction that hides the multiple attributes in the cSC as one attribute from the mcAM.

Example of case MA-6 (selected from project 2): As observed in Table 4.6, for class *Dao*, its attribute *instance: DAO* from the mcAM is divided into more than one attribute in the cSC, e.g., *users: Map<Integer, User>* and *pets: Map<Integer, Pet>*.

Table 4.6: The corresponding example of the case MA-6.

Class	mcAM	cSC
DAO	instance: Dao	users: Map<Integer, User> pets: Map<Integer, Pet>

4.3.3 Cases of MA - Operations

Note that to avoid any confusion on the terminologies “operations” and “methods” used in the remainder of this thesis, “operations” used in the mcAM are represented by the methods (with condensing implementation details) in the cSC. They are ontologically identical.

4.3.3.1 Case MO-1 - Additional operations not in the mcAM are added to the cSC (three subcases)

Explanation of case MO-1: This case covers manual abstraction that hides operations implemented in the cSC from the mcAM. The level of abstraction created for the operations depends on the architects’ different design decision-making according to the required implementation structure of the system.

Note that a limitation exists here: this thesis did not investigate the project’s context. Project’s context, to some extent, relates to the implementation requirements of the system, e.g., some project description documents in the repository. Therefore, it is unknown why some operations not planned out in the mcAM are added to the cSC.

We observed the following 3 subcases:

- Subcase 1 - One or more constructors of the classifier

Explanation of subcase 1: Reference to the Java SE7 specifications [20], constructors are not defined as methods. However, it is observed that the constructors can be shown to be part of the operations in the mcAM. Thereby, one or more constructors of the classifier are considered operations in this thesis.

Example of subcase 1 (selected from project 3): Referring to Table 4.7, with respect to subcase 1, an additional constructor *Income()* not in the mcAM is purely added to the class *Income* in the cSC.

- Subcase 2 - One or more setter and getter operations for the encapsulation of the attributes (variables only)

Explanation of subcase 2: Considering the concept *encapsulation* of the OO paradigm, attributes need to be set to private, and their corresponding public setter and getter methods need to be provided for accessing and updating the values of these private attributes. Note that here the attributes refer to variables only. This is because a constant variable cannot be accessed and initialized by using the public setter and getter methods since the setter and getter methods cannot ensure that a constant variable is invoked once only across the life cycle of the program.

The encapsulated attributes can be the additional attributes (not in the mcAM added to the cSC) or the attributes created in the mcAM. Thereby, we identified the following two causes for subcase 2:

- Cause 1 - One or more setter or/and getter operations are purely added to the cSC for the encapsulation of attributes (variables only).

Example of cause 1 (selected from project 2): As observed in Table 4.7, an additional setter operation *setPrice(): int* not in the mcAM is purely added to the class *Item* in the cSC. This added setter operation in the cSC aims to encapsulate the attribute *price: int* created in the mcAM.

- Cause 2 - One or more setter or/and getter operations are purely added to the cSC for the encapsulation of attributes (variables only) that are caused by another case titled MA-1.

Example of cause 2 (selected from project 2): As observed in Table 4.7, an additional setter operation *setId(id: int): UserBuilder* not in the mcAM is added to the class *UserBuilder* in the cSC.

Notably, this added setter operation is used for encapsulating the attribute *id: int = 0*. This attribute is an additional attribute that is not in the mcAM yet is added to the class *UserBuilder* in the cSC.

- Subcase 3 - One or more operations (besides setter and getter operations)

Explanation of subcase 3: Since the constructors are not considered methods in Java, the methods in the cSC (besides the setter and getter methods) can be modeled as the operations in the mcAM.

Example of subcase 3 (selected from project 1): As observed in Table 4.7, an additional operation, e.g., *onPause(): void*, not in the mcAM is purely added to the class *SinglePlayer* in the cSC.

The above three differentiated subcases with their corresponding causes are illustrated in Figure 4.7.

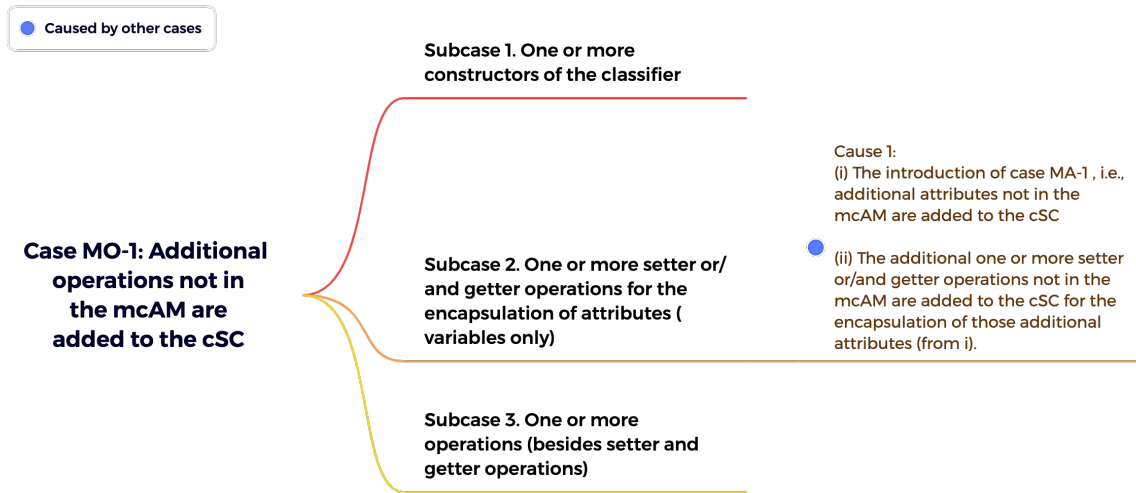


Figure 4.11: Three subcases correspond to this case with their corresponding causes.

Table 4.7: Corresponding examples of the causes for the three subcases of case MO-1.

Subcase ID	Cause No.	Class	mcAM	cSC
1		Income		Income()
2	1	Item		setPrice(): int
	2	UserBuilder	setId(id: int): UserBuilder	
3		SinglePlayer		onPause(): void

4.3.3.2 Case MO-2 - Additional parameter names not in the mcAM are added to the cSC

Explanation of case MO-2: Regarding the design of the parameters in the mcAM, the architects may sometimes not intend to specify the parameter names for some of the parameters. Then in the cSC, the developers design and specify the parameter names for those parameters that are unspecified with names in the mcAM. The parameter names are simply identifiers for parameters, yet the parameter types can imply the invocations between classifiers (or instances). Thus, Regarding the design of the parameters in the mcAM, a possibility is that some architects may prefer to leave the parameter type out solely. However, the possibility of leaving both parameter names and parameter types to be designed and implemented in the cSC by the developers cannot be excluded.

Example of case MO-2 (selected from project 2): As observed in Table 4.8, in the mcAM, the parameter name is unspecified for the parameter type *String* of the operation *setName(String): PetBuilder* in the class *UserBuilder*. However, a parameter name *name* is specified for that parameter type *String* of that operation in the cSC.

4.3.3.3 Case MO-3 - Additional parameter types not in the mcAM are added to the cSC

Explanation of case MO-3: Regarding the design of the parameters in the mcAM, the architects may sometimes not intend to specify the parameter types for some of the parameters. Then in the cSC, the developers design and specify parameter types for those parameters that are unspecified with types in the mcAM under the implementation requirements of the system.

Example of case MO-3 (selected from project 3): Referring to Table 4.8, it is observed that in the mcAM, the parameter type is unspecified for the operation *SetIncomeType()* of the class *IncomeRegisterUI*. However, in the cSC, a parameter type *List<IncomeType>* is added to that operation. Note that in this case, the name and type of the parameters are unspecified in pairs in the mcAM and added in pairs in the cSC.

4.3.3.4 Case MO-4 - Additional return types not in the mcAM are added to the cSC

Explanation of case MO-4: If the operation does not return a value, the colon and return type are omitted, e.g., *void* in Java and C++ [31, p. 45]. However, in this case, if the return type, i.e., *void* together with the colon, is omitted in the mcAM yet shown as *void* in the cSC. This change is not considered to be a difference.

Regarding the design of the operations in the mcAM, the architects may sometimes not intend to specify the return types for some of the operations. Then in the cSC, the developers design and specify the parameter types for those operations that are unspecified with return types in the mcAM under the implementation requirements of the system.

Example of case MO-4 (selected from project 1): As observed in Table 4.8, in the mcAM, the return type is unspecified for the operation *onKeyDown(key: int, event: KeyEvent)* of the class *SinglePlayer*. However, in the cSC, a return type *boolean* is added to that operation.

4.3.3.5 Case MO-5* - Multiplicity (referring to the collection-related interfaces) specified for the return type without its corresponding implementation interfaces/classes specified in the mcAM, yet with them specified in the cSC

Explanation of case MO-5*: Regarding the design of the return types (referring to the collection interfaces) in the mcAM, the architects might sometimes under-

specify them. This means the architects do not intend to specify their corresponding implementation classes and interfaces regarding the specific collection interfaces. Yet, in the cSC, if the interfaces are intended to be implemented, their corresponding implementation interfaces and classes need to be specified. Taking *Map* as an example, it is specified purely as *Map* in the mcAM. Rather, for its implementation, it is needed to be specified with its corresponding implementation classes/interfaces, e.g., *HashMap*. In accordance, the return type will become *Map<key, value> = new HashMap<key, value>* in the cSC.

Note that the specified corresponding implementation classes/interfaces with respect to the interfaces are cSC constructs. It is impossible not to specify them in the cSC. Thus, there is no opposite case caused by disAGTs.

Example of case MO-5* (selected from project 2): Referring to Table 4.8, it is observed that in the mcAM, the return type *Map* interface is specified for the operation named *listUser()*; however, the implementation class *HashMap* of that *Map* interface requires *Map* to be construed as *Map<key, value>* in the cSC. Thereby, that operation from the mcAM is implemented as *listUsers(): Map<Integer, User>* in the cSC.

4.3.3.6 Case MO-6* - The default parameters of the operation (in Android) in the cSC are omitted in the mcAM

Explanation of case MO-6*: This is a particular case for the cases mentioned above, i.e., MO-2 and MO-3. For Android development, multiple embedded APIs in the library can be leveraged directly in the cSC instead of manually building the code. Thus, for designing the operations in the mcAM, the architects may sometimes skip the design of the parameters of those operations (methods) that are embedded APIs in the library. Then in the cSC, the developers directly leverage these embedded operations (methods) from the library. Note that the reference to those embedded operations (methods) can be generated automatically by modern compilers, such as IntelliJ IDEA.

Note that the parameter types and parameter names are cSC constructs. It is impossible not to specify them in the cSC. Thus, there is no opposite case caused by disAGTs.

Example of case MO-6* (selected from project 1): Referring to Table 4.8, it is observed that *onCreate(savedInstanceState: Bundle): void* is an embedded operation (method) in the Android library. In the mcAM, this operation's default parameters are omitted, and it is shown as *onCreate()*. However, since the compiler can parse and generate its complete default parameters, these omitted default parameters are able to be automatically filled in in the cSC.

4.3.3.7 Case MO-7 - One operation created in the mcAM is divided into multiple operations in the cSC

Explanation of case MO-7: For inheritance hierarchies in the mcAM, i.e., more than one subclass derived from one superclass, the architects may sometimes create only one particular operation for that superclass in the mcAM. This operation describes the corresponding operations associated with those subclasses derived from that superclass in the mcAM. Thus, in the cSC, this particular operation of the superclass will be divided into corresponding multiple operations associated with those subclasses derived from that superclass.

Example of Case MO-7 (selected from project 2): In the detailed cSC implementation, a hierarchical inheritance structure exists, i.e., two subclasses *Food* and *Upgrade* inherit from a superclass *Item* in the mcAM. As observed in Table 4.8, an operation *eraseItem(mit: Item): boolean* is created with respect to the superclass *Item* within the class *Control* in the mcAM. However, this operation is divided into two corresponding operations *delFood(f: Food): boolean* and *delUpgrade(u: Upgrade): boolean* associated with those two subclasses *Food* and *Upgrade* in the cSC, respectively. Thus, these two operations from the cSC are hidden in the mcAM and thereby described by only one operation *eraseItem(mit: Item): boolean* in the mcAM.

Table 4.8: Six corresponding examples of the cases titled MO-2, MO-3, MO-4, MO-5*, MO-6*, and MO-7.

Example No.	Case ID	Class	mcAM	cSC
1	MO-2	UserBuilder	setName(String): PetBuilder	setName(name : String): PetBuilder
2	MO-3	IncomeRegisterUI	selectIncomeType()	selectIncomeType(listIncomeType: List<IncomeType>): IncomeType
3	MO-4	SinglePlayer	onKeyDown(key: int, event: KeyEvent)	onKeyDown(key: int, event: KeyEvent): boolean
4	MO-5*	DAO	listUser(): Map	listUsers(): Map< Integer, User >
5	MO-6*	TitlePage	onCreat()	onCreate(final savedInstanceState: Bundle): void
6	MO-7	Control	eraseItem(mit: Item): Boolean	delJob(Job) delUpgrade(Upgrade): Boolean

4.3.4 Cases of MA - Relationships

The following two Figures 4.12 and 4.13 can give the reader an overview of the remainder of this subsection. To be specific, we concluded three cases of MA in terms of relationships, namely cases MR-1 and MR-2, and MR-3, with their corresponding causes. Case MR-1 (with the corresponding subcases and the causes) is illustrated in Figure 4.12. Cases MR-2 and MR-3 (with their corresponding causes) are illustrated in Figure 4.13. Each case (with the corresponding causes) will be respectively detailed in the following subsubsections.

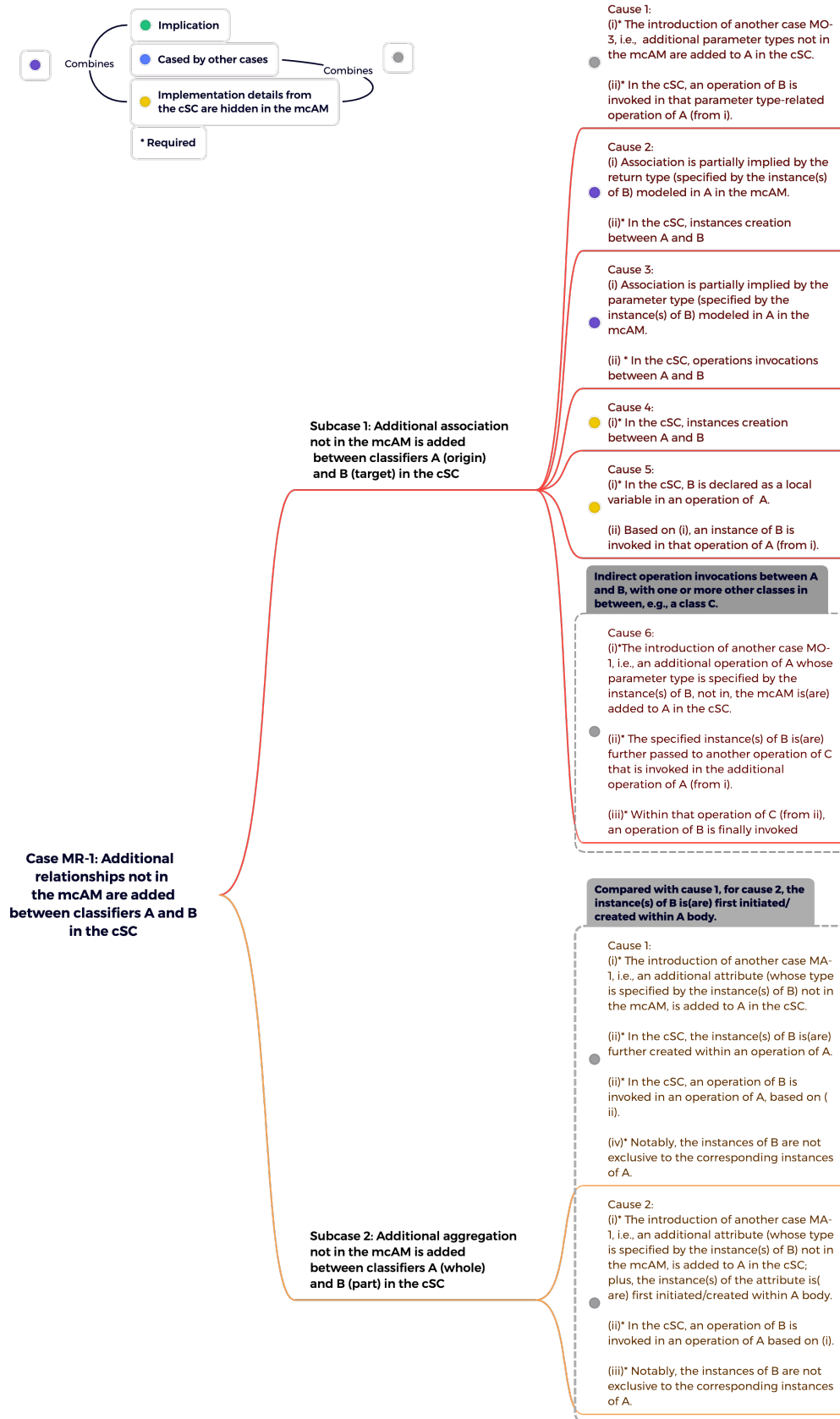


Figure 4.12: Case MR-1 with the corresponding subcases and causes.

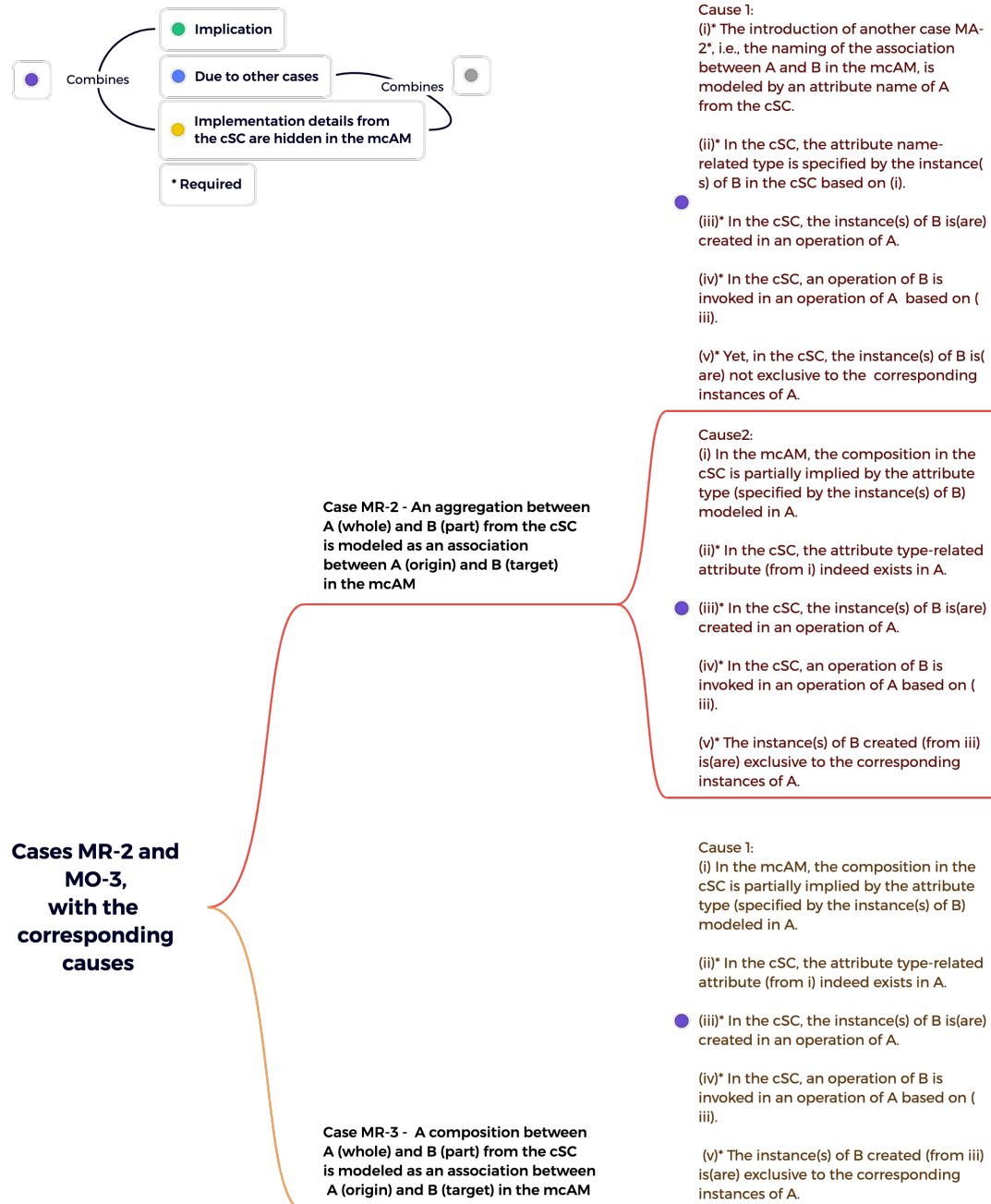


Figure 4.13: Cases MR-2 and MR-3, with the corresponding causes.

4.3.4.1 Case MR-1 - Additional relationships not in the mcAM are added to the cSC (two subcases with their six and two corresponding concluded causes, respectively)

Explanation of case MR-1: This case covers manual abstraction that hides the different types of relationships in the cSC from the mcAM. This is because of the different decision-making of the design of the mcAM relationships.

We observed the following two subcases for this case with their six and two corresponding concluded causes, respectively. The corresponding examples for these causes are given to illustrate each of these causes:

- **Subcase 1: Additional association not in the mcAM is added between classifiers A (origin) and B (target) in the cSC**

Explanation of subcase 1: This subcase covers manual abstraction that hides the association between classifiers A and B in the cSC from the mcAM.

We concluded the following six causes for this subcase, with their corresponding examples to illustrate each of these causes:

- **Cause 1:** (i)* The introduction of another case MO-3, i.e., additional parameter types not in the mcAM are added to A in the cSC. (ii)* In the cSC, an operation of B is invoked in that parameter type-related operation of A (from i).

Explanation of cause 1: The parameter types from the cSC hidden in the mcAM will lead the related associations to be hidden in the mcAM. This is because the parameter types (specified by the instance(s) of B) in A are involved in the operation invocations between A and B in the cSC. Thereby, if these parameter types from the cSC are hidden in the mcAM, their related associations from the cSC are possibly also hidden in the mcAM.

To confirm the existence of an association in the cSC, an instance of A must send a message to an instance of B [40]. This can be represented by the fact that an operation of B is invoked by an operation of A in the cSC. An association hereby can be confirmed to exist between A and B in the cSC.

Note that considering the syntax of the mcAM, the operation invocations between A and B cannot be modeled out in the mcAM.

Example of cause 1 (selected from project 4): Figure 4.14 illustrates that no relationship is modeled between the classes *LaborBilling* and *Pattern* in the mcAM. Yet, Figure 4.15 illustrates that in the cSC, an operation *getId()* of *LaborBilling* is invoked within an operation *get-*

PhaseLabor(LaborBilling, Phase): PhaseLabor of Pattern. Thereby, an association can be inferred to exist between the classes *LaborBilling* and *Pattern* in the cSC. Notably, these parameter types, e.g., *LaborBilling*, from the cSC are not modeled out in the mcAM but they are indeed involved in the operation invocations between *LaborBilling* and *Pattern* in the cSC. Thus, we considered that the parameter types from the cSC hidden in the mcAM can lead their related associations to be hidden in the mcAM.

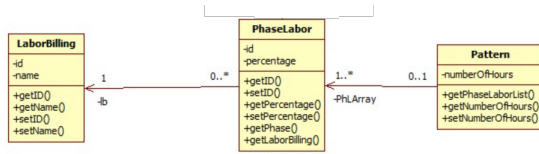


Figure 4.14: In the mcAM, no association is modeled between the classes *LaborBilling* and *Pattern*.

```
public class Pattern {
    // Complete implementation not shown here.
    // An instance of LaborBilling passed in here.
    public PhaseLabor getPhaseLabor(LaborBilling lbS, Phase phS) {
        for(int i = 0; i < plArray.size(); i++) {
            //An operation getId() of LaborBilling invoked here.
            if (plArray.get(i).getLaborBilling().getId().equals(lbS.getId()))
                // Complete implementation not shown here.
        }
        // Complete implementation not shown here.
    }
}
```

Figure 4.15: In the cSC, an operation *getId()* of *LaborBilling* is invoked in an operation *getPhaseLabor(LaborBilling, Phase): PhaseLabor of Pattern*. Yet, the parameter types of the latter operation, e.g., *LaborBilling*, are not modeled out in the mcAM (as shown in Figure 4.14).

- **Cause 2:** (i) Association is partially implied by the return type (specified by the instance(s) of B) modeled in A in the mcAM. (ii)* In the cSC, instances creation between A and B

Explanation of cause 2: The return type (specified by the instance(s) of B) modeled in A in the mcAM implies that a fact that an instance of A sends a message to an instance of B [40]. This instance of B further responds to this message sent by A. This response from B can be represented by returning an instance of B created in an operation of A.

Note that considering the syntax of the mcAM, instances creation between A and B cannot be modeled out in the mcAM.

Example of cause 2 (selected from project 3): Figure 4.16 illustrates that no relationship is modeled between the classes *BaseController* and *CheckingAccount* in the mcAM. Of particular note, in the mcAM, the return type of the operation *buildCheckingAccount(): CheckingAccount*, i.e., *CheckingAccount* has already partially implied an association that might exist between *BaseController* and *CheckingAccount* in the cSC. To confirm this, as illustrated in Figure 4.17, an instance of *CheckingAccount* is indeed created within that operation *buildCheckingAccount()*:

CheckingAccount of *BaseController* in the cSC. This instance is further returned to that operation. Thus, an association is inferred to exist between *BaseController* and *CheckingAccount* in the cSC. Yet, this association in the cSC is hidden from the mcAM and partially implied by the return type *CheckingAccount* modeled in *BaseController* from the mcAM.

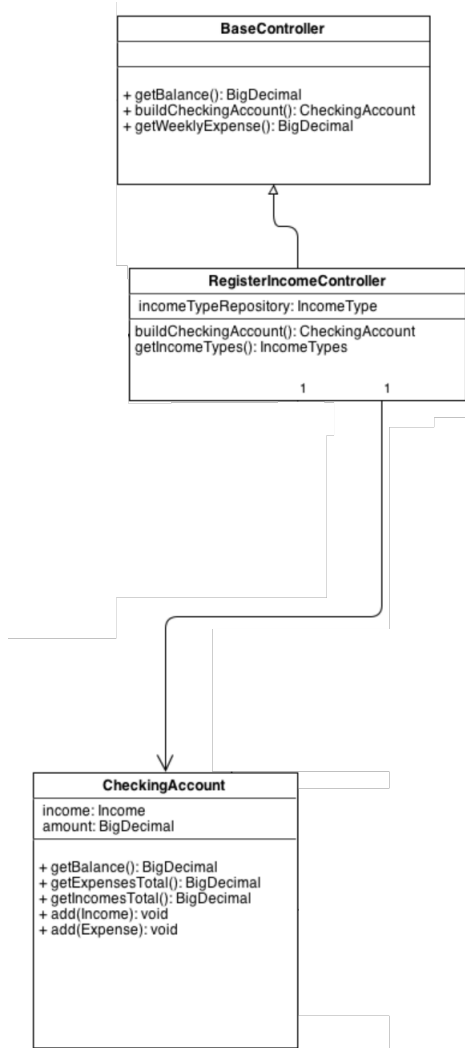


Figure 4.16: In the mcAM, no relationship is modeled between the classes *BaseController* and *CheckingAccount*. Plus, the return type *CheckingAccount* modeled in *BaseController* in the mcAM can partially imply an association to exist in the cSC.

```

public class BalanceController extends BaseController {
    // Complete implementation not shown here.
    public CheckingAccount buildCheckingAccount() {
        // CheckingAccount instances are created.
        return new CheckingAccount();
    }
}
  
```

Figure 4.17: In the cSC, an instance of *CheckingAccount* is created within the operation *buildCheckingAccount()*: *CheckingAccount* of *BaseController*, further being returned to this operation.

- **Cause 3:** (i) Association is partially implied by the parameter type (specified by the instance(s) of B) modeled in A in the mcAM. (ii)* In the cSC, operation invocations between A and B

Explanation of cause 3: The existence of an association between A and B in the cSC must build on the fact that an instance of A sends a message to an instance of B [40]. This can be represented by the operation invocations between A and B.

Note that considering the syntax of the mcAM, operation invocations between A and B cannot be modeled out in the mcAM.

Example of cause 3 (selected from project 2): Figure 4.18 illustrates that no relationship is modeled between the classes *ItemVisitor* and *Food* in the mcAM. Plus, the parameter type specified for the operation of *ItemVisitor*, e.g., *Food*, implies that the operations of *Food* might be invoked in that parameter type-related operation, i.e., *visit(Food): void* in the cSC.

Figure 4.19 illustrates for *Food*, an operation *getPrice(): int* its inherits from the superclass *Item* is indeed invoked within *visit(Food): void* of *ItemVisitor* in the cSC. This means the instances of *ItemVisitor* send a message to the instance *Food*, and their lifetimes are not related. An association thereby can be inferred to exist *ItemVisitor* and *Food* in the cSC, yet this association in the cSC is hidden from the mcAM.

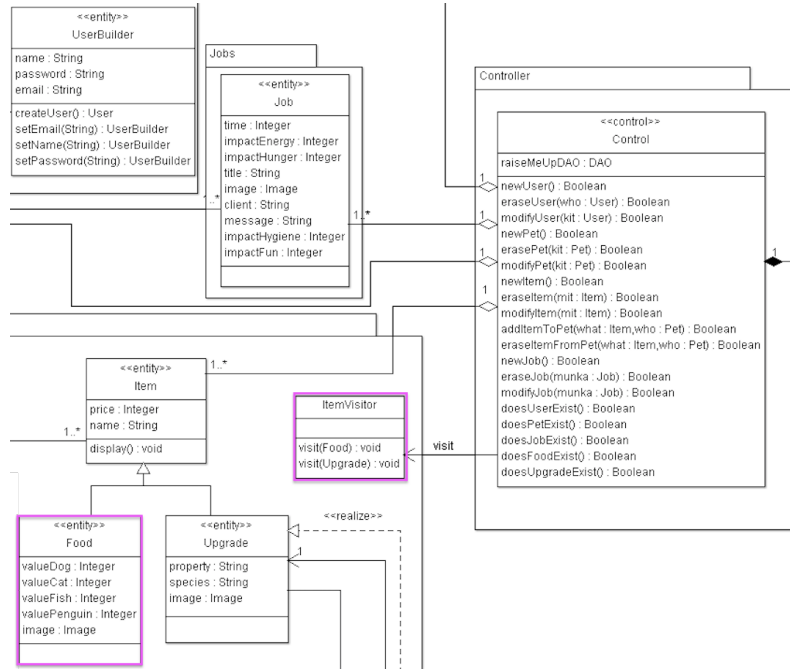


Figure 4.18: In the mcAM, no relationship is modeled between the classes *ItemVisitor* and *Food*. Yet, an instance of *Food*, i.e., is specified as a parameter type of the operation named *visit* of *ItemVisitor* in the mcAM.

```

public class ItemVisitor {
    // An Food instance is passed in here.
    public int visit(Food food) {
        // getPrice() of Food is invoked here.
        return food.getPrice();
    }
    public int visit(Upgrade upgrade) {
        return upgrade.getPrice();
    }
}

public class Food extends Item{
    // Complete implementation not shown here.
}

public abstract class Item {
    // Complete implementation not shown here.
    private int price;
    // Complete implementation not shown here.
    public int getPrice() {
        return price;
    }
    // Complete implementation not shown here.
}

```

Figure 4.19: In the cSC, for *Food*, its operation *getPrice(): int* inherited from the superclass *Item* is indeed invoked within *visit(Food): void* of *ItemVisitor* in the cSC.

- **Cause 4:** In the cSC, instances creation between A and B

Explanation of cause 4: The existence of an association between A and B in the cSC must build on the fact that an instance of A sends a message to an instance of B [40]. This can be represented by the instances creation between A and B.

Note that considering the syntax of the mcAM, instances creation between A and B cannot be modeled out in the mcAM.

Example of cause 4 (selected from project 1): Figure 4.20 illustrates that no relationship is modeled between the classes *SinglePlayer* and *TitlePage* in the mcAM. Yet, Figure 4.21 illustrates that an instance of *TitlePage* is created within an operation *pausedMainMenu(View): void* of *SinglePlayer* in the cSC. This is regarded that the instances of *SinglePlayer* send a message to that instance of *TitlePage*. Thereby, an association is inferred to exist between *SinglePlayer* to *TitlePage* in the

cSC, yet this association is hidden from the mcAM.

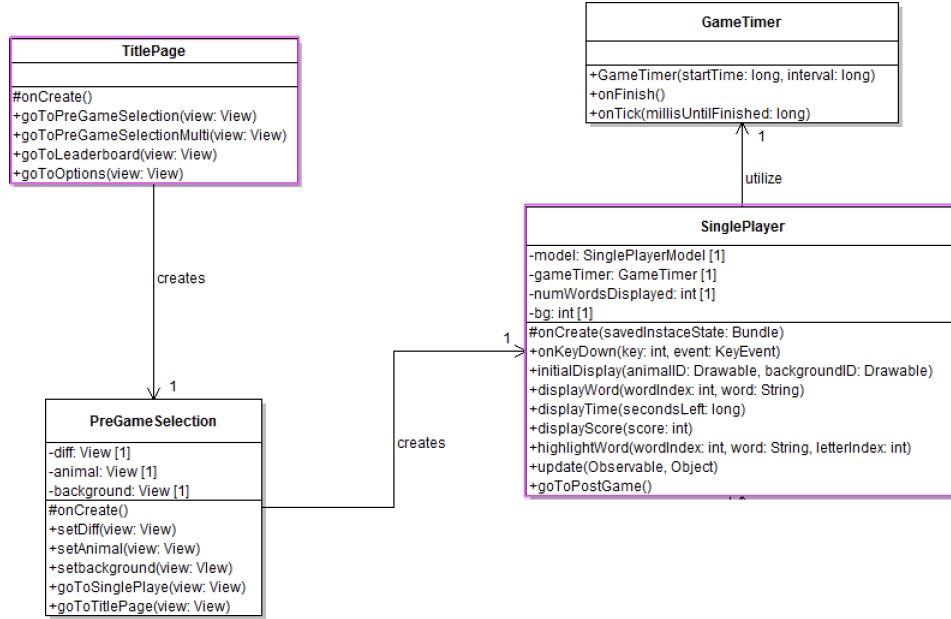


Figure 4.20: In the mcAM, no relationship is modeled between the classes *SinglePlayer* and *TitlePage*.

```

public class SinglePlayer extends Activity implements Observer {
    // Complete implementation not shown here.
    public void pausedMainMenu(View view) {
        final Intent mainMenuIntent = new Intent(this, TitlePage.class);
        startActivity(mainMenuIntent);
        paused = false;
    }
}

```

Figure 4.21: In the cSC, an instance of *TitlePage* is created within an operation *pausedMainMenu(View view): void* of *SinglePlayer*.

- **Cause 5:** (i)* In the cSC, B is declared as a local variable in an operation of A. (ii)* Based on (i), an instance of B is invoked in that operation of A (from i).

Explanation of cause 5: The existence of an association between A and B in the cSC must build on the fact that an instance of A sends a message to an instance of B [40]. This can be represented by the above holds - (i) and (ii).

Note that considering the syntax of the mcAM, local instances creation and further operation invocations between A and B cannot be modeled out in the mcAM.

Example of cause 5 (selected from project 2): Figure 4.22 illustrates that no relationship is modeled between the classes *UserBuilder* and *Control* in the mcAM. Yet, Figure 4.23 illustrates that an instance of *UserBuilder* is declared as a local variable named *userbuilder* within the operation *newUser(int, String, String, String)* of *Control* (in the mcAM)/*RaiseMeUp* (in the cSC). Plus, the operations, e.g., *setEmail(): String* of *userbuilder*, are further invoked within the same operation. An association is thereby inferred to exist between *UserBuilder* and *Control* in the cSC, yet this association is hidden from the mcAM.

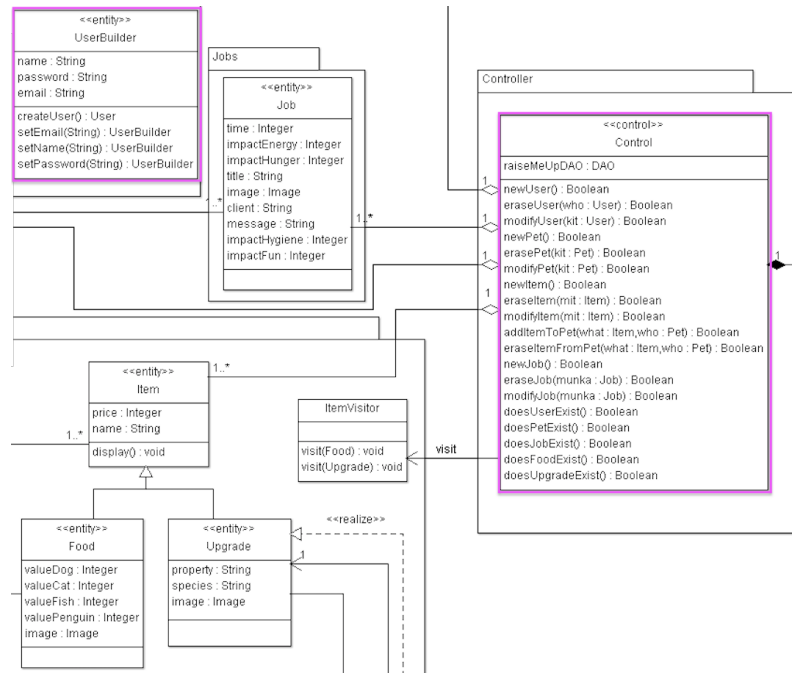


Figure 4.22: In the mcAM, no relationship is modeled between the classes *UserBuilder* and *Control*.

```
public class RaiseMeUp {
    public static boolean newUser(int id, String email,
        String username, String password) {
        UserBuilder userbuilder = new UserBuilder();
        userbuilder.setEmail(email);
        // Complete implementation not shown here.
    }
}
```

Figure 4.23: In the cSC, an instance named *userBuilder* of *UserBuilder* is created within the operation *newUser(int, String, String, String)* of *Control*. Plus, an operation of *userBuilder*, e.g., *setEmail(): String*, is further invoked with the same operation of *Control* (in the mcAM)/*RaiseMeUp* (in the cSC).

- **Cause 6:** (i)* The introduction of another case MO-1, i.e., an additional operation of A whose parameter type is specified by the instance(s) of B, not in the mcAM is added to A in the cSC. (ii)* The specified instance(s)

of B is(are) further passed to another operation of C that is invoked in the additional operation of A (from i). (iii)* Within that operation of C (from ii), an operation of B is finally invoked.

Explanation of cause 6: The operations from the cSC hidden in the mcAM might lead to their related associations being hidden in the mcAM as well. This is because those hidden operations from the cSC involve the operation invocations between A and B in the cSC. Of particular note, the operation invocations between A and B might involve one or more other classes in between. If so, this turns out to be an indirect operation invocation between A and B.

Example of cause 6 (selected from project 2): Figure 4.24 illustrates that no relationship is modeled between the classes *Food* and *Control* in the mcAM. Yet, Figure 4.25 illustrates that an instance of *Food* is specified as a parameter type of the additional operation *removeFood(Food: f): boolean* of *Control/Controll* (not in the mcAM added to the cSC) in the cSC. This instance, named *f*, is used in another operation *delFood(f)* of another C class *Dao*. Within *Dao*, an operation *getName()* of *Food* inherited from the superclass *Item* is further invoked within that operation *delFood(Food): boolean* in the cSC. An association is thereby inferred to exist between *UserBuilder* and *Control* in the cSC, yet this association is hidden from the mcAM.

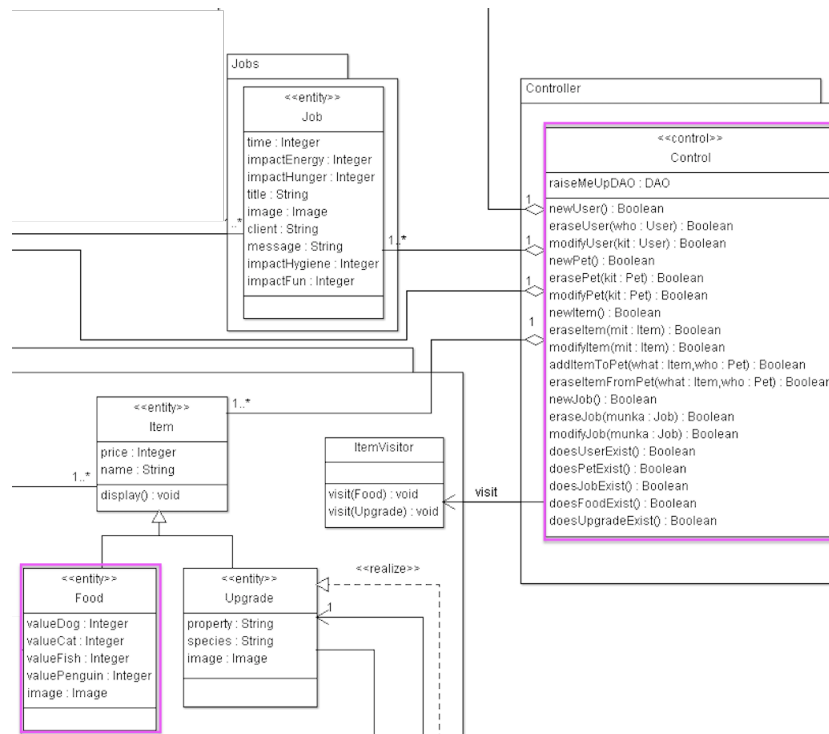


Figure 4.24: In the mcAM, no relationship is modeled between the classes *Food* and *Control*.


```

public class RaiseMeUp {
    private static DAO dao;
    // Complete implementation not shown here.
    // An instance of Food is passed in here.
    public static boolean removeFood(Food f) {
        try {
            /**
             * Food instance named f is used in
             * delFood() of another class DAO.
             */
            dao.delFood(f);
        } catch (SQLException ex) {
            // Complete implementation not shown here.
        }
        return true;
    }
    // Complete implementation not shown here.
}

public class Dao {
    /**
     * An additional operation delFood(Food): boolean
     * not in the mcAM is added to the cSC.
     */
    public boolean delFood(Food f) throws SQLException {
        // Complete implementation not shown here.
        try {
            // Complete implementation not shown here.
            pst.setString(index++, f.getName());
            // Complete implementation not shown here.
        } finally {
            // Complete implementation not shown here.
        }
        return true;
    }
}

```

Figure 4.25: In the cSC, an instance of *Food* (named *f*) is specified as a parameter type of an additional operation *removeFood(Food): boolean* of *Control/RaiseMeUp*. This instance of *Food* is used in another operation *delFood(f): boolean* of another class *Dao*. Plus, in *Dao*, an operation of *Food*, e.g., *getName()*, is further invoked with that *delFood(Food): boolean*.

- **Subcase 2 - Additional aggregation not in the mcAM is added between classifiers A (whole) and B (part) in the cSC**

Explanation of subcase 2: This subcase covers manual abstraction that hides the aggregation between classifiers A and B in the cSC from the mcAM.

We concluded the following two causes for this subcase:

- **Cause 1:** (i)* The introduction of another case MA-1, i.e., an additional attribute (whose type is specified by the instance(s) of B) not in the mcAM, is added to A in the cSC. (ii)* In the cSC, the instance(s) of B is(are) further created within an operation of A. (ii)* In the cSC, an op-

eration of B is invoked in an operation of A, based on (ii). (iv)* Notably, the instances of B are *not exclusive* to the corresponding instances of A.

Note that another case MO-1 might be involved in (ii), i.e., the operation where instance(s) of B is(are) created is an additional operation (not in the mcAM added to the A) in the cSC.

Explanation of cause 1: The attributes or-and operations in the cSC hidden from the mcAM possibly lead to the related aggregations/compositions in the cSC being hidden from the mcAM. This is because the definition of aggregations/compositions requires that the instance(s) of B must be accessed via an attribute type of A which is specified by the instance(s) of B, according to the source [40].

Example of cause 1 (selected from project 5): Figure 4.26 illustrates that no aggregation is modeled between the classes *Neuron* and *NeuralNetwork* in the mcAM. Yet, as illustrated in Figure 4.27, in the cSC, an additional attribute, i.e., *inputNeurons: NeurophArrayList<Neuron>*, not in the mcAM is added to *NeuralNetwork* in the cSC. The type of this attribute, i.e., *NeurophArrayList<Neuron>* implies an aggregation, or a composition between *Neuron* and *NeuralNetwork*. The instance *inputNeurons* further invokes its operation *size()* with the operation *getInputsCount(): int* of *NeuralNetwork*, and the instances of *NeuralNetwork* synchronously send a message to the instance *inputNeurons* of *Neuron*. Plus, the instances of *Neuron* are not exclusive to their corresponding instances of *NeuralNetwork*, i.e., the instances of *Neuron* also created within another class *Layer*. Thereby, an aggregation between *Neuron* and *NeuralNetwork* can be inferred to exist in the cSC. Yet, this aggregation in the cSC is hidden from the mcAM.

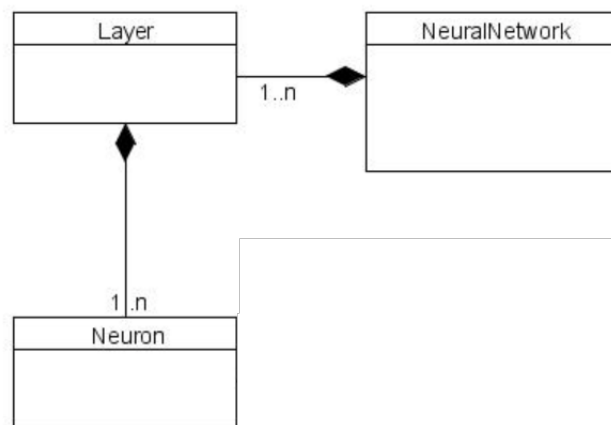


Figure 4.26: In the mcAM, no relationship is modeled between the classes *Neuron* and *NeuralNetwork*.

```

public class NeuralNetwork<L extends LearningRule>
implements Serializable {
    // Complete implementation not shown here.
    // Collection Neuron instances declared here.
    private NeurophArrayList<Neuron> inputNeurons;
    public NeuralNetwork() {
        // Complete implementation not shown here.
        // Collection Neuron instances created here.
        this.inputNeurons = new NeurophArrayList<>(Neuron.class);
    }
    public void setInput(double... inputVector) throws
    VectorSizeMismatchException {
        // size() of Neuron invoked here.
        if (inputVector.length != inputNeurons.size()) {
            // Complete implementation not shown here.
        }
        // Complete implementation not shown here.
    }
}

public class Layer implements Serializable {
    // Complete implementation not shown here.
    // Collection Neuron instances declared here.
    protected NeurophArrayList<Neuron> neurons;
    public Layer() {
        // Collection Neuron instances created here.
        neurons = new NeurophArrayList(Neuron.class);
    }
}

```

Figure 4.27: In the cSC, an additional attribute *inputNeurons*: *NeurophArrayList<Neuron>* not in the mcAM is added to the class *NeuralNetwork*. The operation *size()* of *Neuron* is further invoked within the operation *getInputsCount(): int* of *NeuralNetwork*. Plus, the instances of *Neuron* are contained by not only *NeuralNetwork* but also by *Layer* in the cSC.

- **Cause 2:** (i)* The introduction of another case MA-1, i.e., an additional attribute (whose type is specified by the instance(s) of B) not in the mcAM, is added to A in the cSC; plus, the instance(s) of B is(are) first created within A body. (ii)* In the cSC, an operation of B is invoked in an operation of A based on (i). (iii)* Notably, the instances of B are *not exclusive* to the corresponding instances of A.

Explanation of cause 2: Compared with the cause 1 mentioned above, the only difference between causes 1 and 2 is that *the instance(s) of B is(are) first created within A body* for cause 2. Yet, for cause 1, the instance(s) of B is(are) created within an operation of A.

Example of cause 2 (selected from project 2): Figure 4.28 illustrates no aggregation is modeled between the classes *Job* and *Dao* in the mcAM. Yet, as illustrated in Figure 4.29, in the cSC, an additional attribute, i.e., *jobs: Map<Integer, Food>*, not in the mcAM is added to *Dao* in the cSC. The type of this attribute (a collection of Job instances), i.e., *Map<Integer, Job>*, implies either an aggregation or a composition possibly to exist between *Job* and *Dao* in the cSC. An operation *put()* (an API) is further invoked in the operation *getJob(): Map<Integer, Job>* of *Dao* with the instance *jobs*. However, the instances of *Job* are also contained by another class *Pet*. The lifetimes of *Job* and *Dao* are thereby

not related. Thus, an aggregation can be inferred to exist between *Job* and *Dao* in the cSC. Yet, this aggregation in the cSC is hidden from the mcAM.

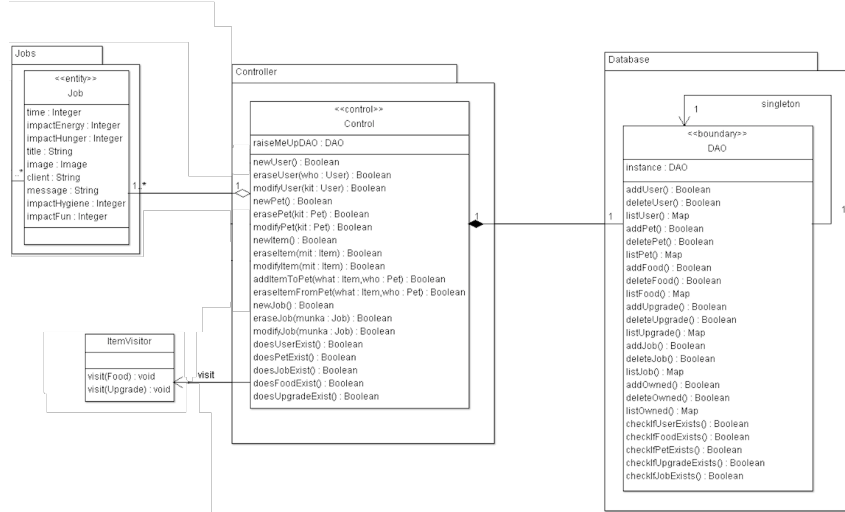


Figure 4.28: In the mcAM, no relationship is modeled between the classes *Job* and *Dao*.

```
public class DAO {
    // Complete implementation not shown here.
    // Collection instances of Job created here.
    Map<Integer, Job> jobs = new HashMap<Integer, Job>();
    public Map<Integer, Job> getJob() throws SQLException{
        // Complete implementation not shown here.
        jobs.clear();
        try {
            // Complete implementation not shown here.
            while(rs.next()){
                // An Job instance created here.
                Job j = new Job();
                j.setId(rs.getInt("id"));
                // Complete implementation not shown here.
                // An API, i.e., put() invoked here with Job instances
                jobs.put(rs.getInt("id"), j);
            }
        } finally {
            // Complete implementation not shown here.
        }
        return jobs;
    }
}

public class Pet {
    // Complete implementation not shown here.
    // Collection instances of Job created here.
    private Map<Job, Integer> ownedjobs = new HashMap<Job, Integer>();
    // Complete implementation not shown here.
}
```

Figure 4.29: In the cSC, an additional attribute *jobs*: *Map<Integer, Food>* not in the mcAM is added to the class *DAO* and it is further created in *Dao* body. An operation *put()* (an API) is further invoked in the operation *getJob()*: *Map<Integer, Job>* of *Dao* with the instance *jobs*. However, the instances of *Job* are contained by not only *DAO* but also by another class *Pet* in the cSC.

4.3.4.2 Case MR-2 - An aggregation between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (two causes)

Explanation of case MR-2: This case covers manual abstraction that models the aggregation from the cSC as an association in the mcAM.

We concluded the following two causes for this case, with their corresponding examples to illustrate each of these causes:

- **Cause 1:** (i) The introduction of another case MA-2*, i.e., the naming of the association between A and B in the mcAM, is modeled by an attribute name of A from the cSC. (ii)* In the cSC, the attribute name-related type is specified by the instance(s) of B in the cSC based on (i). (iii)* In the cSC, the instance(s) of B is(are) created in an operation of A. (iv)* In the cSC, an operation of B is invoked in an operation of A based on (iii). (v)* Yet, in the cSC, the instance(s) of B is(are) *not exclusive* to the corresponding instances of A.

Explanation of cause 1: Referring to the case MA-2* in subsection 4.3.2.2, the naming of the association between A and B in the mcAM is specified by an attribute name contained in A from the cSC. This attribute-related type is specified by the instances of B, which can indicate the existence of an aggregation or a composition (relying on the detailed cSC implementation) between A and B in the cSC. Regarding this case, it refers to an aggregation. This is built on the fact - (ii), (iii), (iv), and (v) mentioned above.

Example of cause 1 (selected from project 4): Figure 4.30 illustrates that in the mcAM, an association is modeled between the classes *Employee* and *Schedule*. Also, the naming of this association is specified by the name of an attribute *schedlist: ArrayList<Schedule>* of *Employee* from the cSC, i.e., *schedlist*. Figure 4.31 illustrates that in the cSC, the type of this attribute, i.e., *ArrayList<Schedule>*, indicates that a group of instances of the class *Schedule* (ranged in $[0, +\infty]$) is referenced as an attribute type of the class *Employee*. These instances are further initiated within the operation *Employee(Integer id, String nm, String position)* of *Employee*. Yet, the instances of *Schedule* are contained not only by the instances of *Employee* in the cSC but also by the instances of another class *MainClass*. An aggregation thus can be inferred to exist in the cSC rather than that association modeled in the mcAM.

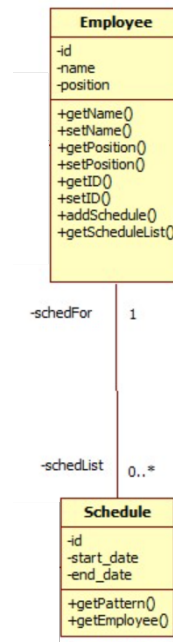


Figure 4.30: The aggregation between *Employee* and *Schedule* from the cSC is modeled as an association in the cSC. Plus, the naming of this association in the mcAM is specified by an attribute name (whose related type is specified by the instances of *Schedule*) of *Employee* from the cSC.

```

public class Employee {
    // Complete implementation not shown here.
    // Collection instances of Schedule declared here.
    private ArrayList<Schedule> schedList;
    public Employee(Integer id, String nm, String position) {
        // Instances of Schedule created here.
        schedList = new ArrayList<Schedule>();
        // Complete implementation not shown here.
    }

    public void addSchedule(Schedule scb) {
        // add() of Schedule invoked here.
        schedList.add(scb);
    }
}

public class MainClass {
    // Complete implementation not shown here.
    public void doMain() {
        Employee k = new Employee(1, "Marcel Codrea", "Trainee");
        // Complete implementation not shown here.
    }
}
  
```

Figure 4.31: In the cSC, an attribute type *ArrayList<Schedule>* contained in the class *Employee* means a group of instances of the class *Schedule* declared in *Employee*. These instances of *Schedule* are further created within the operation *Employee(Integer, String, String)* of *Employee* in the cSC. However, the instances of the class *Schedule* are contained not only by the instances of *Employee* but also by the instances of another class *MainClass* in the cSC.

- **Cause 2:** (i) In the mcAM, the aggregation from the cSC is partially implied by the attribute type (specified by the instance(s) of B) of A in the mcAM. (ii)* In the cSC, instance(s) of B is(are) indeed created in an operation of A. (iii)* In the cSC, an operation of B is invoked in an operation of A based on (ii). (iv)* Yet, in the cSC, the instance(s) of A is(are) *not exclusive* to the corresponding instances of B.

Explanation of cause 2: Either an aggregation or a composition between A and B from the cSC can be implied by the attribute type (specified by the instance(s) of B) modeled in A in the mcAM. Yet, due to the different decision-making by architects in terms of the design of the MA relationships, they may choose to simply model such an aggregation or a composition from the cSC as an association in the mcAM. In this case, it refers to an aggregation in the cSC. This is based on the fact - (ii), (iii), and (iv) mentioned above.

Example of cause 2 (selected from project 3): Figure 4.32 illustrates that in the mcAM, an association is created between the classes *Income* and *IncomeRepository*. Yet, the attribute type (a collection of instances) of *IncomeRepository* in the mcAM, i.e., *List<Income>*, possibly implies the existence of either an aggregation or a composition in the cSC. However, due to the different decision-making by architects during the mcAM relationships design, they choose to simply model such an aggregation or composition between *Income* and *IncomeRepository* from the cSC as an association in the mcAM.

Figure 4.33 illustrates that the attribute modeled in the mcAM, i.e., *listIncome: List<Income>* indeed exists as *listIncome: ArrayList<Income>* in the cSC. The referenced collection of instances of *Income* by *IncomeRepository*, i.e., *List<Income>*, is created within the operation *IncomeRepository()* of *IncomeRepository* in the cSC. The instances of *Income* are further invoked within another operation *save(Income): void* of *IncomeRepository* in the cSC. An aggregation thereby can be inferred to exist between *Income* and *IncomeRepository* in the cSC. Yet, the instances of *Income* are contained by not only the corresponding instances of *IncomeRepository* but also by the corresponding instances of another class *RegisterIncomeController*. Thus, this aggression is not a composition in the cSC.

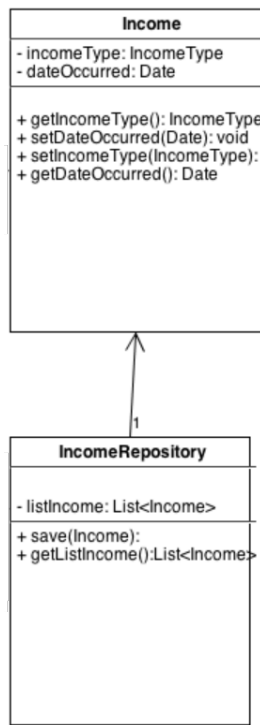


Figure 4.32: An aggregation between *Income* and *IncomeRepository* from the cSC is partially indicated by the attribute type (instances of *Income*) of *IncomeRepository*, i.e., *List<Income>*, in the mcAM.

```

public class IncomeRepository {
    /**
     * Collection instances of Income
     * is declared as an attribute type
     */
    private static List<Income> listIncome;
    public IncomeRepository() {
        // Collection instances of Income are created.
        listIncome = new ArrayList<Income>();
    }
    // Complete implementation not shown here.
    public void save(Income income) {
        if (income == null) {
            throw new IllegalArgumentException();
        }
        // Income instances are invoked.
        getListIncome().add(income);
    }
    // Complete implementation not shown here.
    public static List<Income> getListIncome() {
        return listIncome;
    }
}

public class RegisterIncomeController extends BaseController {
    // Complete implementation not shown here.
    public void createIncome(BigDecimal amount, IncomeType incomeType,
        String what, Date date) {
        // Complete implementation not shown here.
        // Income instances are created.
        checkingAccount.add(new Income(amount, incomeType, what, date));
    }
}
  
```

Figure 4.33: In the cSC, the type of the attribute named *listIncome* is specified by a collection of instances of *Income*, i.e., *ArrayList<Income>* in *IncomeRepository*. This collection of instances of *Income* is further created within an operation *IncomeRepository()* of *IncomeRepository*. Plus, the instances of *Income* are invoked within an operation *save(Income): void* of *IncomeRepository* in the cSC. Yet, the instances of *Income* are contained by not only *IncomeRepository* but also another class *RegisterIncomeController* in the cSC.

4.3.4.3 Case MR-3 - A composition between A (whole) and B (part) from the cSC is modeled as an association between A (origin) and B (target) in the mcAM (one cause)

Explanation of case MR-3: This case covers manual abstraction that models the composition between classifiers A (whole) and B (part) from the cSC as an association in the mcAM.

We concluded the following cause for this case, with the corresponding example to illustrate this cause:

- **Cause 1:** (i) In the mcAM, the composition in the cSC is partially implied by the attribute type (specified by the instance(s) of B) modeled in A. (ii) * In the cSC, the attribute type-related attribute (from i) indeed exists in A. (iii)* In the cSC, the instance(s) of B is(are) created in an operation of A. (iv)* In the cSC, an operation of B is invoked in an operation of A based on (iii). (v)* The instance(s) of B created (from iii) is(are) *exclusive* to the corresponding instances of A.

Explanation of cause 1: If an instance of A is modeled out as an attribute type in A in the mcAM, this can indicate either an aggregation or a composition, which might exist in the cSC. Yet, due to different decision-making by architects, they might sometimes choose to model that aggregation or composition as an association in the mcAM. To confirm whether it is an aggregation or a composition in the cSC, checking the detailed cSC implementation is essential. In this case, it refers to a composition from the cSC based on the fact - (ii), (iii), and (iv) mentioned above.

Example of cause 1 (selected from project 1): Figure 4.34 illustrates that an association is modeled between the classes *SinglePlayer* and *SinglePlayModel* in the mcAM. Of particular note, an instance of *SinglePlayModel* is modeled out as an attribute type in *SinglePlayer* in the mcAM. This thereby indicates either an aggregation or a composition that might exist in the cSC, which is yet modeled as an association in the mcAM.

Figure 4.35 illustrates in the cSC an instance of *SinglePlayerModel* is indeed referenced as an attribute type of *SinglePlayer* in the cSC. This instance named *model* is further created within the operation *onCreate(Bundle): void* of *SinglePlayer* in the cSC and is further invoked in an operation *onKeyDown(int, KeyEvent): boolean* of *SinglePlayerModel* in the cSC. Of particular note, this instance *model* is exclusive to the corresponding instances of *SinglePlayerModel*. A composition thereby can be inferred to exist between *SinglePlayer* and *SinglePlayerModel* in the cSC. Yet, this composition between *SinglePlayer* and *SinglePlayerModel* from the cSC is modeled as an association in the mcAM.

Notably, here the class name *SinglePlayModel* in the mcAM is different from the class name *SinglePlayModel* in the cSC. Yet, they are actually the same class since they have highly similar attributes and operations. Class name differences are considered to be caused by CC, which is illustrated in section 4.4.

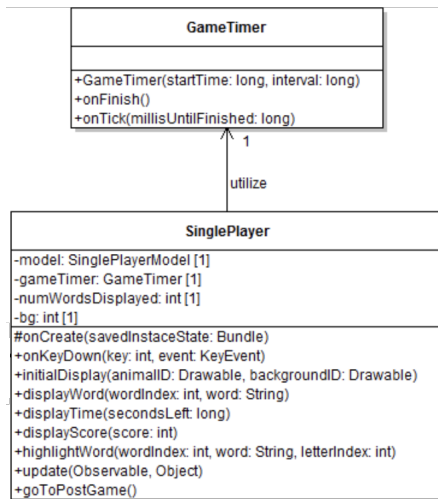


Figure 4.34: A composition between the classes *SinglePlayer* and *SinglePlayModel* from the cSC is modeled as an association in the mcAM.

```

public class SinglePlayer extends Activity implements Observer {
    // Attribute type is specified by an instance of SinglePlayerModel
    private SinglePlayerModel model;

    protected final void onCreate(final Bundle savedInstanceState) {
        // Complete implementation not shown here.
        // SinglePlayerModel instance created here.
        model = new SinglePlayerModel(d, this.getAssets(), NUM_WORDS);
    }

    public final boolean onKeyDown(final int key, final KeyEvent event){
        // Complete implementation not shown here.
        char charTyped = event.getDisplayLabel();
        charTyped = Character.toLowerCase(charTyped);
        // typeLetter() of SinglePlayerModel invoked here.
        model.typeLetter(charTyped);
        return true;
    }
}
  
```

Figure 4.35: In the cSC, an instance *model* of the class *SinglePlayerModel* that is modeled out as an attribute type in the class *SinglePlayer* in the mcAM is indeed declared as an attribute type of the class *SinglePlayer* in the cSC. This created instance *model* of *SinglePlayerModel* is further invoked within an operation of *SinglePlayerModel*, e.g., *onKeyDown(int, KeyEvent): boolean*. Plus, in the cSC, this instance *model* is exclusive to the corresponding instances of *SinglePlayer*.

4.3.5 Cases of disAGTs - Classes

4.3.5.1 Case DC-1 - Hierarchical inheritance structures in the mcAM are removed in the cSC

Explanation of case DC-1: In the mcAM, a concept *a* can be described by a superclass or a class. If the architects decide to model a concept as a superclass for its subclasses, their corresponding second concepts *b(s)* can be described by this superclass. However, a possibility is that the architects might sometimes over-specify the subclasses for a superclass, yet the developers disagree with this; rather, they decide to remove the subclasses derived from that superclass in the cSC. Thereby, the corresponding hierarchical inheritance structures in the mcAM will be removed in the cSC.

Example of case DC-1 (selected from project 2): Comparing Figures 4.36 and 4.37, a hierarchical inheritance structure in the mcAM is removed in the cSC. To be specific, the four subclasses *Dog*, *Cat*, *Fish*, and *Penguin* in the mcAM are removed in the cSC, along with their superclass *Pet* in the mcAM

being changed into a class in the cSC.

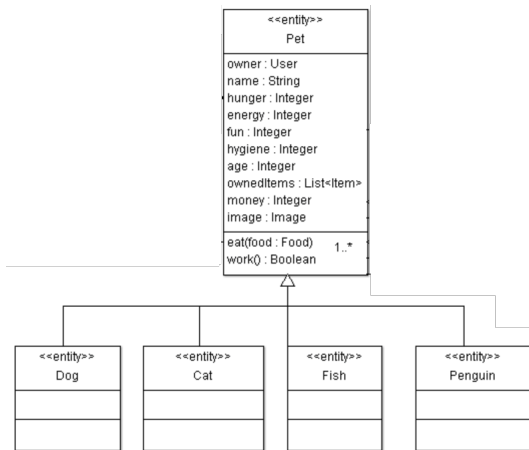


Figure 4.36: A hierarchical inheritance structure in the mcAM.

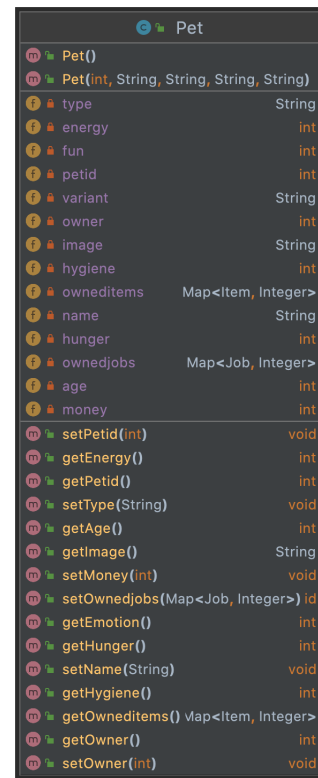


Figure 4.37: The hierarchical inheritance structure in the mcAM is removed in the cSC. Solely the superclass *Pet* in the mcAM is remained yet changed into a class in the cSC.

4.3.6 Cases of disAGTs - Attributes

4.3.6.1 Case DA-1 - Attributes in the mcAM are removed in the cSC

Explanation of DA-1: This case covers deviations between the mcAM design and cSC implementation in terms of attributes. Those over-specified attributes from the mcAM are removed in the cSC.

Example of DA-1 (selected from project 2): As observed in Table 4.9, for the class *User*, the attribute *pets*: *List<Pet>* from the mcAM is removed in the cSC.

Table 4.9: The corresponding example of the case DA-1.

Class	mcAM	cSC
User	pets: List<Pet>	

4.3.6.2 Case DA-2* - Attributes from the mcAM are replaced with additional attributes (that are not in the mcAM added to the cSC) in the cSC

Explanation of case DA-2*: In terms of attributes, there are deviations between the mcAM design and cSC implementation. One possibility is replacing the attributes modeled from the mcAM with fully new attributes in the cSC.

Example of case DA-2* (selected from project 3): As observed in the comparison between Figures 4.38 and 4.39, for the class *CheckingAccount*, all its attributes, i.e., *income: Income* and *amount: BigDecimal* in the mcAM are replaced with fully new attributes *incomeRepo: IncomeRepository* and *expenseRepo: ExpenseRepository* in the cSC.

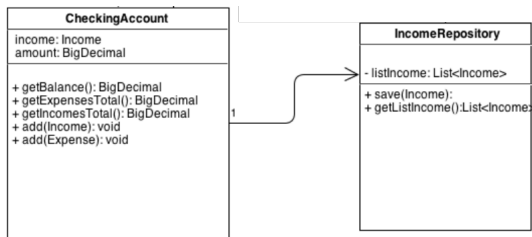


Figure 4.38: For the class *CheckingAccount*, the attributes *income: Income* and *amount: BigDecimal* are modeled in the mcAM.

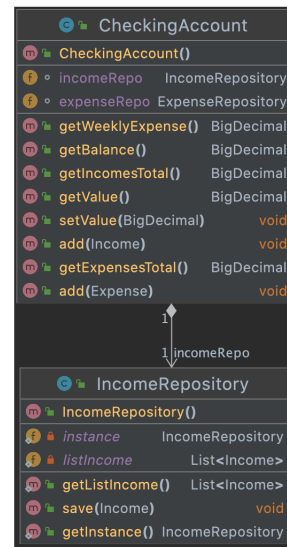


Figure 4.39: For the class *CheckingAccount*, all its attributes in the mcAM are replaced with fully new attributes *incomeRepo: IncomeRepository* and *expenseRepo: ExpenseRepository* in the cSC.

4.3.6.3 Case DA-3* - The attribute type in the mcAM and cSC is different (two causes)

Explanation of case DA-3*: The attribute type can be either *name of the classifier* or *ProgrammingLanguageDataType*. Regarding Java, the former

refers to a non-primitive data type, and the latter refers to a primitive data type.

According to this, we concluded the following two causes for this case:

- **Cause 1:** Conversion between non-primitive data types.

Example of cause 1 (selected from project 2): As observed in Table 4.10, for the operation *ownedItems* of the class *Pet*, its data type is changed from the interface *List* (a non-primitive data type) in the mcAM to another interface *Map* (a non-primitive data type) in the cSC.

- **Cause 2:** Conversion of non-primitive and primitive data types.

Example of cause 2 (selected from project 2): Table 4.10 illustrates that for the operation *owner* of the class *Pet*, its data type changed from the class *User* (a non-primitive data type) to *int* (a primitive data type).

Table 4.10: Two corresponding examples of the causes for case DA-3*.

Example No.	Causes	Class	mcAM	cSC
1	1	Pet	ownedItems: List<Item>	ownedItems: Map<Item, Integer>
2	2	Pet	owner: User	owner: int

The above two causes are illustrated in Figure 4.40.

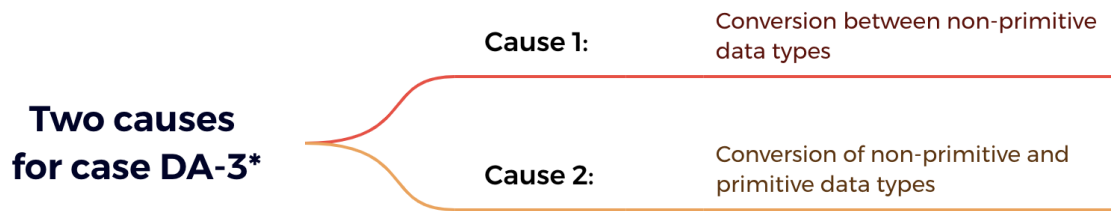


Figure 4.40: Two causes for case DA-3*.

4.3.6.4 Case DA-4* - Converting variables from the mcAM to constant variables in the cSC

Explanation of case DA-4*: This case covers the deviations of the cSC from the mcAM in terms of the types of variables.

Example of case DA-4* (selected from project 1): Referring to Table 4.11, it is observed that the naming of the attribute is changed from *numWordsDisplayed* from the mcAM to *NUM_WORDS* in the cSC. To be specific,

the attribute name *numWordsDisplayed* in a camel-case represents a variable in the mcAM. However, the developers disagree with the design of this variable in the mcAM, and they convert it to a constant variable, i.e., all letters of the attribute name *NUM_WORDS* in uppercase.

4.3.6.5 Case DA-5* - The default value not in the mcAM is added to the cSC

Explanation of case DA-5*: This case is caused by **subcase 1 of case DA-4***, i.e., converting one or more variables in the mcAM to one or more constant variables in the cSC. There are two methods to initialize a constant variable in the cSC: **1.** Assigning a value to the constant variable only if the constant variable is without an assignment prior to being assigned [52]. **2.** The initialization of a constant variable can be done within one or more constructors of the classifier. Method 2 is for ensuring that constant variables are only initialized once for different instances created from different classifiers. Of particular note, if method 2 is adopted, considering the syntax of the class diagrams, the initialization of a constant variable within a specific constructor cannot be shown in the mcAM. This cannot lead to this case; rather, taking subcase 1 of case DA-5* as a basis, further adopting method 1 can lead to this case.

Below is an example of method 1 adopted in the cSC.

Example of case DA-5* (selected from project 1): Taking subcase 1 of case DA-4* as a basis, i.e., a variable named *numWordsDisplayed* in the mcAM is converted to a constant variable named *NUM_WORDS* in the cSC. As observed in Table 4.11, for the class *SinglePlayer*, an additional default value 5 not assigned to the variable *numWordsDisplayed: int* in the mcAM is assigned to its converted constant variable *NUM_WORDS: int* in the cSC.

Note that the attribute naming here is different in the mcAM and cSC, i.e., *numWordsDisplayed* in the mcAM and *NUM_WORDS* in the cSC. The attribute naming difference(s) is(are) illustrated in the example for case DA-4* (in subsection 4.3.6.4).

Table 4.11: The corresponding example of the cases DA-4* and DA-5* .

Class	mcAM	cSC
SinglePlayer	numWordsDisplayed:int	NUM_WORDS: int = 5

4.3.7 Cases of disAGTs - Operations

4.3.7.1 Case DO-1 - Operations in the mcAM are removed in the cSC

Explanation of case DO-1: According to the implementation requirements of the system, the architects might sometimes over-specify the operations in the mcAM. Yet, the developers disagree with that, and in the implementation of the cSC, they decide to remove some of those over-specified operations in the mcAM.

Example of case DO-1 (selected from project 2): Referring to Table 4.12, it is observed that for the class *Pet*, an operation *eat(food: Food)* from the mcAM is removed in the cSC.

4.3.7.2 Case DO-2 - The parameter names in the mcAM are removed in the cSC

Explanation of case DO-2: This case covers a deviation from the mcAM to the cSC in terms of the parameters. The over-specified parameters from the mcAM are removed in the cSC, and their related types are thereby removed as well.

Example of case DO-2 (selected from project 2): As observed in Table 4.12, for the operation named *modifyPet* of the class *Control*, the specified parameter name *kit* is paired with the related type *Pet* from the mcAM to be removed in the cSC.

4.3.7.3 Case DO-3 - The parameter types in the mcAM are removed in the cSC

Explanation of case DO-3: According to the implementation requirements of the system, the architects might sometimes over-specify the parameter types in the mcAM. Yet, the developers disagree with that, and in the implementation of the cSC, they decide to remove some of those over-specified parameter types in the mcAM. Notably, by doing this, those types-related names are removed as well.

Example of case DO-3 (selected from project 2): As observed in Table 4.12, for the operation named *modifyPet* of the class *Control*, a parameter type *Pet* specified for it from the mcAM is removed in the cSC.

4.3.7.4 Case DO-4* - The parameter type in the mcAM and cSC is different (one cause)

Explanation of case DO-4*: Of particular interest is considering the concept *encapsulation* of the OO paradigm, public setter or/and getter operations provided for encapsulating private attributes. If the encapsulated attribute's

type planned out from the mcAM is changed in the cSC, it likely leads to its corresponding setter operation's parameter type from the mcAM being changed in the cSC. Thereby, the conversion of attribute types requires special attention (referring to case DA-3*, in subsubsection 4.3.6.3).

Example of case DO-4* (selected from project 2): As observed in the comparison between Figures 4.41 and 4.42, first, for the attribute named *owner*, an attribute type *User* (a non-primitive data type) specified for it in the mcAM is changed to *int* (a primitive data type) in the cSC. Accordingly, the parameter type specified for its setter operation named *setOwner()* is changed from *User* in the mcAM to *int* in the cSC.

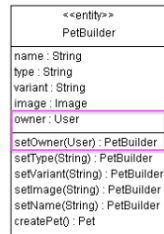


Figure 4.41: In the mcAM, for the attribute named *owner*, *User* (a non-primitive data type) is specified. Also, for its setter operation named *setOwner()*, a parameter type *User* is specified.

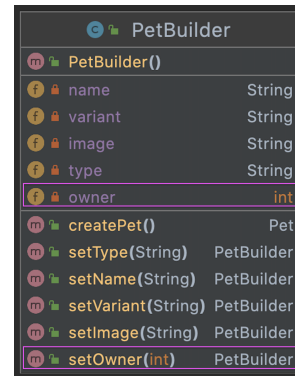


Figure 4.42: In the cSC, for that attribute named *owner* its specified type *User* in the mcAM changed to *int* instead. Accordingly, for the setter operation named *setOwner()* of that attribute, its parameter type changed from *User* to *int*.

4.3.7.5 Case DO-5 - The return types in the mcAM are removed and as void in the cSC

Explanation of case DO-5: According to the implementation requirements of the system, the architects might sometimes over-specify the return types in the mcAM. Yet, the developers disagree with that, and in the implementation of the cSC, they decide to remove some of those over-specified return types in the mcAM.

Example of case DO-5 (selected from project 2): As observed in Table 4.12, for the class *Control*, an operation is named *modifyPet()*. A return type *Boolean* specified for it from the mcAM is removed and as *void* in the cSC.

Table 4.12: Four corresponding examples of the cases DO-1, DO-2, DO-3 and DO-5.

Example No.	Case ID	Class	mcAM	cSC
1	DO-1	Pet	eat(food: FOOD)	
2	DO-2	Control	modifyPet(kit : Pet): Boolean	modifyPet(): void
3	DO-3	Control	modifyPet(kit: Pet): Boolean	modifyPet(): void
4	DO-5	Control	modifyPet(kit: Pet): Boolean	modifyPet(): void

4.3.7.6 Case DO-6* - One or more operations from classifier A in the mcAM are moved to classifier B in the cSC (related to the particular architectural patterns)

Explanation of case DO-6*: This case is about the developers' design decisions to deviate from the mcAM in the cSC in terms of the application of specific architectural patterns. As we know, different architectural patterns are applied to address specific concerns. The related classes thereby take on different roles. One possibility is that the work intended to be assigned to the class from the mcAM was reallocated to other classes in the cSC.

Example of case DO-6* (selected from project 2): As observed in the comparison between Figures 4.43 and 4.44, in the mcAM, an operation in the class *DAO*, e.g., *listUser(): Map*, is moved to the class *RaiseMeUp* as the corresponding operation, e.g., *listUsers(): Map<Integer, User>* in the cSC. This is because of the divergences of the cSC implementation from the mcAM design in terms of the MVC architectural pattern. To be specific, *Dao* is a Model class related to the database, and *RaiseMeUp* is a Controller class. The operation *listUser* from the mcAM is initially planned out, aiming at getting a list of user data from the dataset (as illustrated in Figure 4.44). Considering the application of the MVC in the project, that operation modeled from the mcAM should be contained in a Controller class, which is responsible for managing the data from the Model-related class rather than executing data operations inside the Model-related database. Thus, here a deviation comes up from the mcAM design to the cSC implementation regarding the usage of the MVC architectural pattern.

Note that the operation name in the mcAM and cSC is different here, i.e., *listUser()* and *listUsers()*. This is considered to be caused by CC. The operation name differences are illustrated in section 4.4. Besides, the return type in the mcAM and cSC is also different here, i.e., *Map<>* and *Map<Integer, User>*. This is considered to be caused by MA, which is illustrated in another case MO-5* (referring to subsection 4.3.3.5).

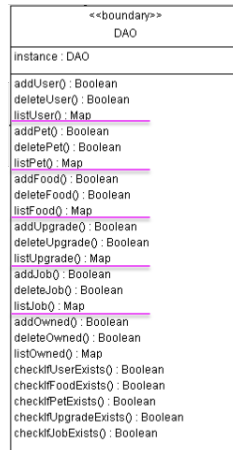


Figure 4.43: In the mcAM, for the class *DAO*, the operations, e.g., *listUser(): Map* is created.

```

public class RaiseMeUp {
    public static Map<Integer, User> listUsers() {
        Map<Integer, User> users = new HashMap<Integer, User>();
        try {
            /**
             * Responsible for interacting with Dao, a
             * Model class related to database.
             */
            users = dao.getUser();
        } catch (SQLException ex) {
            // Complete implementation not shown here.
        }
        return users;
    }
}
  
```

Figure 4.44: In the cSC, the operation, e.g., *listUser(): Map* created in the class *DAO* from the mcAM is moved to another class *RaiseMeUp/Controll* as *listUsers(): Map<Integer, User>*. Then it is used for getting a list of user data from the Model-related class *Dao* in the cSC.

4.3.8 Cases of disAGTs - Relationships

We concluded three cases of disAGTs in terms of relationships, namely Cases DR-1, DR-2, and DR-3* (with the corresponding causes). They are illustrated in Figure 4.45, thereby giving the reader an overview of the remainder of this subsection.

Notably, each case will be detailed separately in the following subsections. Also, the respective example of every cause is given for the illustration.

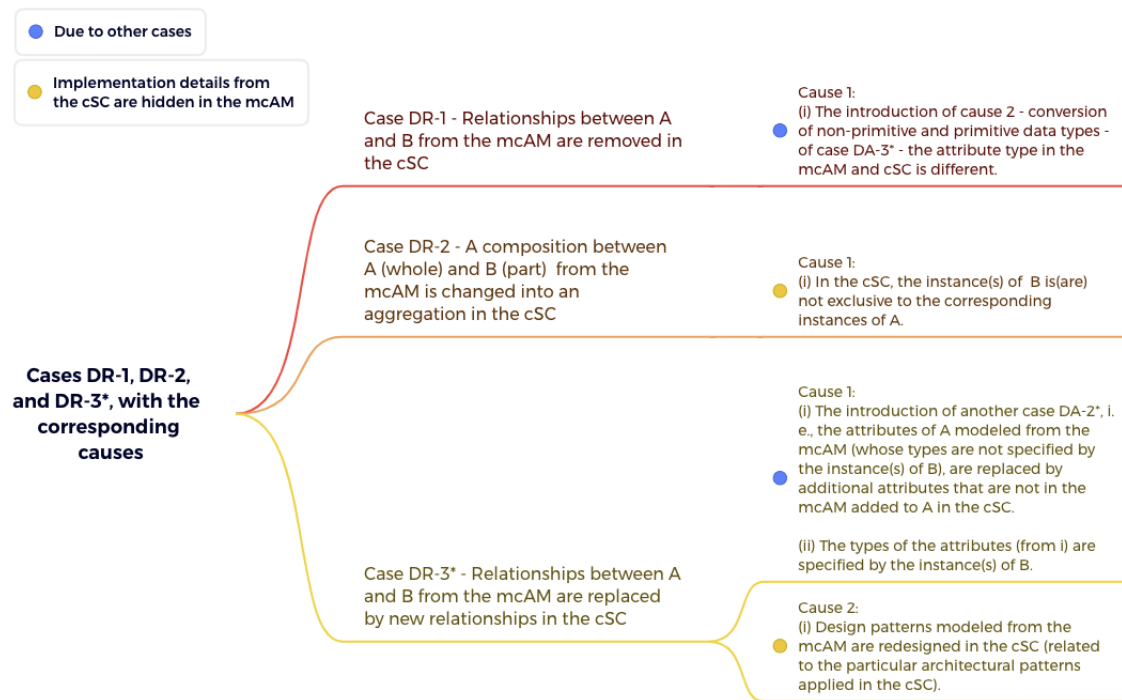


Figure 4.45: Cases DR-1, DR-2, and DR-3*, with the corresponding causes.

4.3.8.1 Case DR-1 - Relationships between A and B from the mcAM are removed in the cSC (one cause)

Explanation of case DR-1: This case is about the deviations between mcAM design and cSC implementation in terms of relationships. Here, it refers to removing the relationships from the mcAM in the cSC.

We concluded the following cause for this case, with the corresponding example to illustrate:

- **Cause 1:** (i) The introduction of cause 2 - conversion of non-primitive and primitive data types - of case DA-3* - the attribute type in the mcAM and cSC is different.

Explanation of cause 1: In the mcAM, an aggregation or a composition between A and B can be implied by an attribute type (specified by the instance(s) of B) in A. However, if this attribute type in the mcAM is converted to a primitive data type, such as *int* in the cSC, then the related link between A and B will be removed.

Example of cause 1 (selected from project 2): As observed in the comparison between Figures 4.46 and 4.47, for the attribute named *owner*, its type is converted from an instance of *User* in the mcAM to a primitive data type *int* in the cSC. This leads to the related composition

between the classes *Pet* and *User* from the mcAM being removed in the cSC.

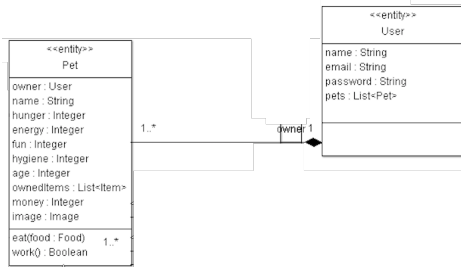


Figure 4.46: In the mcAM, a composition is created between the classes *Pet* and *User*.

```
public class Pet {
    /**
     * Attribute type changed into
     * an primitive data type, i.e., int
     */
    private int owner;
}
```

Figure 4.47: In the cSC, the attribute type whose type is specified by an instance of *User* from the mcAM is converted to a primitive data type *int*.

This leads to the corresponding composition between *Pet* and *User* from the mcAM being removed in the cSC.

4.3.8.2 Case DR-2 - A composition between A (whole) and B (part) from the mcAM is changed into an aggregation in the cSC (one cause)

Explanation of case DR-2: This case covers deviations between the mcAM design and cSC implementation in terms of changing the composition modeled from the mcAM into an aggregation in the cSC.

We concluded the following cause for this case, with a corresponding example to illustrate:

- **Cause 1:** (i) In the cSC, the instance(s) of B are not exclusive to the corresponding instances of A.

Explanation of cause 1: To distinguish aggregation from a composition is that the lifetime of B is not related to the lifetime of A. In other words, the instances of A are not *exclusive* to the corresponding instances of A, according to the source [40]. Thus, in this case, the instance(s) of B is(are) not contained by A but also by another one or more classes in the cSC.

Example of cause 1 (selected from project 5): Figure 4.48 illustrates that a composition is created between the classes *Layer* (whole) and *Neuron* (part) in the mcAM. Yet, as illustrated in Figure 4.49, the instances of *Neuron* are contained by not only *Layer* but also by another class *NeuralNetwork*. This means that the lifetime of *Neuron* is not related to the lifetime of *Layer*. An aggregation rather than that compo-

sition modeled from the mcAM is thereby inferred to exist between *Layer* and *Neuron* in the cSC.

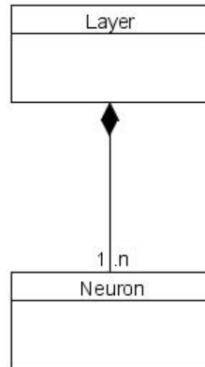


Figure 4.48: In the mcAM, a composition is modeled between the classes *Layer* and *Neuron*.

```

public class Layer implements Serializable {
    // Collection of Neuron instances declared here.
    protected NeurophArrayList<Neuron> neurons;

    public Layer() {
        // Collection of Neuron instances created here.
        neurons = new NeurophArrayList(Neuron.class);
    }
}

public class NeuralNetwork<L> extends LearningRule<L> implements Serializable {
    // Collection of Neuron instances declared here.
    private NeurophArrayList<Neuron> inputNeurons;

    public NeuralNetwork() {
        // Collection of Neuron instances created here.
        this.inputNeurons = new NeurophArrayList<>(Neuron.class);
    }
}
  
```

Figure 4.49: In the cSC, the instances of *Neuron* are contained by not only *Layer* but also *NeuralNetwork*.

4.3.8.3 Case DR-3* - Relationships between A and B from the mcAM are replaced with new relationships in the cSC (two causes)

Explanation of case DR-3*: This case covers deviations between mcAM design and cSC in terms of relationships, i.e., replacing the relationships modeled from the cSC with new relationships in the cSC.

We concluded the following two causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Cause 1: (i) The introduction of another case DA-2*, i.e., the attributes of A modeled from the mcAM (whose types are not specified by the instance(s) of B) are replaced by additional attributes that are not in the mcAM added to A in the cSC. (ii) The types of the attributes (from i) are specified by the instance(s) of B.

Explanation of cause 1: For the attribute type of A, which is specified by the instance(s) of B from the mcAM, either an aggregation or a composition can be hereby implied to exist in the cSC. This can be further confirmed based on the fact that these instances of B are created in an operation of A and further invoked within an operation of A.

Example of cause 1 (selected from project 3): Figure 4.50 illustrates that an association is created between the classes *CheckingAccount* and *IncomeRepository* in the mcAM. Yet, this association from the mcAM is replaced with an aggregation in the cSC instead. This is because the attributes created for *CheckingAccount* in the mcAM, e.g., *income: Income*, are replaced with fully new attributes, e.g., *incomeRepo: IncomeRepository* in the cSC (see Figure 4.51). Thereby, either an aggregation or

a composition is inferred to exist between *CheckingAccount* and *IncomeRepository* in the cSC based on the specified type of that attribute i.e., an instance of *IncomeRepository*. This instance is further created within the operation *CheckingAccount()* of *CheckingAccount*; plus, an operation *save()* of this instance is invoked within another operation *add(Income): void* of *CheckingAccount* in the cSC. Thereby, an aggregation can be inferred to exist between *CheckingAccount* and *IncomeRegister* in the cSC. In addition, the instances of *IncomeRepository* are also contained by another class *ValuesCalculator*. Thus, this aggregation is confirmed as not a composition.

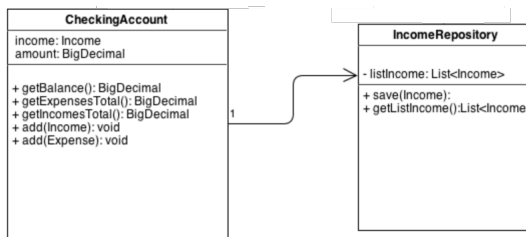


Figure 4.50: In the mcAM, an association is created between *CheckingAccount* and *IncomeRepository*.

```

public class CheckingAccount {
    /**
     * An instance of IncomeRepository
     * declared as an attribute type here.
     */
    IncomeRepository incomeRepo;
    ExpenseRepository expenseRepo;
    public CheckingAccount() {
        // An instance of IncomeRepository created here.
        incomeRepo = new IncomeRepository();
        expenseRepo = new ExpenseRepository();
    }
    public void add(Income income) {
        // save() of IncomeRepository invoked here.
        incomeRepo.save(income);
    }
}

public class ValuesCalculator {
    // Complete implementation not shown here.
    public float getIncomesTotal() {
        // An instance of IncomeRepository created here.
        IncomeRepository repository=new IncomeRepository();
    }
}
  
```

Figure 4.51: For *CheckingAccount*, the attributes from the mcAM, e.g., *income: Income*, are replaced with fully new attributes, e.g., *incomeRepo: IncomeRepository* in the cSC. The specified attribute type, an instance of *IncomeRepository*, is created within an operation *CheckingAccount()* of *CheckingAccount* in the cSC. This instance is further invoked within another operation *add(Income): void* of *CheckingAccount* in the cSC.

Furthermore, the instances of *IncomeRepository* are contained not only by *CheckingAccount* but also by another class *ValuesCalculator* in the cSC.

- **Cause 2:** (i) Design patterns modeled from the mcAM are redesigned in the cSC (related to the particular architectural patterns applied in the cSC).

Explanation of cause 2: In terms of design patterns, there are deviations between the mcAM design and cSC implementation. These deviations may be related to deviations from mcAM to cSC in terms of specific architectural patterns. Of particular note is that the design patterns may cooperate with the architectural patterns used to build the system in the cSC and address the specific system’s concerns. Specifically, the structural design patterns modeled from the mcAM are possibly redesigned to accommodate the actual architectural patterns applied in the cSC. Thus, deviations from a particular architectural pattern may affect the implementation of the design pattern in the mcAM. The relationships associated with that design pattern might be changed accordingly. Notably, a possibility is that concepts described by those classes related to the architectural patterns or-and architectural patterns from the mcAM and cSC are yet consistent. This means that although deviations exist in relationships from the mcAM to cSC, however, it is just about making different decisions for addressing the same concern.

Example of cause 2 (selected from project 2): As observed in Figure 4.52, an *observer* design pattern [17] can be implied based on the fact that the naming of the class *PetObserver* and the significant indicator of the observer design pattern, i.e., the operation *update(): void* of *PetObserver*; plus, the association relates to *PetObserver* to be linked with the class *Pet*. *Pet* is an observable class, and *PetObserver* is an observer class. An association is created between *Pet* (origin) *PetObserver* (target) in the mcAM.

Yet, in the cSC, that association from the mcAM should be reversed, i.e., an association exists between *Pet* (target) *PetObserver* (origin). This is possibly due to the particular MVC architectural pattern applied in the cSC. To be specific, *PetObserver* and *Pet* are Model classes, which are managed by the Controller class *RaiseMeUp* in the cSC. Figure 4.53 illustrates that the operation, e.g., *getCurrentPet()* of *RaiseMeUp* is invoked within an operation, e.g., *PetObserver()* of *PetObserver* in the cSC, and the operation *getEnergy()* of *Pet* is further invoked with the help of *RaiseMeUp*. Thereby, an indirect association between *PetObserver* (origin) and *Pet* (target) is built up, with *RaiseMeUp* in between.

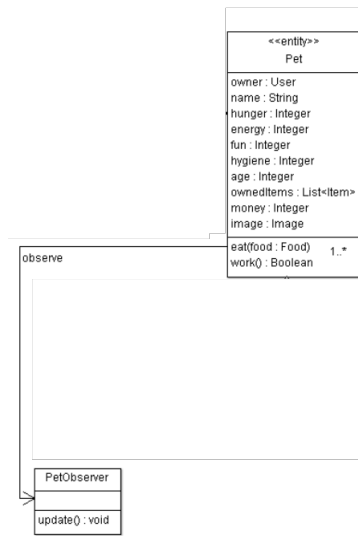


Figure 4.52: In the mcAMA, an association is created between *Pet* (origin) *PetObserver* (target).

```

public class PetObserver {
    public PetObserver() {
        /**
         * First, getCurrentPet() of RaiseMeUp invoked
         * by PetObserver.
         * Second, getEnergy() of Pet invoked by RaiseMeUp.
         */
        prevEnergy = RaiseMeUp.getCurrentPet().getEnergy();
    }
}

public class Pet {
    public int getEnergy() {
        return energy;
    }
}
  
```

Figure 4.53: *RaiseMeUp* (target) is linked with *Pet* (origin); *RaiseMeUp* (origin) is linked with *PetObserver* (target). Thus, an indirect association between *Pet* (origin) *PetObserver* (target) is built up.

4.4 Cases of the Differences Caused by CC

We concluded seven cases for the differences caused by CC in terms of the model elements, namely classes, attributes, and operations. These seven cases are illustrated in Table 4.13. Note that cases CA-2, CO-3, and CO-4 are related to data types where different classifiers (or their instances) interact.

The corresponding causes concluded for these seven cases are illustrated in Figure 4.54. Thereby, this table and figure provided can give the reader an overview of the remainder of this section. To be specific, each case will be detailed separately in subsubsections, with the corresponding examples(s) to explain.

Table 4.13: Seven cases of the differences caused by CC.

	Case ID	CC
Classes	CC-1	Naming of classes in the mcAM and cSC is different (three causes, see Figure 4.54)
Attributes	CA-1	Naming of attributes in the mcAM and cSC is different (two causes, see Figure 4.54)
	CA-2	Naming of attribute types in the mcAM and cSC is different (two causes, see Figure 4.54)
Operations	CO-1	Naming of operations in the mcAM and cSC is different (four causes, see Figure 4.54)
	CO-2	Naming of parameters in the mcAM and cSC is different (two causes, see Figure 4.54)
	CO-3	Naming of parameter types in the mcAM and cSC is different (one cause, see Figure 4.54)
	CO-4	Naming of return types in the mcAM and cSC is different (one cause, see Figure 4.54)

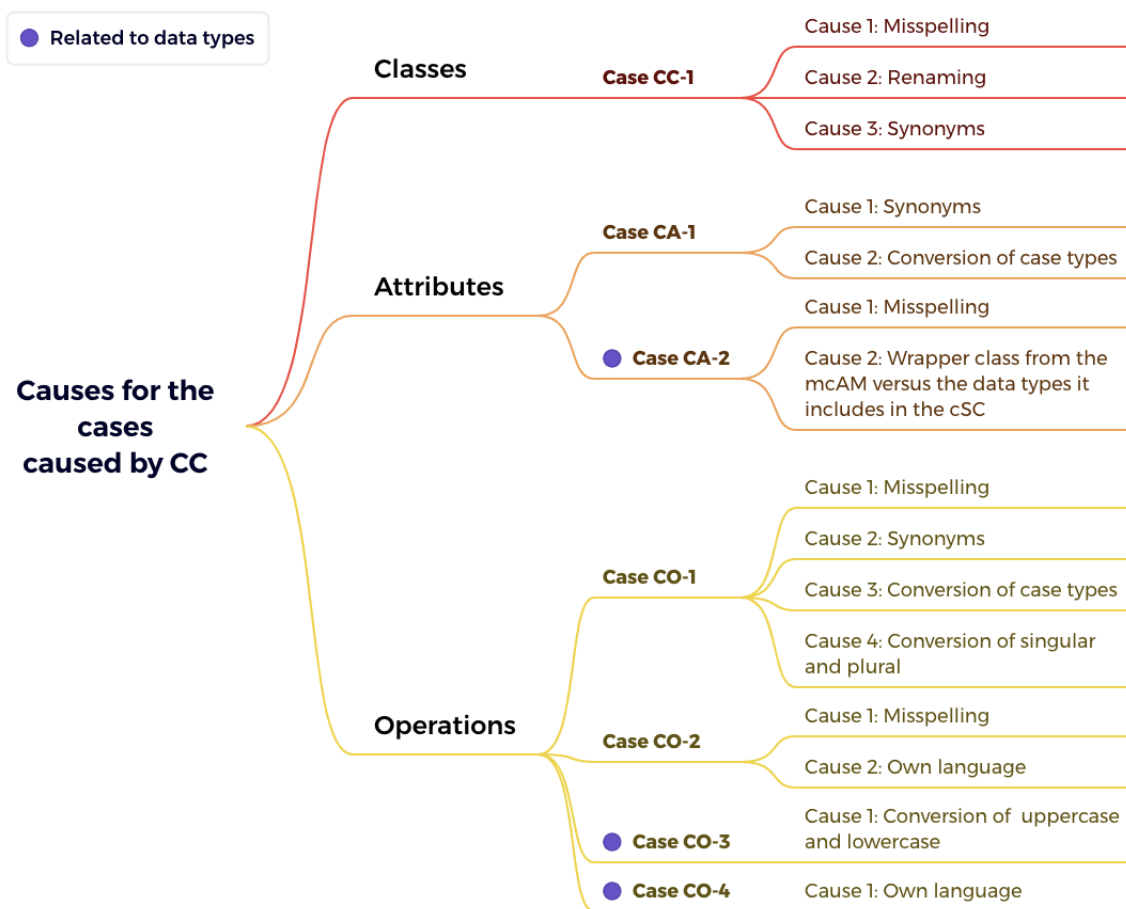


Figure 4.54: Causes for the cases caused by CC.

4.4.1 Cases of CC - Classes

4.4.1.1 Case CC-1 - Naming of classes in the mcAM and cSC is different (three causes)

Explanation of case CC-1: This case is related to common changes between mcAM and cSC in terms of class names.

We concluded the following three causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Misspelling.

Explanation of cause 1: Due to some typo errors, the class names are possibly misspelled in the mcAM. Yet, they got corrected in the cSC.

Example of cause 1 (selected from project 1): Referring to Table 4.14, it is observed that the class name *SinglePlayModel* from the mcAM is corrected as *SinglePlayerModel* in the cSC.

- **Cause 2:** Renaming.

Explanation of cause 2: Considering the semantics carried by classes, their names from the mcAM are possibly renamed in the cSC. This can be confirmed by checking the attributes and operations included in the class; also, the relationships related to this class with other classes.

Example of cause 2 (selected from project 2): As observed in Table 4.14, the class name *Control* from the mcAM is renamed as *RaiseMeUp* in the cSC. This is because this project applies a MVC design pattern, *Control* from the mcAM means a *Controller* class. Yet, it is purely renamed by the project's name, i.e., *RaiseMeUp*.

- **Cause 3:** Synonyms.

Explanation of cause 3: Plenty of synonyms exist in the dictionary. Out of individual preferences, the developers might replace the class names from the mcAM with synonyms in the cSC.

Example of cause 3 (selected from project 3): As observed in Table 4.14, the class name *IncomeRegisterUI* from the mcAM is replaced with a synonym *RegisterIncomeUI* in the cSC. They carry exactly the same semantics, a UI-related class.

Table 4.14: Three corresponding examples of the causes for case CC-1.

Example No.	Causes	mcAM	cSC
1	Misspelling	SinglePlayModel	SinglePlayerModel
2	Renaming	Control	RaiseMeUp
3	Synonyms	IncomeRegisterUI	RegisterIncomeUI

4.4.2 Cases of CC - Attributes

4.4.2.1 Case CA-1 - Naming of attributes in the mcAM and cSC is different (two causes)

Explanation of case CA-1: This case is related to common changes between mcAM and cSC in terms of attribute names.

We concluded the following two causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Synonyms.

Explanation of cause 1: Plenty of synonyms exist in the dictionary. Out of individual preferences, the developers possibly replace the attribute names from the mcAM with synonyms in the cSC.

Example of cause 1 (selected from project 2): Referring to Table 4.15, it is observed that the attribute name *name* from the mcAM is replaced with a synonym *username* in the cSC. They carry exactly the same semantics, a name.

- **Cause 2:** Conversion of case types.

Explanation of cause 2: The naming conversions in Java possibly cause the deviations between mcAM and cSC in terms of attribute names.

Example of cause 2 (selected from project 2): As observed in Table 4.15, the attribute name *valueDog* in camel-case from the mcAM is changed into *valuedog*, all letters in lowercase in the cSC.

Table 4.15: Two corresponding examples of the causes for case CA-1.

Example No.	Causes	Class	mcAM	cSC
1	Synonyms	User	name: String	username: String
2	Conversion of case types	Food	valueDog: Integer	valuedog: int

4.4.2.2 Case CA-2 - Naming of attribute types is different (two causes)

Explanation of case CA-2: This case is related to common changes between mcAM and cSC in terms of the naming of the attribute types. This can be only caused by unintentional human mistakes involved in the mcAM design. This is because the attribute types are code constructs; if there is any unintentional mistake involved in the cSC implementation process, these mistakes will be promoted by compilers and hereby be corrected by developers.

We concluded the following two causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Misspelling.

Explanation of cause 1: Due to some typo errors, the naming of the attribute types from the mcAM possibly involves misspelling. Yet, it got corrected in the cSC.

Example of cause 1 (selected from project 3): Referring to Table 4.16, it is observed that the naming of the attribute type *IncomeType* involves misspelling. This can be inferred by observing its related name *incomeTypePository*, which carries the semantics of the existence of a class *IncomeTypeRepository*. In the cSC, that attribute type *IncomeType* from the mcAM is corrected as *IncomeTypeRepository* in the cSC.

- **Cause 2:** Wrapper class from the mcAM versus the data types it includes in the cSC.

Explanation of cause 2: Referring to the definition of attribute type, an attribute type can be either a primitive data type or a non-primitive data type. Interestingly, the architects possibly model an attribute type as a wrapper class (a non-primitive data type), such as *Integer* rather than *int* (a primitive data type) included in *Integer*.

Example of cause 2 (selected from project 2): As observed in Table 4.16, the attribute type *Integer* from the mcAM is changed into *int* in the cSC.

Table 4.16: Two corresponding examples of the causes for case CA-2.

Example No.	Causes	Class	mcAM	cSC
1	Misspelling	RegisterIncomeController	incomeTypeRepository: IncomeType	incomeTypeRepository: IncomeTypeRepository
2	Wrapper class from the mcAM versus the data types it includes in the cSC	Pet	money: Integer	money: int

4.4.3 Cases of CC - Operations

4.4.3.1 Case CO-1 - Naming of operations in the mcAM and cSC is different (four causes)

Explanation of case CO-1: This case is related to common changes between mcAM and cSC in terms of operation names.

We concluded the following four causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Misspelling.

Explanation of cause 1: Due to some typo errors, the operation names are possibly misspelled in the mcAM. They got corrected in the cSC.

Example of cause 1 (selected from project 1): Referring to Table 4.17. The operation name *SinglePlayModel* from the mcAM is corrected as *SinglePlayerModel* in the cSC.

- **Cause 2:** Synonyms.

Explanation of cause 2: Plenty of synonyms exist in the dictionary. Out of individual preferences, the developers might replace the operation names from the mcAM with synonyms in the cSC.

Example of cause 2 (selected from project 4): Referring to Table 4.17, the operation name *getNumberOfHours* from the mcAM is replaced with a synonym *getNoOfHours* in the cSC. They carry exactly the same semantics, getting the number of hours.

- **Cause 3:** Conversion of case types.

Explanation of cause 3: The naming conversions in Java possibly cause the deviations between mcAM and cSC in terms of operation names.

Example of cause 3 (selected from project 1): Referring to Table 4.17, the operation name *setbackground*, all letters in lowercase from the mcAM is changed into *setBackground* in camel-case in the cSC.

- **Cause 4:** Conversion of singular and plural.

Explanation of cause 4: The conversion of singular and plural carries semantics about the conversion of the concepts of one and more entities/instances.

Example of cause 4 (selected from project 2): As observed in Table

4.17, the operation name *listFood* from the mcAM carries the semantics of one *Food* instance. Yet, it is exactly a collection of *Food* instances based on the fact that the related return type *Map*, a collection-related interface, is used to specify that operation. That singular name *listFood* from the mcAM is corrected as a plural name *listFoods* in the cSC to make the semantics carried by the name consistent with the semantics reflected by the related return type.

Table 4.17: Four corresponding examples of the causes for case CO-1.

Example No.	Causes	Class	mcAM	cSC
1	Misspelling	SinglePlayModel	<code>SinglePlayModel(diff: States.difficulty, am: AssetManager, wordsDis: int)</code>	<code>SinglePlayerModel(diff: States.difficulty, am: AssetManager, wordsDis: int)</code>
2	Synonyms	Pattern	<code>getNumberOfHours()</code>	<code>getNoOfHours(): Integer</code>
3	Conversion of case types	PreGameSelection	<code>setBackground(view: View)</code>	<code>setBackground(view: View): void</code>
4	Conversion of singular and plural	DAO	<code>listFood(): Map</code>	<code>listFoods(): Map<Integer, Food></code>

4.4.3.2 Case CO-2 - Naming of parameters in the mcAM and cSC is different (two causes)

Explanation of case CO-2: This case is related to common changes between mcAM and cSC in terms of parameter names.

We concluded the following two causes for this case, with the corresponding examples to illustrate each of these causes:

- **Cause 1:** Misspelling.

Explanation of cause 1: Due to some typo errors, the parameter names are possibly misspelled in the mcAM. They are corrected in the cSC.

Example of cause 1 (selected from project 1): As observed in Table 4.18, the parameter name *savedInstaceState* from the mcAM is corrected as *savedInstanceState* in the cSC.

- **Cause 2:** Own language.

Explanation of cause 2: Interestingly, we observed that the architects might use pronouns to describe the related parameter type. This is not caused by MA and disAGTs; rather, we consider it to come from individual preferences. Yet, the usage of pronouns might be too comprehensive and a little imprecise in describing a parameter type.

Example of cause 2 (selected from project 2): Referring to Table 4.18, the parameter name *who*, a pronoun, from the mcAM is changed into *u* in the cSC. They carry exactly the same semantics, but this can only be inferred by the same parameter type, namely *User* related to them. Both parameter types are comprehensive.

Table 4.18: Two corresponding examples of the causes for case CO-2.

Example No.	Causes	Class	mcAM	cSC
1	Misspelling	SinglePlayer	onCreat(savedInstanceState : Bundle)	onCreat(savedInstanceState : Bundle): void
2	Own language	Control	eraseUser(who : User): Boolean	removeUser(u : User): Boolean

4.4.3.3 Case CO-3 - Naming of parameter types in the mcAM and cSC is different (one cause)

Explanation of case CO-3: This case is related to common changes between mcAM and cSC in terms of the naming of parameter types.

We concluded the following cause for this case, with the corresponding example to illustrate:

- **Cause 1:** Conversion of uppercase and lowercase.

Explanation of cause 1: It might be common to have the conversion of uppercase and lowercase regarding the names. And we observed the following example.

Example of cause 1 (selected from project 1): Referring to Table 4.19, the naming of the parameter type *Char* with the initial capitalization from the mcAM is converted to *char* with all of its letters in lowercase in the cSC.

Table 4.19: The corresponding example of the cause for case CO-3.

Example No.	Causes	Class	mcAM	cSC
1	Conversion of uppercase and lowercas	SinglePlayModel	typedLetter(letter: Char)	typedLetter(letter: char): void

4.4.3.4 Case CO-4 - Naming of return types in the mcAM and cSC is different (one cause)

Explanation of case CO-4: This case is related to common changes between mcAM and cSC in terms of the naming of return types.

We concluded one cause, i.e., own language, for this case. Yet, this cause is particularly interesting. Thus, we use the following two examples to illustrate this cause.

- **Cause 1:** Own language.

Explanation of cause 1: We observed that the architects might sometimes use their own language to convey semantics. In this way, the structure of the system related to the linkage between different classifiers (or

their instances) can be presented in the mcAM.

Example 1 of cause 1 (selected from project 3): As observed in Table 4.20, the return type *list* from the mcAM is changed into *List<IncomeType>* in the cSC. *list* is neither a primitive data type nor a non-primitive data type. It conveys the semantics defined by architects. To be specific, in the mcAM, an attribute *list*: *List<IncomeType>* is created in the same class *IncomeTypeRepository* where the operation *getIncomeTypes()*: *list* is contained. The name of that attribute i.e., *list*, is used to specify the return type of the operation named *getIncomeTypes* in the mcAM. In this way, the linkage between the classes *IncomeTypeRepository* and *IncomeType* can be hereby presented in the mcAM.

Example 2 of cause 1 (selected from project 3): Referring to Table 4.20, it is observed that the return type *IncomeTypes* from the mcAM is changed into *List<IncomeType>* in the cSC. According to the specification of *List<IncomeType>*, we can know that *IncomeType* is a class; also, a collection of this class is referenced by the instances of *RegisterIncomeController* in the cSC. Yet, in the mcAM, the return type is specified by a plural string. This is used to convey the semantics that a collection of instances of *IncomeType* is possibly referenced by the instances of *RegisterIncomeController* in the cSC.

Table 4.20: Two corresponding examples of the cause for case CO-4 (* = particular interest).

Example No.	Causes	Class	mcAM	cSC
1	Own language*	IncomeTypeRepository	getIncomeTypes(): list	getIncomeTypes(): List<IncomeType>
2		RegisterIncomeController	getincomeTypes(): IncomeTypes	getIncomeTypes(): List<IncomeType>

4.4.4 Ratios of Cases with Related Projects

Table 4.21 presents the ratios of the cases of MA, disAGTs, and CC discovered among the five projects studied, with the related projects. It is observed that we finally concluded 40 cases. Among these cases, 18 are caused by MA, 15 are caused by disAGTs, and 7 are caused by CC. For MA, 4 of 18 are their own cases. For disAGTs, 7 of 15 are their own cases. For the other respective 14 and 8 cases of MA and disAGTs, we observed that some of them are opposite to each other. One concern is that if one of the opposite paired cases of MA and disAGTs has not been observed, that unobserved case can be inferred by the other observed. We named such inferred cases as suspected cases, which is presented in Table 4.2.

Table 4.21: Ratios of the cases with involved projects (* = own case).

No. of cases	ID					
	MA	Project	disAGTs	Project	CC	Project
1	MC-1	5	DC-1	2	CC-1	1, 2, 3, 5
2	MC-2	2	DA-1	2	CA-1	1, 2, 4
3	MA-1	1, 2, 3, 4	DA-2*	3	CA-2	2, 3
4	MA-2*	4	DA-3*	2	CO-1	1, 2, 4
5	MA-3*	4	DA-4*	1	CO-2	1, 2
6	MA-4	2	DA-5*	1	CO-3	1
7	MA-5	2	DO-1	2, 4	CO-4	2, 3
8	MA-6	2	DO-2	2		
9	MO-1	1, 2, 3, 4	DO-3	2		
10	MO-2	1, 2, 3, 4	DO-4*	2		
11	MO-3	2, 3, 4	DO-5	2		
12	MO-4	1, 2, 3, 4	DO-6*	2		
13	MO-5*	2	DR-1	2		
14	MO-6*	1	DR-2	5		
15	MO-7	2	DR-3*	2, 3		
16	MR-1	1, 2, 3, 4, 5				
17	MR-2	3, 4				
18	MR-3	1				
Total no.	18		15		7	40

4.4.5 Typical/Common Cases with Related Projects

We define that cases involved in three or above three projects are typical/common cases. A sorted list of 9 typical/common cases with the related projects is presented in Table 4.22. Of particular note, there is no typical/common case caused by disAGTs. Among these 9 cases, 6 cases and 3 cases are related to MA and disAGTs, respectively.

For MA, it is common for the additional attributes and operations not in the mcAM yet to be added to the cSC (in regard to the respective cases MA-1 and MO-1). With respect to cases MO-2 and MO-3, it is common for the additional parameter names and types not in the mcAM yet to be added to the cSC. One concern here is that they might be paired up to be added to the cSC since the parameter name is the identifier of the parameter type. Surprisingly, the return types are also common for not specified in the mcAM yet added to the cSC (in regard to case MO-4). Of particular interest, case DR-1 is involved in every project we studied. To be specific, additional relationships not in the mcAM are added to the cSC, which is quite common. Moreover, for CC, it is common to have differences in the naming of classes, attributes, and operations between the mcAM and cSC (in regard to respective cases CC-1, CA-1, and CO-1).

Table 4.22: Typical/common cases with related projects.

No. of cases	ID			
	MA	Project	CC	Project
1	MA-1	1, 2, 3, 4	CC-1	1, 2, 3, 5
2	MO-1	1, 2, 3, 4	CA-1	1, 2, 4
3	MO-2	1, 2, 3, 4	CO-1	1, 2, 4
4	MO-3	2, 3, 4		
5	MO-4	1, 2, 3, 4		
6	MR-1	1, 2, 3, 4, 5		
Total No.	6		3	9

4.4.6 Aggregated Results of the Cases for the Project

For the five projects studied, the respective list of sorted cases of MA, disAGTs, and CC are presented in the following:

Table 4.23 illustrates the sorted list of cases of MA, disAGTs, and CC for project 1. It is observed that there are 12 cases involved in project 1, and among these cases, 6 cases of MA, 2 cases of disAGTs, and 4 cases of CC.

Table 4.23: Project 1 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).

No. of cases	Case ID		
	MA	disAGTs	CC
1	MA-1	DA-4*	CC-1
2	MO-1	DA-5*	CA-1
3	MO-2		CO-1
4	MO-4		CO-2
5	MO-6*		
6	MR-3		
Total no.	6	2	4

Table 4.24 presents the sorted list of cases of MA, disAGTs, and CC for project 2. It is observed that there are 29 cases involved in project 2, and among these cases, 12 cases of MA, 11 cases of disAGTs, and 6 cases of CC.

Table 4.24: Project 2 - Respective differentiated cases of MA, disAGTs, and CC
(* = own case).

No. of cases	Case ID			
	MA	disAGTs	CC	
1	MC-2	DC-1	CC-1	
2	MA-1	DA-1	CA-1	
3	MA-4	DA-3*	CA-2	
4	MA-5	DO-1	CO-1	
5	MA-6	DO-2	CO-2	
6	MO-1	DO-3	CO-4	
7	MO-2	DO-4*		
8	MO-3	DO-5		
9	MO-4	DO-6*		
10	MO-5*	DR-1		
11	MO-7	DR-3*		
12	MR-1			
Total no.	12	11	6	29

Table 4.25 presents the sorted list of cases of MA, disAGTs, and CC for project 3. It is observed that there are 12 cases involved in project 3, and among these cases, 7 cases of MA, 2 cases of disAGTs, and 3 cases of CC.

Table 4.25: Project 3 - Respective differentiated cases of MA, disAGTs, and CC
(* = own case).

No. of cases	Case ID			
	MA	disAGTs	CC	
1	MA-1	DA-2*	CC-1	
2	MO-1	DR-3*	CA-2	
3	MO-2		CO-4	
4	MO-3			
5	MO-4			
6	MR-1			
7	MR-2			
Total no.	7	2	3	12

Table 4.26 presents the sorted list of cases of MA, disAGTs, and CC for project 4. It is observed that there are 12 cases involved in project 4, and among these cases, 9 cases of MA, 1 case of disAGTs, and 2 cases of CC.

Table 4.26: Project 4 - Respective differentiated cases of MA, disAGTs, and CC (* = own case).

No. of cases	Case ID		
	MA	disAGTs	CC
1	MA-1	DO-1	CA-1
2	MA-2*		CO-1
3	MA-3*		
4	MO-1		
5	MO-2		
6	MO-3		
7	MO-4		
8	MR-1		
9	MR-2		
Total no.	9	1	2

Table 4.27 presents the sorted list of cases of MA, disAGTs, and CC for project 5. It is observed that there are 4 cases involved in project 5, and among these cases, 2 cases of MA, 1 case of disAGTs, and 1 case of CC.

Table 4.27: Project 5 - Respective differentiated cases of MA, disAGTs, and CC.

No. of cases	Case ID		
	MA	disAGTs	CC
1	MC-1	DR-2	CC-1
2	MR-1		
Total no.	2	1	1

5

Discussion

In this chapter, the reflection on the methodology of data selection is presented first, followed by the discussion in regard to the limitations of the reverse engineering tools and the advantages of manual studies. The guideline enlightened by the cases that we have discovered is detailed and discussed. Last but not least, the related technologies that can automatically heal the discovered cases are discussed. Finally, the identified threats to the validity of this thesis work are presented.

5.1 Reflection on Data Selection

In the data selection, we assumed that the voSC that includes the selected mcAM is most likely to be the cSC of that mcAM since the voSC and that mcAM are created at the same time. This assumption can be considered reliable based on the fact that in regard to the five projects we studied, for each of them, the voSC that includes the mcAM is proven to be a cSC of that mcAM. Considering that no study exists on how to select the modeled code for the model in regard to GitHub open-source projects, our assumption can thereby be taken by other researchers as a good starting point for selecting the modeled code (cSC) of the model (mcAM). Accordingly, a number of questions about the models and modeled code can thereby be answered, e.g., for the GitHub projects which use models, whether the modeled code are likely to be created before or after the created model in the commit history.

We did not encounter a situation in which the voSC that includes the selected mcAM is not a cSC of the mcAM. However, this situation cannot be excluded. Thereby, we proposed two reasons for this situation, with the corresponding cases of these two reasons to illustrate this situation (in section 3.2.3). We point out that the comparison between mcAM concepts and cSC classes in terms of number is a significant indicator for respective decisions made on comparing the voSCs before or after that voSC (that includes the selected mcAM) for selecting an ideal cSC among multiple cSCs of a mcAM. Nevertheless, these two proposed reasons and their corresponding cases need to be validated by the actual case in the future.

5.2 Limitations of Reverse Engineering Tools

In fact, in the beginning, we intended to adopt an automatic reverse engineering tool to visualize the code. Then we considered once the one-to-one mappings from the mcAM to the cSC are created in terms of classes, attributes, operations, and even relationships, the related differences can hereby be detected by our tool. Obviously, purely relying on an automatic reverse engineering tool to detect the differences is not an ideal approach, given the fact that the differences caused by MA, disAGTs, and even CC in terms of classes, attributes, operations, and relationships cannot be fully detected. Taking cause 1 of case CO-4 (in subsubsection 4.4.3.4) as an example, the architects might use their “own language” rather than the syntax of the model elements defined in UML to convey the semantics of the return type. Yet, the conveyed semantics can only be fully and precisely interpreted by a full understanding of the cSC implementation. That means we need to fully understand the primary roles taken over by the classes and the functionalities of every related attribute and operation associated with the classes, and especially the inter-relations between the attributes and operations in the cSC. This is because the relationships are highly dependent on the design of the attributes and operations. For instance, a composition or an aggregation between classifiers A (whole) and B (part) must be built on the fact that the instances of B are created in A.

Tools can help us identify some structural differences through visualization, but the generated reverse-engineered class diagrams cannot reveal other non-structural differences. This is caused by the fact that they cannot take the semantics conveyed by the mcAM elements into account and thereby automatically create one-to-many or many-to-many mappings between the mcAM and cSC. This can only be achieved by manual studies, which can jointly interpret the semantics conveyed by different model/mcAM elements. Based on our complete understanding of the relevant cSC implementation, the selected tool (IDEA) is considered only an aid for us to turn the relevant reverse-engineered class diagrams generated over the code into pictures. These pictures are used to illustrate the examples of the cases. The source code is always taken as a baseline to identify the differences between the mcAM and the cSC of that mcAM.

One point that can be taken from here is that only the classes of the cSC related to the mcAM concepts are essential to be visualized for studying the characteristics of the differences. This step is done by manual selection. Considering that there might be a large number of classes in a project, the unrelated classes’ source code files do not need to be visualized, which even hinders the capture of class information. Moreover, if a project applies a particular architectural pattern, knowing about it is a plus in the file selection. This is because the related classes’ source code files that are respectively differentiated into the corresponding directory/package of the architectural pattern can be quickly

located, and thereby selected to be visualized by the tool.

5.3 Advantages of Manual Mappings

As mentioned in section 5.2, compared with using a tool, one advantage of manual studies/mappings is jointly taking the semantics conveyed by different model/mcAM elements into account and thereby one-to-many, and many-to-many mappings between the mcAM elements and the cSC constructs can be created. The differences between the mcAM and cSC can thereby be fully and precisely detected. Another advantage is that we can scale the mappings up to a larger blueprint, with the consideration of the specific architectural patterns and design patterns applied in the projects. Of particular note, the architectural patterns and design patterns might sometimes cooperate to address the particular system's implementation concern. While few cases in our results (e.g., cases MC-2 and DO-6*) are related to their usage, we suggest manual studies on the characteristics of manual abstraction need to take the architectural patterns or-and design patterns used in the project into account. Some related similarities can thereby be discovered in regard to manual abstraction.

5.4 Guideline Enlightened by Cases

Adapting the architectural and design patterns to solve the particular system's concerns. As observed from the discovered MA cases, if a project applies a particular design pattern or-and architectural pattern, this will increase the difficulty of modeling the related model elements from the cSC out in the mcAM. This is because the particular design pattern or-and architectural pattern may cooperate to address the particular system's implementation concern. For that reason, the semantics conveyed by the related model elements need to be jointly considered. This means how to adapt the design pattern and architectural pattern to each other to address the same concern/goal. Of particular note here, we should not be constrained by the fixed structure of a particular design pattern frequently used in a model design or even purely naming a key class to represent the usage of a particular design pattern in the project (referring to case MC-2 in subsection 4.3.1.2). We suggest that the model design in regard to a particular design pattern or-and an architectural pattern should be based on a full understanding of the system's implementation structure and knowing about the concern needed to be solved. Then focusing on addressing that concern, the design pattern or-and architectural pattern is designed accordingly to adapt to that concern.

Being proactive in specifying model constituents related to data types. As we know, the relationships of association, aggregation, and composition between the classifiers are dependent on three model constituents related to data types (i.e., attribute types, parameter types, and return types). Furthermore, the conversion of primitive and non-primitive data types in regard to those three

model constituents will lead to the removal or addition of the related relationships. Thereby, when it comes to modeling these data types-related model constituents in a model, we need to be proactive in the potential circumstances that will be triggered (e.g., referring to cause 2 of case DA-3* in subsubsection 4.3.6.3).

Following the concepts of OO paradigm. In essence, model design and Java code implementation are related to the conceptualization of abstraction. Thus, a good design must follow the concepts of OO paradigm, such as *encapsulation* and *inheritance*. This can allow the design decisions made in the model prone to be accepted by developers in the code implementation. The abstraction concept of design and implementation can thus be synchronized. Referring to the example of case MA-5 (in subsubsection 4.3.2.5), we are not convinced that the common attributes are modeled in the respective subclasses rather than moving them into the superclass inherited by these subclasses in a hierarchical chain is a good design. This is because code reuse is not achieved. Code reuse not achieved also adapts to another possible case not observed, i.e., common operations are modeled in the respective subclasses rather than moving them into the superclass inherited by these subclasses in a hierarchical chain.

Trade-off between over-specification and over-abstraction in regard to hierarchical inheritance structure. We observed that a case, namely case MC-1 (in subsubsection 4.3.1.1), covers MA that hides the subclasses from the cSC in the mcAM and only models the superclass inherited by these subclasses from the cSC out in the mcAM. Accordingly, this modeled superclass in the mcAM can describe the respective second concepts of the concepts described by one or more subclasses derived from that superclass in the cSC. However, on the contrary, we also observed another case caused by disAGTs, namely case DC-1 (in subsubsection 4.3.5.1), where the over-specified subclasses from the mcAM are removed in the cSC, and thereby the corresponding inheritance chain is removed as well. Thus, for designing a hierarchical inheritance structure in the model, one concern is that we would ideally consider whether the inheritance structure is necessary and whether the commonalities of the objects of the subclasses can be differentiated into an object of a superclass. Before the related abstract concepts are formed, it would be ideal to stay away from over-specifying subclasses. This is not only poorly abstracted but also possibly misleading the developers in the code implementation.

Paying attention to the naming conversions. Of particular interest, we should pay attention to the naming convention of a constant variable (named by letters all in uppercase) and a variable (in camel case) in a model design. This will have an impact on specifying the related default values. That means for a variable created in the model, there is no need to assign a default value to it. This is because the default value is likely to be updated frequently in the code implementation. Yet, in regard to creating a constant variable in a model, this might depend on different decision-making by the architect. To be specific,

a constant variable can only be initiated once across the program life cycle. Thus, if the architect wants to emphasize the value of the constant variable, then the corresponding default value will be assigned accordingly.

Rethinking the model design of relationships. In [40], Guéhéneu and Albin-Amiot refined the definitions for relationships of association, aggregation, and composition. This provides invaluable input for us to identify the related relationships in the cSC from the mcAM. The causes for the deviations of the cSC from mcAM in terms of relationships can thereby be concluded. From the concluded causes, we found that in some cases, an association between A (origin) and B (target) in the cSC is possibly already implied by the parameter types or/and return types specified by the instance(s) of B within A. This is because they enable the instance of A to send a message to an instance of B (given the definitions by Guéhéneu and Albin-Amiot). The same to an aggregation or a composition between A (whole) and B (part) that can possibly be implied by the attribute types (specified by the instance(s) of B) within A body. Yet, those associations or aggregations are not modeled out in some cases. We cannot exclude the possibility that this might be due to different mcAM design decision-making. Yet our concluded causes for why some associations and aggregations not in the mcAM are added to the cSC could provide input for the architect to think about whether it is necessary for these implicit relationships to surface in the model design for a particular system implementation. In addition, considering the maintenance burden imposed by the cyclic relations as they are mutually binding, whether there is a need to refactor the implementation structure.

5.5 Related Technologies for Healing the Cases

In regard to the mapping of classes between code and design, Antoniol *et al.* [32] discovered that the class name is the class property that performs best among the other properties (e.g., class methods and class fields). Dennnis *et al.* [43] suggested that the package information would help limit the space for searching the related class differentiated into the specific directory. Thereby, the corresponding class can be easily located. In accordance with our manual mappings between model and code in terms of classes, we agree with the opinion of Dennnis *et al.*, i.e., package information does help the selection of the related class. Thus, with package information as a plus, one can use the automatic method [32] developed by Antoniol *et al.* to create the mapping from the classes between the code and design. Thereby, the left classes that are not successfully matched can be mapped manually.

Model-Driven Engineering (MDE) offers an approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [53]. Plus, one can use Object Constraint Language (OCL) to specify precise and unambiguous constraints on model elements, which can be used to validate, analyze, and transform software sys-

tems [54, 55]. Thus, the joint usage of MDE and OCL allows the code to be automatically generated by the model with the validation for the required system's implementation. This can also help to reduce the risk of errors and inconsistencies that can arise when generating code manually [56], and allows the generated code to be fully consistent with the model. Thereby, the deviations of the code from the model can be fully automatically eliminated. However, this requires a highly refined model design.

Software reflexion models [57] can be used to conduct the consistency checks between design and code in terms of relationships, built on the assumption that the mapping exists between design and code. With the code and design as a basis, the mappings can be defined manually, which serves as the input of the reflection models. A reflexion model can thus be created, and the “absence” and “divergence” relationships of the code from the design can further be detected [57]. Noteworthy, we accept the refinement of the definitions of the binary relationships of association, aggregation, and composition by Guéhéneu and Albin-Amiot [40]. Based on this, we identified the relationships that exist in the cSC and then manually created the mappings between the code (cSC) and design (mcAM). Thereby, the cases we summarized for MA and disAGTs in terms of relationships can provide such input for a software reflexion model. We agree with the assertion by Guéhéneu and Albin-Amiot [40] that the gap between model design and code implementation will be bridged once the consensus definitions of the relationship between model design and code implementation can be guaranteed.

For the slightly varying naming in the code from the model in terms of the model constituents of class name, attribute name, and operation name, a Java parser related to the abstract syntax tree (AST) can be used to locate the relevant places of the code and further generate a tree. The name of related code constructs can be changed accordingly, and the tree-represented code can automatically be adapted to the model. However, from another perspective, since AST cannot accept non-formalized syntax, if a model constituent is defined by “own language” to convey the semantics, that constituent cannot be parsed and is further matched with the parsed related code constructs in formalized syntax (an example with reference to the cause 1 of case CO-4 in subsection 4.4.3.4). Interestingly, if that non-formalized syntax defined by “own language” do help software engineers to understand the implementation structure by using such a related model, is there a need to enable the code and model to synchronize would be a good inquiry, given the usage of drawing.

5.6 Threats to Validity

In [58], Gren proposed a set of checklists regarding the existing four types of validity threats (i.e., internal, external, construction, and conclusion). We referred to these checklists and then categorized these identified threats into three types: threats to construction validity, threats to external validity, and

conclusion validity. Each of these threats is detailed in this section.

5.6.1 Threats to Construction Validity

Own subjective definitions of the “differences”. Individuals might have their own definitions of the “differences” between the mcAM and the cSC of that mcAM. Yet, we considered this due to the lack of systematic studies of the mcAM elements/constituents definitions. Thereby, it demands us to give clear definitions for the constituents of attributes and operations being studied in the mcAM. We take the UML v2.4.1 specifications [21] as the basis to define every constituent in the mcAM. By referring to the Java SE7 specifications [59] that is published at the approximate time of that UML v2.4.1 specifications [21], the mappings between the mcAM elements/constituents and cSC constructs can be created precisely. This is because we exclude the threat that the latest UML specifications might not adapt to the mcAMs we studied. In consequence, this threat to the construction validity can be eased.

Underestimation of the differences caused by MA and disAGTs. Considering the data selection in section 3.2, if multiple cSCs correspond to one mcAM, only one cSC is selected among those cSCs. We cannot ensure that the selected cSC is without any missed attributes and operations that will be implemented in other voSCs. This will lead to results that underestimate the differences caused by MA and disAGTs. However, considering the time constraints and the size of the projects, if a project has hundreds or thousands of voSCs, it is not possible for us to check the voSC one by one to confirm which of them are cSCs. However, we argue that the perfect selection regarding the cSC does not exist. This is because, still, the possibility that the missed attributes and operations in a particular cSC will be implemented in other voSCs/cSCs, cannot be excluded. That means the differences not investigated in the selected cSC might exist in other voSCs/cSCs and vice versa. Thereby, the summarized causes caused by MA in this thesis can provide valuable inputs for future mapping rules development between the model/mcAM and the modeled code/cSC. On the other hand, the concluded causes caused by disAGTs can enable software engineers to know which design decisions of the mcAM are prone to deviate in the cSC.

5.6.2 Threats to External Validity

Coverage of project domain types. This thesis focuses only on Java domain projects. However, the investigation of the differences between the mcAM and the cSC of that mcAM has scaled the cSC up to its related architectural pattern. Thereby, the reliability of our results can be guaranteed as the differences, to some extent, are related to the specific architectural patterns applied in the project. On the other hand, two of the five projects selected for study can be confirmed as coming from the industry. The remaining three projects are uncertain (due to the lack of the project’s information in the repository).

However, to ensure the generalization in the data selection process, the relevant information of each project that can be found in the repository has been manually gone through. This is intended to avoid, as much as possible, these five projects coming from academia, such as teaching materials, thesis, homework, etc. For the causes of MA, disAGTs, and CC, their cases are concluded by employing five case studies. Thus, this thesis work is considered a qualitative analysis. We are convinced that this qualitative analysis provides valuable inputs for developing future mapping rules between the model/mcAM and the modeled code/cSC in terms of Java.

5.6.3 Threats to Conclusion Validity

Limitation of the Lindholmen dataset [1]. Five thesis study subjects are yielded from this dataset. As described in section 3.1, this dataset is a set of open-source projects collected from GitHub with the UML models used in these projects. The model files are limited to image formats (.jpeg, .png, .gif, .svg, and .bmp) and standard formats (.xmi, and .uml files) without considering the models embedded in pdf, powerpoint, etc [1]. One concern is that we only studied the class diagrams in image formats, not for standard formats, even the other formats e.g., pdf, not covered by the Lindholmen dataset. Yet, given the limitations of manual studies, it is not ideal for us to study models in formats other than images. Taking .xmi as an example, the information in the .xmi file can be too extensive, making it hard for us to abstract the key elements/constituents of the model/mcAM and then create the mappings between these elements and the constructs of the modeled code/cSC. Thereby, we are convinced that image format models are the best choice for manual studies.

6

Conclusion and Future Work

In this chapter, this thesis work is concluded first, followed by suggested extensions to this thesis work.

6.1 Conclusion

In most cases, the existing reverse engineering tools/methods can not reverse code into class diagrams with abstraction. This is due to the lack of input in regard to the characteristics of manual abstraction. Yet, to investigate the characteristics of manual abstraction, we need to interpret the semantics conveyed by the model elements. This must build on the full understanding of the relevant code implementation, especially given that different systems have their own required implementation structure. Thus, to study the characteristics of manual abstraction with taking the semantics conveyed by the model elements into account, a systematic manual study of the actual case is required.

To fill this gap, we manually studied five Java projects in regard to the differences between the model (mcAM) and the modeled code (cSC). The manual mappings are created between the mcAM elements/constituents and cSC constructs in terms of model elements of classes, attributes, operations, and relationships. As a result, we concluded 18 MA-related cases, which can be used as the input for future improvement of the reverse-engineering methods/tools in terms of abstraction. Besides, another two causes for the differences between the mcAM and cSC were also found in the manual mapping process, namely disAGTs, and CC. In consequence, this thesis concluded three primary causes for the differences between the model (mcAM) and modeled code (cSC), namely MA, disAGTs, and CC. The respective 18, 15, and 7 cases for MA, disAGTs, and CC are provided as well. In all, a sorted list of 40 cases for the differences between the model (mcAM) and modeled code (cSC) is provided.

Of particular interest is the fact that in the case of MA and disAGTs, they do have their own cases, yet some of them are opposite to each other. This is because the decisions made in the mcAM design are not acceptable in the cSC implementation, and thereby different decisions are taken. Thus, one concern arises as to how to create a model at an appropriate level of manual abstraction that would be easily accepted by the developer in the code implementation. The cases of MA and disAGTs can provide this guideline. Moreover, in regard

to the consistency checks of the differences and similarities between the model and modeled code, this is a lack of developing mapping rules between the model and modeled code. Existing checks are purely structural without interpreting the semantics conveyed by the model elements. Thus, our concluded 40 cases of MA, disAGTs, and CC can provide input for developing such mappings rules in the future.

6.2 Future Work

Covering additional OOP languages and conducting the quantitative analysis could be a future extension of this thesis work, which is detailed in this section.

6.2.1 Covering Additional OOP Projects

Merely Java projects are selected as the thesis study subjects. However, considering the generalization of our results, future work can consider covering additional OOP projects, such as C++, and the generalization will be better guaranteed. While different OOP languages have their own specifications/structures, the essence of MA characteristics is expected to be more or less adaptable to other OOP languages. This is because all OOP projects follow the concepts of the OO paradigm. Thus, the characteristics of MA, disAGTs, and CC that we investigated will provide valuable insights into the differences between the model and the modeled code in terms of other OOP languages.

6.2.2 Quantitative Analysis

This thesis provides a qualitative analysis of the differences between mcAM and the cSC of that mcAM by conducting five case studies with the aim of discovering the MA characteristics. However, the extent to which the MA created in the mcAM is more acceptable in the implementation of the cSC will be of interest. Thus, a quantitative analysis is needed to investigate which MA cases concluded are more acceptable than others during the implementation of the cSC.

Bibliography

- [1] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, “The quest for open source projects that use uml: Mining github,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS ’16, 2016, pp. 173–183.
- [2] M. Stoica, M. Mircea, and B. Ghilic-Micu, “Software development: Agile vs. traditional.,” *Informatica Economica*, vol. 17, no. 4, 2013.
- [3] M. H. Osman, T. Ho-Quang, and M. R. V. Chaudron, “An automated approach for classifying reverse-engineered and forward-engineered uml class diagrams,” in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 396–399.
- [4] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of mde in industry,” in *Proceedings of the 33rd international conference on software engineering*, pp. 471–480, May 2011.
- [5] Y.-G. Gueheneuc, “A systematic study of uml class diagram constituents for their abstract and precise recovery,” in *11th Asia-Pacific Software Engineering Conference*, 2004, pp. 265–274.
- [6] D. Berardi, D. Calvanese, and G. De Giacomo, “Reasoning on uml class diagrams,” *Artificial Intelligence*, vol. 168, no. 1, pp. 70–118, 2005, ISSN: 0004-3702. DOI: <https://doi.org/10.1016/j.artint.2005.05.003>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0004370205000792>.
- [7] Y.-G. Guéhéneuc, “A reverse engineering tool for precise class diagrams,” in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 2004, pp. 28–41.
- [8] *SDLC - Overview*. [Online]. Available: https://www.tutorialspoint.com/sdlc/sdlc_overview.htm (visited on 09/25/2022).
- [9] D. Galin, *Software Quality Assurance: Concepts and Practice*. Wiley-IEEE Computer Society PR, Feb. 2017.
- [10] M. H. Osman, “Interactive scalable condensation of reverse engineered uml class diagrams for software comprehension,” Leiden University, Doctoral Thesis, Mar. 2015. [Online]. Available: <https://scholarlypublications.universiteitleiden.nl/handle/1887/32210>.
- [11] T. Ho-Quang, R. Hebig, G. Robles, M. R. V. Chaudron, and M. A. Fernandez, “Practices and perceptions of uml use in open source projects,” in *2017 IEEE/ACM 39th International Conference on Software Engi-*

- neering: *Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 203–212.
- [12] M. H. Osman and M. R. V. Chaudron, “Uml usage in open source software development: A field study,” in *3rd International Workshop on Experiences and Empirical Studies in Software Modeling co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2013, pp. 23–32.
- [13] M. H. Osman, M. R. V. Chaudron, and P. Van Der Putten, “Interactive scalable abstraction of reverse engineered uml class diagrams,” in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1, 2014, pp. 159–166.
- [14] J. D. Blischak, E. R. Davenport, and G. Wilson, “A quick introduction to version control with git and github,” *PLOS Computational Biology*, vol. 12, no. 1, e1004668, 2016. [Online]. Available: <https://app.dimensions.ai/details/publication/pub.1012419886>.
- [15] “Github - zootypers.” (Apr. 29, 2013), [Online]. Available: <https://github.com/orgs/ZooTypers/repositories> (visited on 12/12/2022).
- [16] “Github - lekogabi/raisemeup.” (Oct. 17, 2015), [Online]. Available: <https://github.com/lekogabi/RaiseMeUp> (visited on 12/12/2022).
- [17] H. Mu and S. Jiang, “Design patterns in software development,” in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, pp. 322–325. DOI: 10.1109/ICSESS.2011.5982228.
- [18] “Github - tekosds/neurophchanges.” (Mar. 21, 2013), [Online]. Available: <https://github.com/tekosds/NeurophChanges> (visited on 12/12/2022).
- [19] “Github - antoniorochaoliveira/eapli_pl_2nb.” (Mar. 30, 2013), [Online]. Available: https://github.com/AntonioRochaOliveira/EAPLI_PL_2NB (visited on 12/12/2022).
- [20] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, “Chapter 8. classes,” Feb. 28, 2013. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html> (visited on 12/12/2022).
- [21] Object Management Group, Inc., “About the unified modeling language specification version 2.4.1,” Jul. 2011. [Online]. Available: <https://www.omg.org/spec/UML/2.4.1> (visited on 12/27/2022).
- [22] GeeksforGeeks, “Java collection tutorial,” May 9, 2022. [Online]. Available: <https://www.geeksforgeeks.org/java-collection-tutorial/> (visited on 12/29/2022).
- [23] D. Poo, D. Kiong, and S. Ashok, *Object-Oriented Programming and Java*. New York, United States: Springer Publishing, 2008.
- [24] B. M. Ahmed, A. A. Boudhir, and A. Younes, *Innovations in Smart Cities Applications Edition 2: The Proceedings of the Third International Conference on Smart City Applications (Lecture Notes in Intelligent Transportation and Infrastructure)*, 1st ed. 2019. Springer, Feb. 7, 2019.

-
- [25] F. B. Abreu and R. Carapuça, “Object-oriented software engineering: Measuring and controlling the development process,” in *Proceedings of the 4th international conference on software quality*, vol. 186, 1994.
 - [26] A. Snyder, “Encapsulation and inheritance in object-oriented programming languages,” *SIGPLAN Not.*, vol. 21, no. 11, pp. 38–45, Jun. 1986.
 - [27] J. Micallef, “Encapsulation, reusability and extendibility in object-oriented programming languages,” *Journal of Object-Oriented Programming*, vol. 1, no. 1, pp. 12–36, 1988.
 - [28] Sparx Systems Ltd. and SparxSystems Software GmbH., *Model driven uml tool*. [Online]. Available: <https://www.sparxsystems.eu/> (visited on 01/11/2023).
 - [29] JetBrains s.r.o. “IntelliJ idea – the leading java and kotlin ide.” (Nov. 10, 2018), [Online]. Available: <https://www.jetbrains.com/idea/> (visited on 01/11/2023).
 - [30] “Uml class diagram tutorial.” (Aug. 27, 2022), [Online]. Available: <https://www.lucidchart.com/pages/uml-class-diagram> (visited on 04/07/2022).
 - [31] Object Management Group, Inc., “About the unified modeling language specification version 1.5,” Mar. 2003. [Online]. Available: <https://www.omg.org/spec/UML/1.5> (visited on 12/27/2022).
 - [32] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, “Design-code traceability recovery: Selecting the basic linkage properties,” *Science of Computer Programming*, vol. 40, no. 2, pp. 213–234, 2001, Special Issue on Program Comprehension.
 - [33] L. Gupta, “Java naming conventions,” Jan. 30, 2022. [Online]. Available: <https://howtodoinjava.com/java/basics/java-naming-conventions/> (visited on 12/21/2022).
 - [34] Oracle, Co., “Naming conversions,” Apr. 20, 1999. [Online]. Available: <https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html> (visited on 12/21/2022).
 - [35] M. H. Osman, A. van Zadelhoff, D. R. Stikkolorum, and M. R. V. Chaudron, “Uml class diagram simplification: What is in the developer’s mind?,” ser. EESSMod ’12, 2012.
 - [36] M. H. Osman, A. Zadelhoff, and M. R. V. Chaudron, “Uml class diagram simplification: A survey for improving reverse engineered class diagram comprehension,” pp. 291–296, Jan. 2013.
 - [37] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, “Reverse engineering: A roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 47–60.
 - [38] E. Chikofsky and J. Cross, “Reverse engineering and design recovery: A taxonomy,” *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990. DOI: 10.1109/52.43044.
 - [39] M. Booshehri and P. Luksch, “Condensation of reverse engineered uml diagrams by using the semantic web technologies,” in *Proceedings of the International Conference on Information and Knowledge Engineering (IKE)*, 2015, p. 95.

- [40] Y.-G. Guéhéneuc and H. Albin-Amiot, “Recovering binary class relationships: Putting icing on the uml cake,” *SIGPLAN Not.*, vol. 39, no. 10, pp. 301–314, Oct. 2004.
- [41] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron, “Condensing class diagrams by analyzing design and network metrics using optimistic classification,” in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014, 2014, pp. 110–121.
- [42] M. H. Osman, M. R. V. Chaudron, and P. Van Der Putten, “An analysis of machine learning algorithms for condensing reverse engineered class diagrams,” in *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 140–149.
- [43] D. J. van Opzeeland, C. F. Lange, and M. R. V. Chaudron, “Quantitative techniques for the assessment of correspondence between uml designs and implementations,” in *9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2005.
- [44] R. Shatnawi and A. Alzu’bi, “A verification of the correspondence between design and implementation quality attributes using a hierarchical quality model,” *IAENG International Journal of Computer Science*, vol. 38, no. 3, pp. 225–233, 2011.
- [45] S. Baltes and S. Diehl, “Sketches and diagrams in practice,” ser. FSE 2014, Association for Computing Machinery, 2014, pp. 530–541. DOI: 10.1145/2635868.2635891.
- [46] K. Yatani, E. Chung, C. Jensen, and K. N. Truong, “Understanding how and why open source contributors use diagrams in the development of ubuntu,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’09, 2009, pp. 995–1004. DOI: 10.1145/1518701.1518853.
- [47] E. Chung, C. Jensen, K. Yatani, V. Kuechler, and K. N. Truong, “Sketching and drawing in the design of open source software,” in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 195–202. DOI: 10.1109/VLHCC.2010.34.
- [48] B. Karasneh and M. R. V. Chaudron, “Online img2uml repository: An online repository for uml models,” in *EESSMod@MoDELS*, 2013.
- [49] K.-J. Stol and B. Fitzgerald, “The abc of software engineering research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 1–51, Sep. 2018.
- [50] A. Egyed, “Automated abstraction of class diagrams,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 4, pp. 449–491, Oct. 2002.
- [51] “Github - fmacicasan/freedaysintern.” (Jan. 29, 2013), [Online]. Available: <https://github.com/fmacicasan/FreeDaysIntern> (visited on 12/12/2022).
- [52] Oracle, Co., “Chapter 4. types, values, and variables,” Oct. 30, 2022. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html> (visited on 12/21/2022).
- [53] D. C. Schmidt *et al.*, “Model-driven engineering,” *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

- [54] R. B. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *Proceedings of the Future of Software Engineering*, IEEE, 2007, pp. 37–54.
- [55] Object Management Group, Inc., “Object constraint language (ocl) specification, version 2.0,” 2003. [Online]. Available: <https://www.omg.org/spec/OCL/2.0/> (visited on 02/23/2023).
- [56] M. A. Babar, T. Dingsøyr, and P. Lago, Eds., *Software Architecture: 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011, Proceedings*, vol. 6903, Lecture Notes in Computer Science, Springer, 2009. DOI: 10.1007/978-3-642-23798-0.
- [57] G. Murphy, D. Notkin, and K. Sullivan, “Software reflexion models: Bridging the gap between source and high-level models,” *ACM Software Engineering Notes*, vol. 20, May 1996.
- [58] L. Gren, “Standards of validity and the validity of standards in behavioral software engineering research: The perspective of psychological test theory,” ser. ESEM ’18, Association for Computing Machinery, 2018.
- [59] Java Techies Pvt. Ltd. “Data types.” (Oct. 28, 2022), [Online]. Available: <http://jstechies.in/core-java/data-type/java-datatypesd41d.php?> (visited on 04/04/2022).

A

Corresponding mcAMs in the Five Projects Studied

The corresponding five mcAMs of the five Java projects studied are presented in the following, which is used for illustrating the examples of the cases presented in Chapter 4.

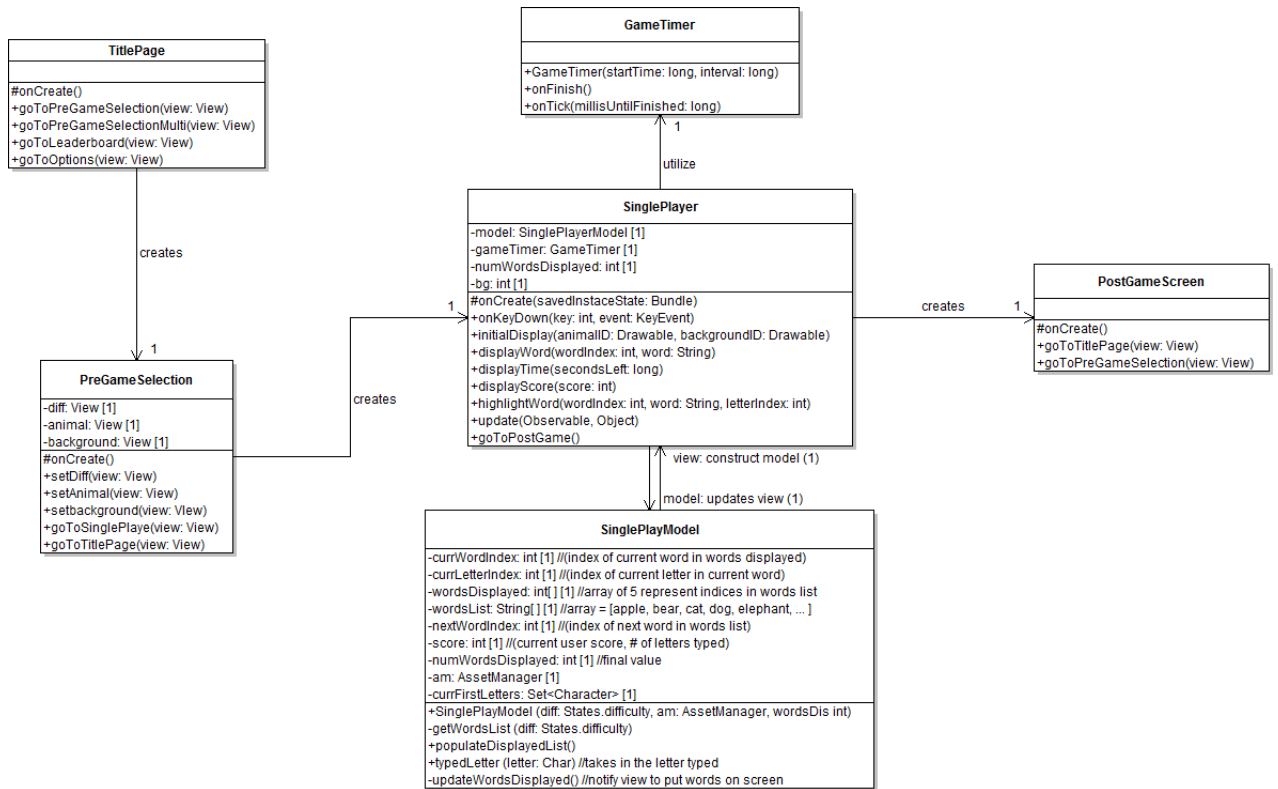


Figure A.1: Selected mcAM included in project 1.

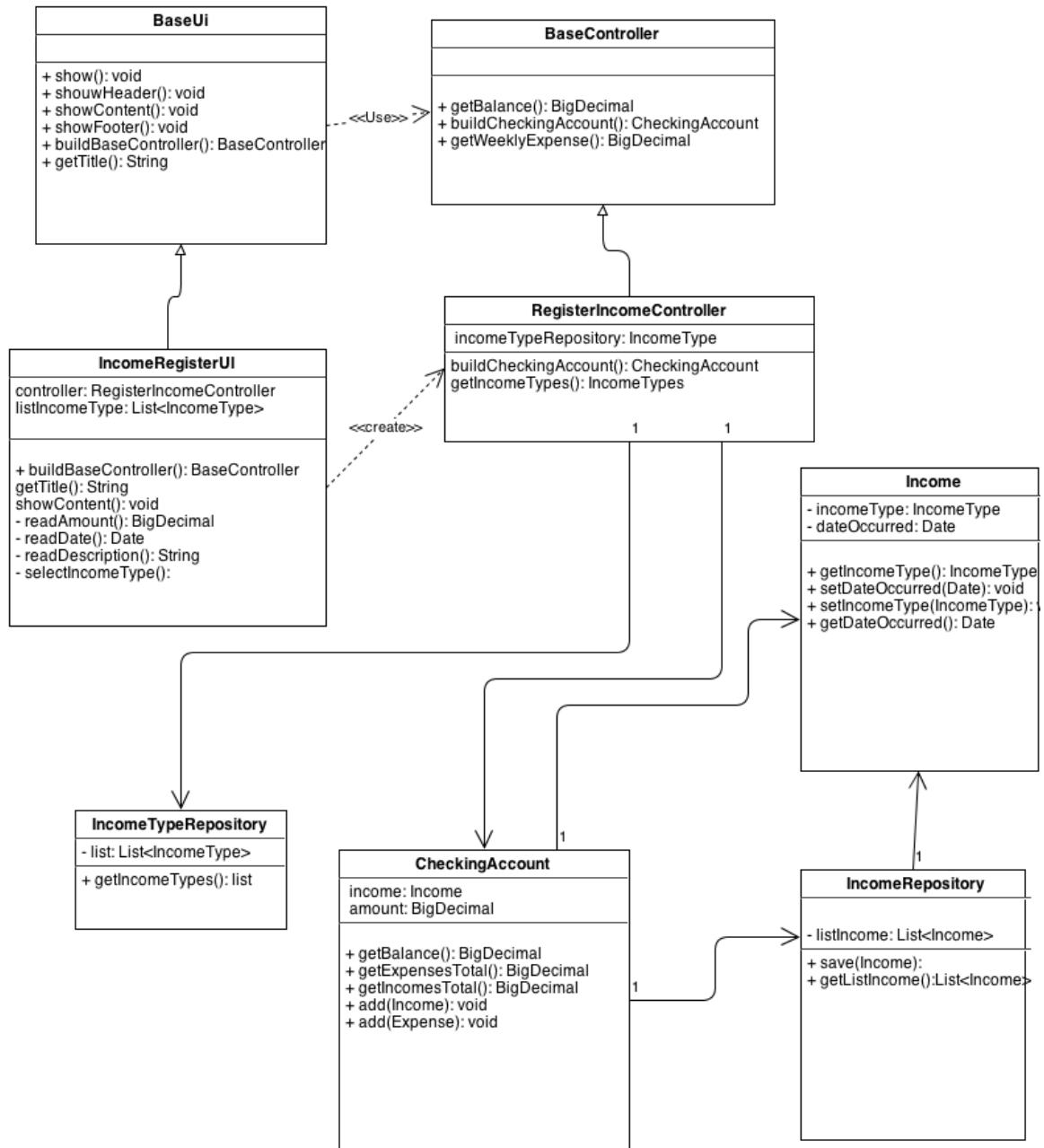


Figure A.3: Selected mcAM included in project 3.

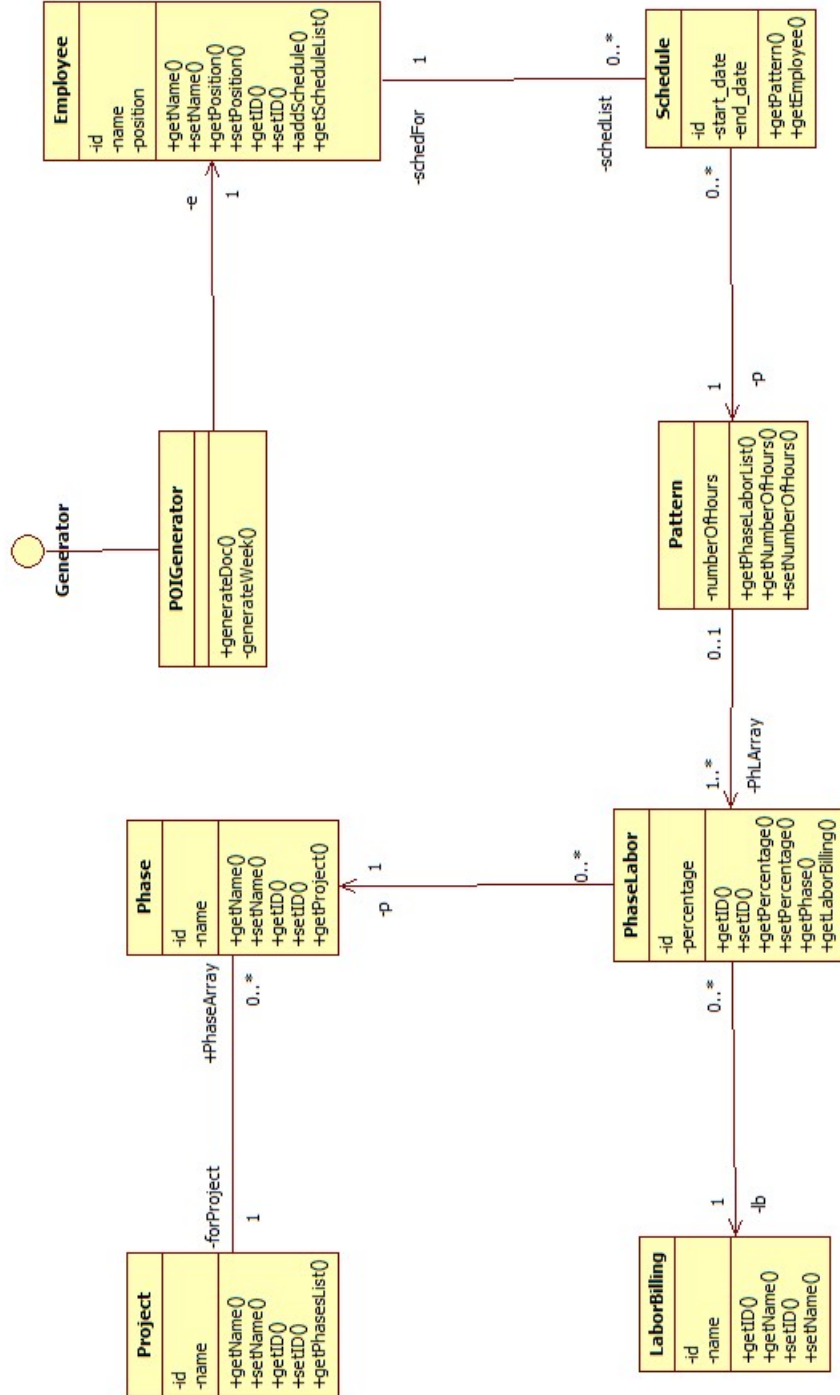


Figure A.4: Selected mcAM included in project 4.

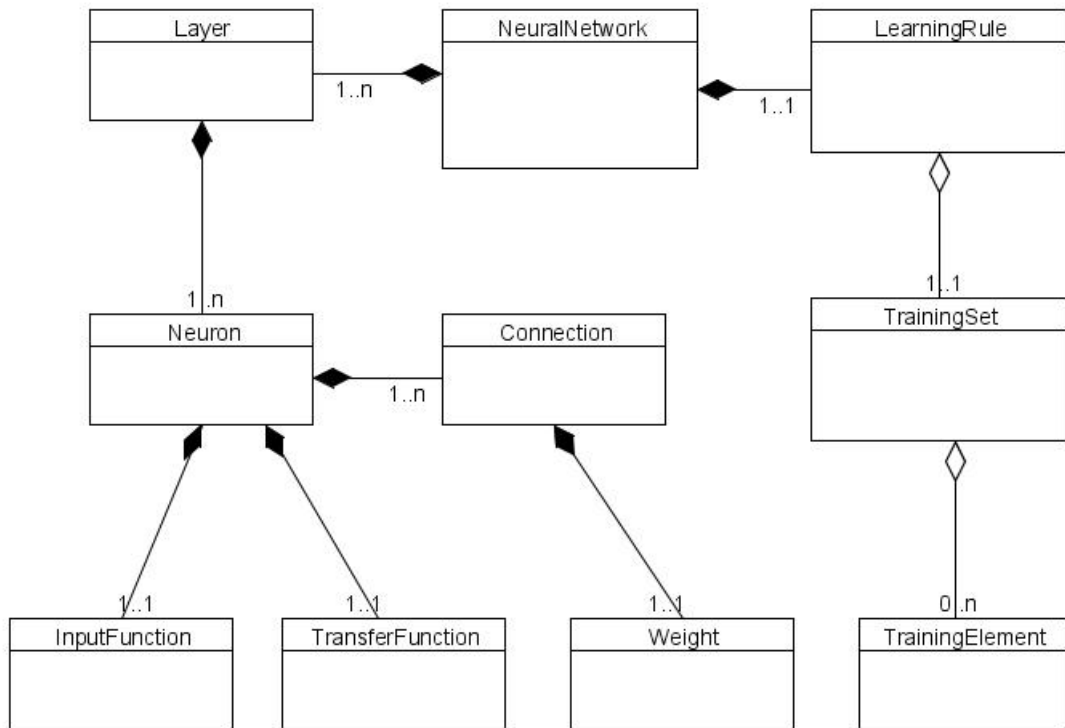


Figure A.5: Selected mcAM included in project 5.