# Evolvability Analysis of Multiple Inheritance and Method Resolution Order in Python

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 as a relation for expressing generalisations and specialisations or taxonomies is natural for conceptual modelling, but causes evolvability problems in software implementations. Each inheritance relation represents a tight coupling between a superclass and a subclass. Coupling in this case leads to a combinatorial effect or even combinatorial explosion in case of complex hierarchies. This paper analyses how multiple inheritance and method resolution order affect these problems in the Python programming language. The analysis is based on the design of inheritance implementation patterns from our previous work. Thanks to the flexibility of Python, it shows that inheritance can be implemented with minimisation of combinatorial effect using the patterns. Nevertheless, it is crucial to generate helper constructs related to the patterns from the model automatically for the sake of evolvability, including potential future in the patterns themselves.**

***Keywords–Multiple Inheritance; Python 3; Evolvability; Method Resolution Order; Composition Over Inheritance.***

## I. Introduction

Inheritance in software engineering is a widely used term and technique in both system analysis and software development. During analysis, where we want to capture a specific domain, inheritance serves for refining more generic entities into more specific ones, e.g., an employee as a specialisation of a person. It is natural to have multiple inheritance, e.g., a wooden chair is a seatable physical object and is also a flammable object. On the other hand, in Object-Oriented Programming (OOP), inheritance can be used or even misused for various purposes, including re-use of methods and attributes. In OOP, it causes so-called ripple effects violating evolvability of software [1]–[3].

The Python programming language is (according to [4]) the most popular multi-paradigm and general-purpose programming language. It is dynamically typed, but allows multiple inheritance and also type hints [5]. Sometimes, Python is also referred as "executable pseudocode" thanks to its easy-to-read syntax and versatility [6]. For the implementation of conceptual-level inheritance in OOP, there are several patterns suggested in [2]. Although the patterns are compared and evaluated, implementation examples and empirical proofs are missing.

In this paper, we want to design implementation of the conceptual-level inheritance patterns in Python using its specific constructs to allow easy use of the patterns instead of traditional OOP inheritance that causes ripple effects. The prototype implementation is design to serve for comparison and demonstration of complexity added in terms of additional constructs versus complexity caused by combinatorial effects. First, Section II acquaints the reader with terminology and the overall context. Then in Section III, we analyse how traditional inheritance work in Python and then describe the design and implementation of each pattern. Finally, Section IV evaluates the implementations, as well as the traditional Python inheritance and suggests possible future research.

## II. Related Work and Terminology

In this section, we briefly introduce the related research and terminology required for our approach. It refers to vital sources related to software evolvability, both conceptual-level and OOP inheritance, and Python programming languages.

### A. *Normalized Systems Theory*

Normalized Systems Theory [1] (NST) explains how to design systems that are evolvable using the fine-grained modular structure and elimination of combinatorial effects, i.e., size of change impact is proportional to the size of the system. The book [1] also describes how to build such software systems based on four elementary principles: Separation of Concerns, Data Version Transparency, Action Version Transparency, and Separation of States. Violation of any of these principles leads to combinatorial effects. A code generation techniques producing skeletons from the NS model and custom code fragments are applied to make the development of evolvable information systems possible and efficient.

The theory [1] states that the traditional OOP inheritance inherently causes combinatorial effects. Without multiple inheritance, it even leads to the so-called "combinatorial explosion", as you need a new class for each and every combination of all related classes to make an instance that inherits different things from multiple classes, e.g., a class `JuniorBelgianEmployeeInsuredPerson`. But even with multiple inheritance, the generalisation/specialisation relation is special and carries potential obstacles to evolvability. First, the coupling between subclasses and superclasses with the propagation of non-private attributes and methods is evident. Also, persisting the objects in traditional databases is challenging [1] [2] [7].

### B. *Conceptual-Level Inheritance*

Inheritance in terms of generalisation and specialisation relation is ontologically aligned with real-world modelling. It is tightly related to ontological refinements, where some concept is further specified in higher detail. It forms a taxonomy – a classification of things or concepts. For example, an employee is a special type of a person, or every bird is an animal. In conceptual modelling, inheritance is widely used to capture taxonomies and refine concepts under certain conditions. Although it is named differently in various languages, e.g., is-a hierarchy, generalisation, inheritance, all usually work with the ontological refinements. As shown in [8] different views on inheritance can be made with respect to implementation, where it can be (mis)used for reuse of classes without a relevant conceptual sense [2] [9].

### C. Object-Oriented Programming and Inheritance

When talking about inheritance in OOP, it is crucial to distinguish between class-based and prototype-based style. In prototype-based languages, objects inherit directly from other objects through a prototype property. Basically, it is based on cloning and refining objects using specially prepared objects called prototypes [10]. On the other hand, a more traditional and widespread class-based programming creates a new object through a class's constructor function that reserves memory for the object and eventually initialises its attributes. In both cases, inheritance is used for polymorphism by substituting superclass instance by subclass instance with eventually different behaviour [8] [11].

Both single and multiple inheritance can be used for reuse of source code. In [8], a clear explanation between essential and accidental (i.e., purely for reuse) use of inheritance is made. Moreover, [12] shows how multiple inheritance leverages reuse of code in OOP, including its consequences. According to [13], Python programs use widely (multiple) inheritance and it is comparable to use of inheritance in Java programs of the similar sample set.

### D. The Python Programming Language

Python is a high-level and general-purpose programming language that supports (among others) the object-oriented paradigm with multiple inheritance. It allows redefinition of almost all language constructs including operators, implicit conversions, class declarations, or descriptors for accessing and changing attributes of objects and classes. Both methods and constants for such redefinitions start and end with a double underscore and are commonly called "magic", e.g., magic method `__add__` for addition operator. The syntax is clean as it uses indentation for code blocks and limits the use of brackets. Python can be used for all kinds of application from simple utilities and microservices to complex web application and data analysis [5] [13].

Often, Python is used for prototyping, and then the production-ready system is built in different technologies such as Java EE or .NET for enterprise applications and C/C++ or Rust for space/time optimisation. Another essential aspect that makes Python a suitable language for prototyping is its dynamic type system that allows duck typing [14], however static typing is supported using annotations since version 3.5. A Python application can be then checked using type checkers or linters similarly to compilers, while preserving a flexibility of dynamic typing [5] [15].

"The diamond problem" related to multiple inheritance is solved using "Method Resolution Order" (MRO) that is based on the C3 superclass linearisation algorithm. Normally, a class in Python has method `mro` that lists the linearised superclasses. It can also be redefined using metaclasses, i.e., classes that have classes as its instances. By default, a class is a subclass of class `object` and an instance of metaclass `type`. Class `object` has no superclass and it is an instance of `type`. Class `type` is a subclass of `object` and is an instance of itself [5] [11].

## III. Python Inheritance Analysis

In this section, we analyse how conceptual-level inheritance implementation patterns proposed in [2] can be used in Python. We discuss the implementation options with respect to evolvability and ease of use, i.e., the impact of the pattern on the potential code base.

For demonstration, we use a conceptual model depicted in Figure 1 using OntoUML [9]. We use monospaced names of class and object names in the following text, e.g., `Person`. We strive to design the implementation of such a model where object `marek` is an instance of multiple classes with minimal development effort but also minimal combinatorial effects. Our model also contains the potential diamond problem, i.e., class `AlivePerson` inherits from `Locatable` via `Insurable` but also via `LivingBeing` and `Person`. Overriding is also included using derived attributes, as we avoid methods for the sake of clarity; however, it would work equivalently.

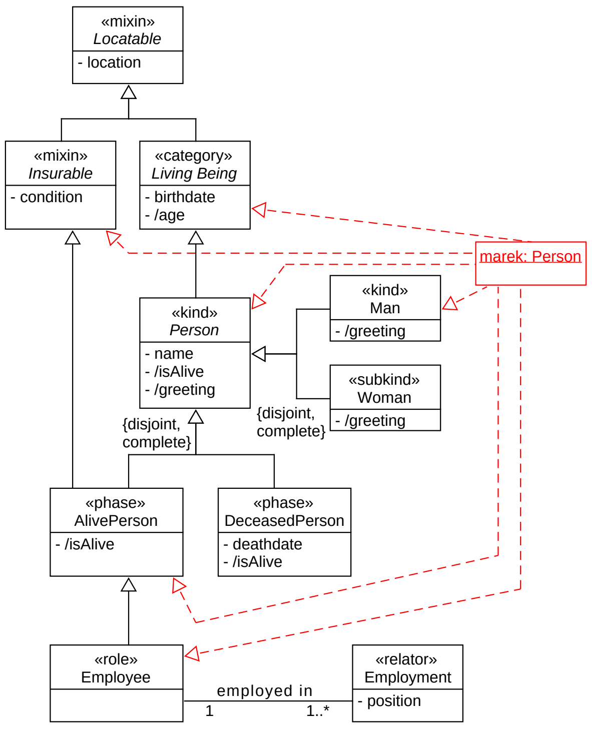

Figure 1. Diagram of OntoUML example model with instance

### A. Traditional OOP Inheritance

The first of the patterns uses a default implementation of inheritance in the underlying programming language. In case of Python, multiple inheritance with MRO allows creating subclasses for combinations given by the conceptual model. We immediately run into the combinatorial effect. First, we need to implement classes according to the model with inheritance and call the initializer of superclass(es) in the initializer (i.e. `__init__` method). In case of single inheritance, it can be easily resolved using the built-in `super` function, but in case of multiple inheritance, all superclasses must be named again as call of the function `super` returns only the first matching according to MRO as shown in Section III-A. Also notice that all arguments of the initializer must be propagated and repeated. A possible optimization would be to use variadic `*args` and `**kwargs`, but in exchange for readability and checks with respect to number (and type) of arguments passed. Another interesting fact in our example is that `EmployeeMan` does not need to define the initializer, as it inherits the one from `Employee` and `Man` inherits it from `Person`. If `Man` has its own attributes, then `EmployeeMan` would have the initializer similarly to `AlivePerson`.

```
class LivingBeing(Locatable):

    def __init__(self, birthdate, location):
        super().__init__(location)
        self.birthdate = birthdate

    @property
    def age(self):
        # computation of age
        return result

class Man(Person):

    @property
    def greeting(self):
        return f'Mr. {self.name}'

class AlivePerson(Person, Insurable):

    def __init__(self, name, birthdate, location,
    ↪   condition):
        Person.__init__(self, name, birthdate,
        ↪   location)
        Insurable.__init__(self, condition)

    @property
    def is_alive(self):
        return True

class EmployeeMan(Employee, Man):
    pass  # Employee __init__ inherited


marek = EmployeeMan("Marek", ...)
```

Figure 2. Part of the traditional inheritance implementation

After having the model classes implemented, extra classes must be generated as an object can be instance of only one class. For example, `marek` is instance of such class `EmployeeMan`. For our simple case, number of extra classes is six – `Man` and `Woman` combined with `AlivePerson`, `DeceasedPerson`, and `Employee`. Adding a single new subclass of `Person`, e.g., `DisabledPerson`, would result in doubling the number and therefore a combinatorial explosion. The second point where a combinatorial effect resides is the order of superclasses (*bases* or *base classes* in Python), which influences MRO. For instance, if `Person` and `Insurable` define the same method – in our case the one from `Person` – it would be resolved for execution according to order in list `EmployeeMan.mro()`. On the attribute level, each change propagates to all subclasses, i.e., it is again a combinatorial effect. This can be avoided using the mentioned `**kwargs` and their enforcing, as shown in Section III-A. Knowledge of superclasses for initialization can be then used to automatically call the initializer of all the superclasses. We implement this in helper function `init_bases`, where superclasses are iteratered a initialized in the reverse order to follow the MRO, i.e., the initializer of first listed superclass is used as the last one to eventually override effects of others.

With implementation shown in Section III-A, all classes with initializers can be easily generated automatically from the model with a single exception. The order of classes – i.e., if `AlivePerson` should be a subclass of `Person` and then `Insurable` or vice versa – is not captured in the model, but it is crucial for MRO. The order of superclasses has to be encoded in the model, or alternatively all permutations must be generated, which would result in a significantly higher number of classes that are not necessarily needed. Navigation is done naturally thanks to MRO and Python itself, for example, `marek.greeting` or `marek.location`.

```
def init_bases(obj, cls, **kwargs):
    for base in reversed(cls.__bases__):
        if base is not object:
            base.__init__(obj, **kwargs)

class Person(LivingBeing):

    def __init__(self, *, name, **kwargs):
        init_bases(self, Person, **kwargs)
        self.name = name

class AlivePerson(Person, Insurable):

    def __init__(self, **kwargs):
        init_bases(self, AlivePerson, **kwargs)

    ...

class EmployeeMan(Employee, Man):

    def __init__(self, **kwargs):
        init_bases(self, EmployeeMan, **kwargs)


marek = EmployeeMan(name="Marek", ...)
```

Figure 3. Implementation of initializers and extra classes with use of keyword arguments and helper function for model-driven development

## B. *The Union Pattern*

The Union pattern basically merges an inheritance hierarchy into a single class. In our case, the “core” class of hierarchy can be naturally selected as `Person`. All subclasses are uniquely merged into `Person` and `Person` merges also all superclasses as shown in Section III-B. For example, if there is another subclass of `Insurable`, it would not be merged into `Person`. On the other hand, for example, a new subclass of `Man` would be merged. This pattern is inspired closely by the “single-table inheritance” used in relational databases, but it immediately runs into problems once behaviour should be implemented.

According to the pattern, each decision on generalisation set of subclasses must be captured in the class that unions the hierarchy. In our case, we need three discriminators – for `Man` and `Woman`, for `AlivePerson` and `DeceasedPerson`, and for `Employee`. Value of each discriminator described what subclass(es) are “virtually” instantiated. All of these generalisation sets are disjoint and complete with the exception of the one with `Employee` that is not complete (i.e., not all alive persons must be employees). If there is a non-disjoint generalisation set, it would be solved using enumeration of all possibilities for the discriminator. For example, if `Man/Woman` is not disjoint nor complete, there would be four possible options (no, just man, just woman, both) instead of current two (just man or woman).

To allow polymorphism without branching and checking the discriminator value and taking a decision on behaviour with combinatorial effect, we use directly classes for delegation as values for discriminators, similarly to the well-known "State pattern". With this implementation, it incorporates separation of concerns and improves re-usability. It is crucial that all attributes, i.e., data, are encapsulated in the single object that is passed during the calls. Section III-B shows `Delegation` descriptor for secure delegation of behaviour to separate classes that even do not need to be instantiated; therefore, static methods are used, and an instance of the union class is passed.

```
class Man:

    @staticmethod
    def greeting(person):
        return f'Mr. {person.name}'

class Delegation:

    def __init__(self, discriminator, attr):
        self.discriminator = discriminator
        self.attr = attr

    def __get__(self, instance, owner):
        d = getattr(instance, self.discriminator)
        a = getattr(d, self.attr) if d else None
        return a(instance) if callable(a) else a

class Person:

    greeting = Delegation('_d_man_woman',
    ↪   'greeting')
    is_alive = Delegation('_d_alive_deceased',
    ↪   'is_alive')
    age = Delegation('_x_living_being', 'age')

    def __init__(self, name, birthdate, location,
    ↪   condition):
        self.location = location
        self.condition = condition
        self.birthdate = birthdate
        self.name = name
        # optional-subclass attributes
        self.employment = None
        self.deathdate = None
        # discriminators
        self._d_man_woman = None
        self._d_alive_deceased = None
        self._d_employee = None
        # superclasses with behaviour
        self._x_living_being = LivingBeing

    def d_set_man(self):
        self._d_man_woman = Man

    def d_set_employee(self, employment):
        self._d_employee = Employee
        self.employment = employment

    ...
```

Figure 4. Part of union pattern implementation

It is essential to point out that this solution may reduce the number of classes, but only of purely data classes without behaviour. Union classes can be then easily generated from a conceptual model. The detection of the "core" class is a matter of the model – if OntoUML is used, naturally all identity providers (e.g., with stereotype `Kind`) are suitable. In modelling languages that have no such explicit indication, a special flag has to be encoded in the model. Classes encapsulating behaviour can also be easily generated from the model and related to data class using the explained `Delegation` descriptor. There is one problem with this pattern implementation – it does not support `isinstance` checks. When avoiding inheritance, the only possible solution lies in special metaclass that would override `__instancecheck__`. This would also require to forbid instantiation of behaviour classes, so it is unambiguous if the object is an instance of a data or a behaviour class.

## C. *Composition Pattern*

The composition pattern follows the well-known precept from OOP – "composition over inheritance". Similarly to union pattern, a "core" class per hierarchy in the model must be identified. Classes are then connected using association *is-a* instead of inheritance. The Union pattern basically merges an inheritance hierarchy into a single class. For the original subclass, it is required to have a link to its superclass(es), but the other direction is optional unless the generalization set is complete or the superclass is abstract.

The final implementation of this pattern is based on the improved traditional OOP inheritance. Instead of inheritance, i.e., specification of superclasses, all superclasses from the conceptual model are instantiated during the object initialization. During this step, a bidirectional link must be made to allow navigation from both superclass and subclass instances. The "core" class must be again chosen to allow creation of composed object using multiple subclasses, e.g., an instance of `Person` that is also an `Employee` and a `Man`.

```
class Delegation:

    def __init__(self, p_name, a_name):
        self.p_name = p_name
        self.a_name = a_name

    def __get__(self, instance, owner):
        p = getattr(instance, f'{self.p_name}')
        a = getattr(p, self.a_name) if p else None
        return a(instance) if callable(a) else a

    def __set__(self, instance, value):
        p = getattr(instance, f'{self.p_name}')
        setattr(p, self.a_name, value)

class LivingBeing:

    location = Delegation('_p_locatable',
    ↪   'location')

    def __init__(self, *, birthdate, _c_person=None,
    ↪   _p_locatable=None, **kwargs):
        self._p_locatable = _p_locatable or
        ↪   Locatable(_c_living_being=self,
        ↪   **kwargs)
        self._c_person = _c_person
        self.birthdate = birthdate

    ...
```

Figure 5. Part of composition pattern implementation

The example in Section III-C shows that we also incorporated a *Delegation* descriptor. Although it results in repetition when defining where to delegate, it clearly describes the origin of a method or an attribute, and it can be generated easily. With the fact that these parts can be generated, combinatorial effects related to renaming or other changes of methods and attributes used for delegation are mitigated. The diamond problem is solved directly by passing child and parent class objects as optional arguments during initialisations. It could also be solved using metaclasses, but as this code can be generated, it allows higher flexibility and eventual overriding.

As the built-in MRO is not used, the resolution must be made manually on the model level similarly to the Union pattern, i.e., to decide what overrides and what is overridden. By replacing inheritance with bidirectional links, we managed to significantly limit combinatorial effects, but in exchange for the price in requiring additional logic and moving the MRO into the model itself. Unfortunately, this implementation needs also to incorporate model-consistency checks, as we do not enforce multiplicity in child-parent links according to the pattern design.

### D. Generalisation Set Pattern

This pattern enhances the Composition pattern by adding particular constructs that encapsulate logic regarding generalisation sets. Inheritance relation is not transformed into *is-a* association but into connection via a special entity that handles related rules, such as complete or disjoint constraints and cardinality. As we present in Section III-D this helps to remove shortcomings of the Composition pattern and its difficult links and composed-object instantiation. Instead of multiple child links, there is just one per Generalisation Set (GS), and parent links are changed accordingly. An object of `GS` class maintains the inheritance and ensures the bi-directionality of links.

```
class Delegation:

    ...

    def __get__(self, instance, owner):
        gs = getattr(instance, f'{self.gs_name}')
        p = getattr(gs, f'{self.p_name}')
        a = getattr(p, self.a_name) if p else None
        return a(instance) if callable(a) else a

    ...


class GS_ManWoman:

    _gs_name = '_gs_man_woman'

    disjoint = True
    complete = True

    def __init__(self, person, man=None,
    ↪   woman=None):
        self.person = person
        self.man = man
        self.woman = woman
        self.update_links()

    def update_links(self):
        setattr(self.person, self._gs_name, self)
        if self.man is not None:
            setattr(self.man, self._gs_name, self)
        if self.woman is not None:
            setattr(self.woman, self._gs_name, self)

    ...
```

Figure 6. Generalisation Set implementation example

Introduction of an intermediate object to encapsulate inheritance and related constraints adds complexity in two aspects. First, the diamond problems must be still treated by sharing superclass objects in the hierarchy for eventual reuse. Second, the delegation must operate with the intermediary object when accessing the target child (or parent) object. However, solutions to these issues can be also generated directly from the model and in principle – despite their complexity – they do not hinder evolvability.

Finally, this solution (if entirely generated from a model) is the most suitable, since it limits combinatorial effects and allows to efficiently check consistency with the model in terms of inheritance and generalisation set constraints. Although in some cases, the `GS` object is not adding any value (e.g., a single child and a single parent case), implementing a combination of a generalisation set and composition patterns would make the software code harder to understand. Unity in implementation of conceptual-level inheritance is crucial here.

## IV. Implementation Summary and Future Research

In this section, we summarize and evaluate achievements of our research. Based on our observations and implementation of inheritance using patterns, we evaluate inheritance in Python. The patterns and its key aspects are compared in Table I. Then, we also describe the future steps that we plan to do as follow-up research and projects based on outputs described in this paper.

### A. Resulting Prototype Implementation

We demonstrated our implementation of all four previously designed patterns. Mostly, results and related usability options are consistent with the design. The more we minimize or constrain combinatorial effect, the more complex and hard-to-use (in terms of working with final objects) the implementation gets. We were able to simplify use of objects for the price of repetition and use of special constructs for delegation. Contrary to the original patterns design, we were not able to efficiently combine multiple patterns together based on various types of inheritance used in the model. As a result, our implementation of the most complex generalisation set pattern is suggested as a prototype of how inheritance may be implemented if one wants to avoid combinatorial effects while still needing to capture inheritance in a generic way for models of any size and complexity.

### B. Evolvability of Python Inheritance

During the implementation of the patterns, it became obvious that even high flexibility of programming language and allowed multiple inheritance do not help in terms of coupling and combinatorial effects caused by using class-based inheritance. With a simple real-world conceptual model, we were able to show how the combinatorial explosion endangers the evolvability of software implementation. MRO algorithm used in Python does not help with limiting combinatorial effects. Rather it is the opposite since order in which superclasses are enumerated significantly influences implementation behaviour. Also, it makes harder to combine overriding from two superclasses, for example, both class `A` and `B` implement methods `foo` and `bar` but subclass `C` cannot inherit one from `A` and other from `B` (solution is to override both and call it from subclass manually).

TABLE I. COMPARISON OF THE INHERITANCE IMPLEMENTATION PROTOTYPES

| Implementation | Classes* | Extra constructs | CE-handling | Issues |
|---|---|---|---|---|
| Traditional | $N + 2^N$ | 0 | none | initialization, order of superclasses, uncontrolled change propagation |
| Traditional + init_bases | $N + 2^N$ | init_bases function | shared initialization | shared attributes across hierarchy, order of superclasses, uncontrolled change propagation |
| Union pattern | 2 | Delegation class | shared class (merged) | Separation of Concerns violated, maintainability, discriminators |
| Composition pattern | $N$ | Delegation class | shared initialization, delegation | manual handling of GS constraints, added complexity (for humans) |
| GS pattern | $N + 1$ | Delegation class, GS helpers | shared initialization, delegation | added complexity (for humans) |

(*) per single hierarchy of $N$ classes, worst case (all combinations needed)

On the other hand, the flexibility of Python proved to be useful while we were implementing the patterns. Thanks to magic methods, descriptors, and metaclasses, the final implementations allow creating easy-to-use and inheritance-free objects even though underlying complex relations with constraints are needed as shown in the examples. Notwithstanding, such possibilities of Python are similar to constructs and methods in other languages (e.g., reflection). While trying to implement the patterns efficiently, we concluded that generating implementation from a model is crucial for evolvability regardless of what technologies are used, as a lot of repetition is needed.

### *C. Production-Ready Technology Stack*

The goal of the paper was to use Python to show reference patterns implementation prototypes. The next step may be to leverage the lessons learned to formulate a single transformation description using production-ready technology stack, e.g., Java EE. This final transformation of conceptual-level inheritance should allow simple extensibility and customizations. Moreover, it should cover all possibilities with respect to the underlying modelling language. This language does not have to be OntoUML used in this paper; however, it must be expressive enough to capture all the necessary details for a correct implementation – for instance, hierarchy "core" classes.

### *D. Model-Driven Development*

Having a final, production-ready, and well-described transformation of conceptual-level inheritance into implementation, it is not expected that it will be used to manually write source code. An essential part would be generation of a source code directly from the model capturing the inheritance together with domain knowledge. Normalized Systems (NS) theory [1] and related tooling can provide a way here using the so-called expanders. The challenge here would be to encode inheritance in the NS models directly or propose its extension, on the other hand we can expect flexibility and guarantees in terms of software evolvability.

## V. CONCLUSIONS

In this paper, we analysed and demonstrated the evolvability of inheritance in Python using already-designed implementation patterns. Although Python offers multiple inheritance based on the method resolution order algorithm, software written in Python that uses inheritance suffers from coupling and related combinatorial effects similarly to software in other languages. Nevertheless, thanks to the Python's flexibility and ability to redefine core constructs, we managed to implement the conceptual-level inheritance implementation patterns easily with minimisation of combinatorial effects, while maintaining code readability. The suggestions for future development go mostly in direction of generating a production-ready software systems from conceptual models, where inheritance is used as a natural construct used to reflect real-world domains.

ACKNOWLEDGEMENTS

This research was supported by the grant of Czech Technical University in Prague No. SGS20/209/OHK3/3T/18.

## REFERENCES

[1] Herwig Mannaert, Jan Verelst, and Peter De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Kermt (Belgium): Koppa, 2016.

[2] Marek Suchánek and Robert Pergl, "Evolvability Evaluation of Conceptual-Level Inheritance Implementation Patterns," in PATTERNS 2019, The Eleventh International Conference on Pervasive Patterns and Applications, vol. 2019. Venice, Italy: IARIA, May 2019, pp. 1–6, [retrieved: Aug, 2020]. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=patterns_2019_1_10_78001

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

[4] Pierre Carbonnelle, "PYPL: PopularitY of Programming Language," Feb 2020, [retrieved: Aug, 2020]. [Online]. Available: http://pypl.github.io/PYPL.html

[5] Python Software Foundation, "Python 3.8.0 Documentation," 2019, [retrieved: Aug, 2020]. [Online]. Available: https://docs.python.org/3.8/#

[6] David Hilley, "Python: Executable Pseudocode," [retrieved: Aug, 2020]. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.7674&rep=rep1&type=pdf

[7] Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, ser. C++ in-depth series. Addison-Wesley, 2001.

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

[9] Giancarlo Guizzardi, Ontological Foundations for Structural Conceptual Models. Centre for Telematics and Information Technology, 2005.

[10] Alan Borning, "Classes versus prototypes in object-oriented languages." in FJCC, 1986, pp. 36–40.

[11] John Hunt, "Class Inheritance," in A Beginners Guide to Python 3 Programming. Springer, 2019, pp. 211–232.

[12] Fawzi Albalooshi and Amjad Mahmood, "A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code," International Joutnal of Advanced Computer Science and Applications, vol. 8, no. 6, 2017, pp. 109–116.

[13] Matteo Orru et al., "How Do Python Programs Use Inheritance? A Replication Study," in 2015 Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2015, pp. 309–315.

[14] Ravi Chugh, Patrick M Rondon, and Ranjit Jhala, "Nested refinements: a logic for duck typing," ACM SIGPLAN Notices, vol. 47, no. 1, 2012, pp. 231–244.

[15] John Hunt, Advanced Guide to Python 3 Programming, ser. Undergraduate Topics in Computer Science. Springer International Publishing, 2019.