Evolvability Evaluation of Conceptual-Level Inheritance Implementation Patterns

Marek Suchánek and Robert Pergl

Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
Email: marek.suchanek,robert.pergl@fit.cvut.cz

Abstract—**Inheritance is a well-known construct in conceptual modelling, as well as in the object-oriented programming, where it is often used to enable reusability and to modularize complex applications. While it helps in conceptual modelling and understanding of complex domains, it usually results in evolvability issues in software implementations. This paper discusses problems caused by single and multiple inheritance with respect to increasing accidental complexity of a model and evaluates various patterns that can be used to transform conceptual-level inheritance into implementation with respect to code evolvability. The points are illustrated on the transformation of an example ontological conceptual model in OntoUML into various software implementation models.**

Keywords*–*Inheritance; Generalization; UML; OntoUML; Evolvability; Normalized Systems.

I. Introduction

Software engineering uses conceptual notions of inheritance, generalization, or subtyping for decades in an analysis of a problem domain as well as in design and implementation of applications [1]. It captures a very essential and natural property of something being a special case of something more generic. Inheritance is one of the abstractions that we can use to describe and model real-world concepts based on our cognition principles. On the other hand, in software implementations, inheritance is often abused and used inappropriately causing ripple effects and thus negatively affecting software evolvability [2]. This worsens obviously when it comes to multiple inheritance, which is, however, again natural in the real world but complicated and sometimes even unsupported in software implementations.

Evolvability problems caused by inheritance in Object-Oriented Programming (OOP) are well-known and it is often suggested to follow composite reuse principle clause "composition over inheritance" [3]. Majority of OOP design patterns use composition together with generic interfaces instead of class generalizations. Our goal here is to explore, describe, and evaluate patterns for translating conceptual-level inheritance into combinatorial effect-free (CE-free) implementation and discuss its suitable use cases. The result should help to reduce the *accidental complexity* of models that is introduced due to technical reasons – as opposed to domain-given and irreducible *essential complexity*.

In Section III, we summarize the related terminology and the current state of the art in solving the problem of inheritance in the software engineering domain. Then, in Section IV we use this knowledge to discuss and evaluate the evolvability of various solutions in the form of patterns. Evaluation is summarized and all patterns are compared in Subsection IV-E. Finally, we propose possible next steps in this work and describe related research questions in Section V.

II. Methodology

For the conceptual ontological models, we use the OntoUML language and for the object-oriented models, we use UML. OntoUML is an ontology-driven conceptual modelling language based on terms from Unified Foundational Ontology (UFO) [4] focused on producing expressive, highly ontologically-relevant models. The Unified Modeling Language (UML) by OMG [5] is probably not necessary to introduce. Its difference to OntoUML in our context is mainly its historical focus on depicting object-oriented programmes rather than ontological committment.

The core of this paper are proposals of model transformation patterns. As such, we suppose generation of code from the models and then customizations in the style of model-driven development [6] or Normalized Systems Expanders [7]. In descriptions of models, we strictly distinguish between an *entity* on the conceptual level and a *class* in UML and a related OOP implementation since there is a significant semantic difference between these two – a (UML) class is an OOP representation of (possibly more) OntoUML entities, as we show later. To avoid misunderstanding we also respectively use *instance* of entity or class instead of word *object*.

We discuss the evolvability issues with the respect to a "user", being an end user of a software application implementing CRUD operations related to a discussed model and also with the respect to "programmer" being a programmer developing customizations of the generated code. These customizations are considered to be any piece of code that enhances or otherwise changes the functionality of the generated code.

III. Related Work and Terminology

In this section, we very briefly review necessary terminology for understanding various notions and use cases of inheritance on conceptual-level and its counterparts in implementation. Then, we also shortly describe existing and related solutions that affect our evaluation.

A. Generalization Typology

The notion of inheritance may mean semantically different things and such differences are unfortunately often ignored; a thorough discussion of various types of inheritance in software engineering is provided by Taivalsaari in [1]. When we talk about conceptual-level inheritance, we speak about *generalization* and *specialization*, or *is-a* relationship. It reflects our cognitive resolution of an entity being related to a different entity in a way that all instances of the first are also instances of the second, but in some specific aspects differ.

For inheritance in implementation, the terminology is a bit more complicated. The term *subclassing* describes using inheritance to reuse code, which is also called "accidental" use of inheritance. This sort of inheritance causes misunderstanding because it is used mainly for maintaining Do-not-Repeat-Yourself (DRY) code even though it creates combinatorial effects and there is no semantic foundation of such relation. The designation "essential" use of inheritance is used for *subtyping* that ensures substitutability in terms of the Liskov substitution principle [8]. [1] states that *subtyping* ensures *specialization*, but of course it can be again used inappropriately without alignment to the real world.

B. Is-a Hierarchies

The simplest way of modelling inheritance is the so-called *is-a hierarchy*, where two concepts are connected with is-a relationship as a subclass and a superclass of subsumption, e.g., *Student is a Person*. This started to appear widely in Extended Entity-Relationship (EER) models that can be considered as the predecessor of today's widely used object-oriented modelling languages, such as UML or Object-role modeling (ORM) [9]. Although such representation of inheritance is simplified, it is well-defined and allows straightforward mathematical reasoning [10].

C. OntoUML Conceptual-Level Inheritance

Generalization and specialization are essential in OntoUML; we use it to demonstrate conceptual-level inheritance. An instance in OntoUML can be an instance of multiple entities, however it must obey the single identity principle (given by `<<kind>>` or `<<subkind>>`), as depicted in Figure 1. This notion of multi-class instances is natural at the ontological level, however, as we discuss further, it is not typical in OOP and as such in standard UML. To bridge the ontological level with implementation, a research has been performed of transformation of OntoUML into UML and relational model [11]–[15], summarized in [16]. We evaluate here some of the patterns described in this work.

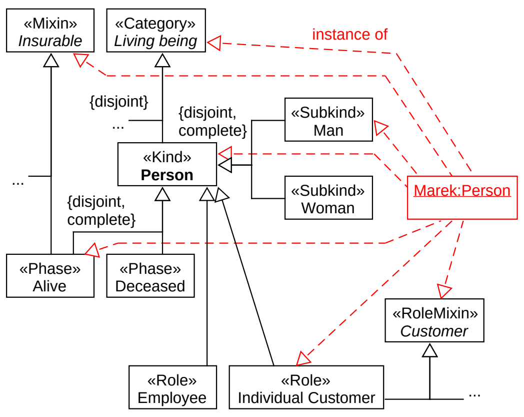

Figure 1. An excerpt of an OntoUML diagram showing inheritance and instantiation in OntoUML

D. Generalization Sets

Unified Modelling Language (UML) [5] defines a *generalization set* as an element whose instances define collections of subsets of generalization relationships. The purpose of such elements is to tell more about multiple generalizations via meta attributes. For example, a generalization set can be used to express that an instance of entity *Person* is either an instance of entity *Alive Person* or *Deceased Person* with meta attributes *disjoint* and *complete*.

E. Object-Oriented Inheritance

In OOP, inheritance is very often used as an enabler of well-known DRY and the mentioned Liskov substitution principle, one from SOLID principles recommended by Robert C. Martin [17]. A subclass can easily extend its superclass(es) by adding or changing attributes and methods. Deletion or cancellation of methods or attributes is often possible only by throwing an exception if called on a subclass. We take into further considerations only OOP where an object is an instance of exactly one class since the most widely used languages (Java, C#, C++, Python, etc.) do not support making instances of several classes [8]. Other languages that allow duck-typing and prototyping deal with some inheritance problems but creating other, usually leading to runtime errors caused by weak type system [18].

F. Inheritance in Normalized Systems

The Normalized System (NS) theory [2] deals with evolvability of systems in general and declares fine-grade modularization based on four main principles: Separation of Concerns, Separation of States, Data Version Transparency, and Action Version Transparency. Using those principles leads to a system with eliminated or at least controlled combinatorial effects (CE) – effectively being a measure of evolvability over time. Software systems are the core domain of NS theory application in practice [7].

In the NS theory, inheritance is criticized for causing CEs in software systems [2]. It is important to stress that CEs are caused by inheritance in OOP, not inheritance at the conceptual-level, as it may be transformed into a CE-free implementation, as will be discussed further. For the transformation of models in our work, the Separation of Concerns principle is crucial. Nevertheless, the remaining three principles are relevant for the implementation even though they are not applied directly in the transformation.

IV. Evaluation of Inheritance Patterns

The main part of this work is the description, evaluation, and summarization of various ways of implementing a conceptual-level inheritance in various use cases. For a demonstration of different patterns, we use Figure 1 as a conceptual model that should be implemented. We first introduce a simple approach with traditional inheritance implementation and discuss its problems. Then, we explore three patterns from the least to the most complex, their possible combinations and summarize their comparison. Those patterns are based on [9] [16], and can be found in many other software engineering resources. Our focus is not just directed on a generation of CE-free implementation of conceptual-level inheritance, but also on its impact on usability for both the user and the programmer.

We observe and discuss mainly the following measures that we quantify if possible to support our evaluation of the patterns:

- Total number of classes needed for implementation, composed of conceptual-level entity and additional classes.
- How many steps it takes to navigate to superclass or subclass.
- Change boundary, i.e., how many classes need to be changed when one changes in the conceptual model.

A. *Traditional OOP Inheritance*

A straightforward way how to implement conceptual-level inheritance is using *subtyping*. First, inheritance from conceptual model remains as is; the only change is that OntoUML's stereotypes are stripped off and some of the classes are made abstract to forbid their instantiation, as can be seen in Figure 2. Then, for each possible and instantiatable combination of conceptual entities, an empty concrete class is generated if needed using multiple inheritance depicted in gray. A generated concrete class is a subclass of appropriate abstract classes that implement entities. If there is a case that single inheritance is used (a class matches an entity $1 : 1$), such generated class can be removed and the superclass turned into a concrete one; in other cases, the accidental complexity grows and we must generate 1 class mixing $n > 1$ entities. In our model, instead of generating trivial classes with single inheritance *Man* and *Woman*, we just turn them from abstract to concrete resulting into 6 generated classes instead of 8.

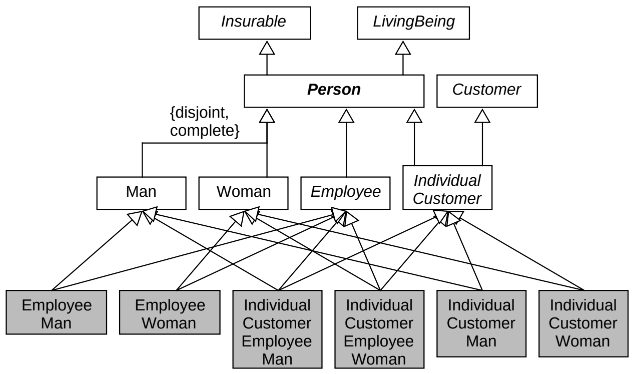

Figure 2. Traditional (multiple) inheritance pattern

Problem with this approach is that a language that supports multiple inheritance is needed, but more importantly, the introduced combinatorial effect is huge. A class instance implements a single or multiple conceptual entity instances. Therefore, there will be $\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{n} = 2^n$ generated classes, where $n + 1$ is the number of entities that can be instantiated (if there is just one entity, $n = 0$). For example, if we want to incorporate subclasses *Alive* and *Deceased* as in Figure 1, we need to completely change everything. There would be 16 generated classes in total and none of them would be the same as in the previous situation with 8 classes, which means a potential breaking change.

Another problem is caused by the rigidity of this design. When you want to turn an instance of class `AliveEmployeeWoman` into instance of `DeceasedEmployeeWoman`, a transformation procedure must be implemented – making system more complex internally, as well as for the user, as new forms are required. Last, programming customizations would be very difficult since, as shown, adding or removing classes at the conceptual level leads into significant changes in code that can make customizations invalid. Imagine that there is a customization for `CustomerWoman`, but then there is no such class only two `AliveCustomerWoman` and `DeceasedCustomerWoman` – a decision how to apply the customization on those would need to be made. As result, the only advantage is direct inheritance without any necessary navigation steps.

B. *The Union Pattern*

This new pattern we introduce here is very close to *Single Table Inheritance* (in some frameworks also referred as *Table-Per-Hierarchy*) that is used widely in relational databases. Basically, it is a class that unions all properties from the class and its subclasses – thus we call it *union pattern*. For each core class (identity provider entity in OntoUML) and its inheritance tree including all abstract superclasses and all subclasses, a union class is made as in Figure 3. Every subclass decision is covered by so-called *discriminator* that can be generated as an attribute and it tells which subclass(es) subset of every generalization sets is picked (e.g., `dManOrWoman`) or which single subtype is used (e.g., `dEmployee`).

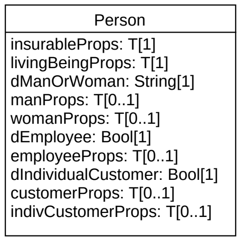

Figure 3. Union pattern

A significant disadvantage can be seen in the massive complexity of a single class and violating the Separation of Concerns principle. Adding a new subclass results in a definition of a new discriminator value and new fields in an existing class. It acts as a single complex data element with optional properties based on some other attributes. With this pattern, checks of actual subtypes must be implemented manually if needed, for instance, if there is a need to create a relationship only with *Employee*. Even more problems emerge where the hierarchy of subclasses is deeper. However, a user can be fully shielded from this implementation by having a single form with picking what is needed. For a programmer, this may result in tedious work (meaning time-consuming and error-prone) to handle all cases of complex class with multiple discriminators and many optional properties.

From the perspective of observed measures, there are precisely 0 of additional classes and one whole hierarchy from conceptual model is merged into one class. For navigation to subclass a single step of discriminator comparison has to be made (when relying on the integrity checks). Although this pattern is simple and seems to solve evolvability issues, it is practically a conceptual anti-pattern.

C. *Composition Pattern*

We call this pattern after the already-mentioned advice *composition over inheritance*. It is similar to the previous Union Pattern, but instead of merging all properties into one class, a composition is used. Thanks to that, no *discriminator* is needed and also no *subclassing* is needed as shown in Figure 4. All generalization relations are replaced by bidirectional *is-a* associations that are always mandatory for a subclass, but can be optional for superclass if the used generalization set is not *complete*. For *disjoint* sets or more complex constraints, additional checks, such as *xor*, must be implemented and a mere multiplicity setting is not sufficient.

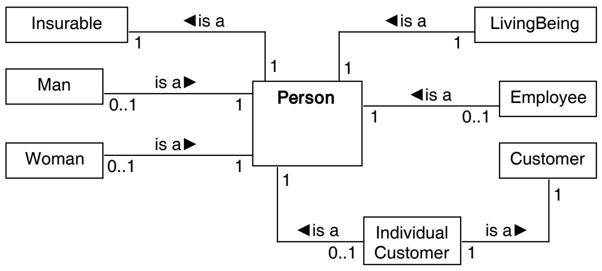

Figure 4. Composition pattern

An important change against the original conceptual model is that every abstract class is turned into concrete in order to be instantiated and such instance must be related (transitively) to exactly one instance of a core class. Transformation of generalizations into *is-a* associations can be also done fully automatically including integrity checks (for instance, that *Person* is *Woman* or *Man*, but never both). It is possible again to fully shield the implementation from the user as it works from this point of view similar to union pattern, just with special associations. Instead of additional comparison of discriminator value for navigation to subclass, the relation is checked and used if is realised. There is the same number of classes as in the conceptual model.

A programmer should have easier work than in the previously described patterns thanks to Separation of Concerns and possibility to use the power of a type system when passing subclass or superclass instances. Implementation and support mechanisms of *is-a* should allow a programmer to easily navigate to superclass instances, check if subclass instances are set and if so, navigate to it. When the inheritance hierarchy is changed within the model, some of the previous *is-a* links can be removed or edited causing incompatibility of customization. However, it is the same sort of ripple effect as for the union pattern, only the change is not encapsulated within a single (but huge) class but in a limited number of classes within a single hierarchy.

D. Generalization Set Pattern

The main weakness of the composition pattern is the need of handling generalization set constraints by implementing or generating constraints, and it can still make a problem when accessing instances of subclasses. We introduce this pattern to solve this problem and also to avoid having multiple *is-a* links attached directly to the instance together with all integrity checks – we want to separate this concern out. As depicted in Figure 5, for each class that has subclass(es), a special class for generalization sets (marked by «GS» stereotype) is generated. If there are other specialized generalization sets for an entity in the conceptual model (such as the one for *Man* and *Woman*), an extra generalization set class is generated and linked from the core one.

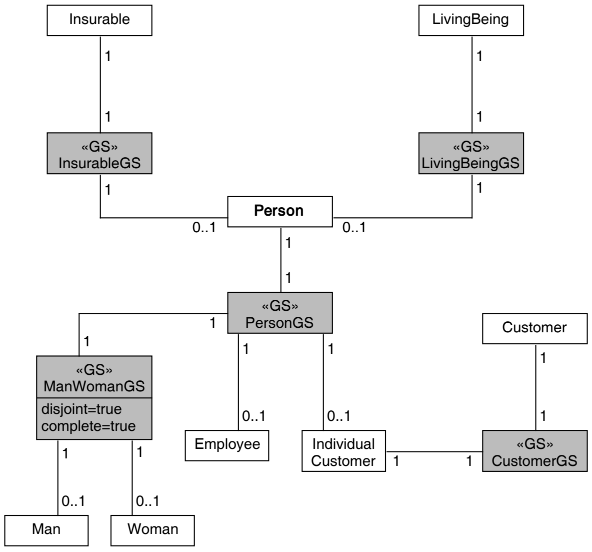

Figure 5. Generalization set pattern

Each «GS» class can be defined with class attributes such as *complete* and *disjoint* that can adjust the behaviour of generalization set including integrity checks or default form generation. Moreover, new attributes can be introduced to enhance functionality in the future without breaking the existing model. For a programmer, the work is simplified by cleaner data classes and «GS» encapsulating necessary utility for implementing the inheritance in exchange for extra navigation per a single generalization set ($A \leftrightarrow GS \leftrightarrow B$ instead of directly $A \leftrightarrow B$). An end user is again shielded and thanks to the generated «GS», the user interface can be generated easily and universally thanks to straightforward reuse and composition of simpler parts.

The significant negative issue with this pattern comes in form of overhead caused by a number of extra classes and extension of the generation tool by a new and non-trivial concept. Sometimes the «GS» class is unnecessary, for example, if there is no other *LivingBeing* than *Person* in the model. In such cases, extra «GS» class could be omitted but then a customization would use direct navigation. However, such a simplification leads to an increased impact of changes when a new subclass is added and generalization set is additionally created.

E. Comparison of Inheritance Patterns

We showed benefits of the discussed three patterns above in terms of evolvability compared to traditional OOP inheritance. We deliberately introduced the patterns in order so that the following removes the most significant problems of the previous. Unfortunately but naturally, it is never “for free” and each solution has its own problems, mainly in terms of additional complexity, which is, however, not an issue when we talk about automatically generated code. A summary of the discussion is provided in Table I.

We evaluated the generalization set pattern as generally the most suitable from those covered in this paper, however, there are use cases where some of the others may be better applicable. Below, we make such discussion from the conceptual point. In cases where time and/or space complexity needs to be optimised, conclusions about patterns suitability may differ.

TABLE I. COMPARISON OF INHERITANCE PATTERNS EVOLVABILITY

Evolvablity aspect	Traditional inheritance	Union pattern	Composition pattern	GS pattern
Number of additional classes	2^n	0	0	1 per GS
Multiple-inheritance support	yes (if the PL supports)	causing CE	with constraints	yes
Customizations evolvability	breaking changes	SoC violated	OK	OK
Accessing superclass	by the PL used	discriminator	association	associations
UI uniformity obstacles	migrations, repetition	repetition	constraints	no
Hierarchy change impact	whole hierarchy	class	association(s)	association(s), GS class(es)

The union pattern, when compared to the generalization set pattern, is suitable when the inheritance hierarchy is not deep but shallow with a single or very low number of generalization sets. In such case, multi-valued discriminator and optional fields will be used and handling a single instance will be very simple. On the other hand, if there are more levels of inheritance, it is too complex to use. The first option is that a superclass is merged into subclass(es) resulting in repetition (e.g., *Insurable* for *Person* but also *Building*). The second option is to merge at the top level, which causes a serious problem of a too complex class including non-related properties. The last serious problem is that this pattern cannot be used without repetition for multiple inheritance.

The composition pattern is very similar to the generalization set pattern and the only advantage is a reduced complexity by leaving out special «GS» classes. It allows multiple inheritance with simple reuse thanks to creating linked instances and it has no problem with deeper inheritance hierarchies compared to the union pattern. The key problem is with handling inheritance and generalization set constraints inside the superclass and subclass. That is not a problem for simple unconstrained generalizations, as already mentioned. The violation of Separation of Concerns principle is removed in the generalization set pattern, where it can be hidden using various OOP design patterns (such as Proxy) even for programmers.

F. Combinations of the Previous Patterns

For a transformation of a complex OntoUML model, it may also be a possibility to select different patterns for different parts of the model according to the discussion above. However, new concerns arise:

- The uniformity of user interface (UI) and experience user (UX) should be maintained.
- Using different patterns will necessarily result in different data accessing strategies (especially navigation of subclass/superclass), which complicates the job of the programmer and requires extra knowledge of the various patterns and their characteristics, also during the transformation phase.

It might seem a good idea to go for the Union pattern in case of a simple inheritance with multiple subclasses in one generalization set, which leads to one *discriminator*, or to remove «GS» class when it is not needed, but after considering the listed negative effects or additional needs, sticking with the most powerful and flexible generalization set pattern is generally recommended in spite of its higher internal complexity.

V. FUTURE RESEARCH AND RELATED PROBLEMS

The obvious future work is implementation of the transformations in an MDA-enabled CASE tool and design the necessary transformations and code generation. The proposed transformations should be now encoded and verified by real-world practice on larger code bases. There are also topics for more theoretical research on the topic.

A. Interfaces Evolvability

In this paper, we evaluated the inheritance of OOP in terms of subclassing. Some object-oriented programming languages, for instance, Java, comes with a construct called *interface* and a relation *implements*. Implementing an interface is in some aspects similar to extending a class but it is more constrained. Although this construct is not generally used at the conceptual-level and is tied to implementation, possible future research could evaluate those differences and its relation to evolvability. Similarly, there are additional constructs such as traits (e.g., in Pharo [19]) or mixins (e.g., in Ruby [20]) that call for similar analysis. Their potential to serve similarly as multiple inheritance in some respects make them interesting for this purpose. However, as they seem not the mainstream of today's OOP, we did not deal with them in the first case.

B. Automatic Transformation with Pattern Selection

In Section IV-F, we sketched an idea of patterns combination. If the selection is implemented in an automatic way and moreover, the API of the generated model is further abstracted so that it provides a unified approach for all the patterns, benefits of the patterns may be used without the pointed drawbacks. However, we are aware that this is a quite challenging task.

VI. CONCLUSION

In this paper, we briefly described and summarized terminology of inheritance in software engineering and the relation between inheritance on the conceptual level and its implementation in the source code. The possible patterns of implementing conceptual-level inheritance in OOP were introduced, evaluated, and compared in multiple aspects of evolvability. The implementation itself is shown to be non-trivial and there are more options suitable for different cases. Further verification of our contribution on real-world use cases is desirable as follow up in this research. During the work, we have encountered multiple related important questions and problems and suggested future research.

ACKNOWLEDGEMENTS

This research was supported by the grant of Czech Technical University in Prague No. SGS17/211/OHK3/3T/18. Initial steps in this research were done during the Normalized Systems Summer School 2018 organized by the University of Antwerp.

REFERENCES

[1] A. Taivalsaari, "On the Notion of Inheritance," ACM Computing Surveys, vol. 28, no. 3, September 1996, pp. 438–479.

[2] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Kermt (Belgium): Koppa, 2016.

[3] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, March 1995.

[4] G. Guizzardi, Ontological Foundations for Structural Conceptual Models. CTIT, Centre for Telematics and Information Technology, 2005.

[5] B. Selic et al., "OMG Unified Modeling Language (Version 2.5)," March 2015.

[6] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture – Practice and Promise. Addison-Wesley Professional, 2003.

[7] G. Oorts, P. Huysmans, P. D. Bruyn, H. Mannaert, J. Verelst, and A. Oost, "Building Evolvable Software Using Normalized Systems Theory: A Case Study," in 2014 47th Hawaii International Conference on System Sciences(HICSS), January 2014, pp. 4760–4769.

[8] R. W. Sebesta, Concepts of Programming Languages, 10th ed. Pearson, 2012.

[9] R. Elmasri and S. Navathe, Fundamentals of Database Systems. London: Pearson, 2016.

[10] A. Artale, D. Calvanese, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev, "Reasoning over Extended ER Models," in International Conference on Conceptual Modeling. Springer, 2007, pp. 277–292.

[11] Z. Rybola and R. Pergl, "Towards OntoUML for Software Engineering: Introduction to the Transformation of OntoUML into Relational Databases," in Enterprise and Organizational Modeling and Simulation, ser. LNBIP. Ljubljana, Slovenia: Springer, June 2016.

[12] ——, "Towards OntoUML for Software Engineering: Transformation of Rigid Sortal Types into Relational Databases," in Proceedings of {FedCSIS} 2016, ser. ACSIS, vol. 8. Gdańsk, Poland: IEEE, September 2016, pp. 1581–1591.

[13] ——, "Towards OntoUML for Software Engineering: Transformation of Anti-Rigid Sortal Types into Relational Databases," in Model and Data Engineering, ser. LNCS. Aguadulce, Almería, Spain: Springer, September 2016, pp. 1–15.

[14] ——, "Towards OntoUML for Software Engineering: Transformation of Kinds and Subkinds into Relational Databases," Computer Science and Information Systems, vol. 14, no. 3, 2017, pp. 913–937.

[15] ——, "Towards OntoUML for Software Engineering: Optimizing Kinds and Subkinds Transformed into Relational Databases," in Enterprise and Organizational Modeling and Simulation, ser. LNBIP. Tallinn, Estonia: Springer, Cham, November 2018, pp. 31–45.

[16] Z. Rybola, "Towards OntoUML for Software Engineering: Transformation of OntoUML into Relational Databases," Ph.D. thesis, Czech Technical University in Prague, Prague, Czech Republic, August 2017, [retrieved: Apr, 2019]. [Online]. Available: https://www.fit.cvut.cz/sites/default/files/PhDThesis-Rybola.pdf

[17] R. C. Martin, "Design Principles and Design Patterns," Object Mentor, vol. 1, no. 34, 2000, p. 597.

[18] B. C. Pierce and C. Benjamin, Types and Programming Languages. MIT press, 2002.

[19] A. Bergel, D. Cassou, S. Ducasse, J. Laval, and J. Bergel, Deep into Pharo. Square Bracket, 2013.

[20] D. Thomas et al., Programming Ruby: The Pragmatic Programmers' Guide. Raleigh, NC: Pragmatic Bookshelf, 2005.