Towards Expanding Conceptual-Level Inheritance Patterns for Evolvable Systems

Marek Suchánek
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
ORCID: 0000-0001-7525-9218
marek.suchanek@fit.cvut.cz

Robert Pergl
Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
ORCID: 0000-0003-2980-4400
robert.pergl@fit.cvut.cz

***Abstract*—Inheritance as a relation for expressing generalisations and specialisations or taxonomies is natural for human perception and essential for conceptual modelling. However, it causes evolvability problems in software implementations. Each inheritance relation represents a tight coupling between entities with uncontrolled propagation of changes. Such a coupling leads to a combinatorial effect or even a combinatorial explosion in the case of complex hierarchies with multiple inheritances. This extended paper uses our analysis of multiple inheritance and method resolution order in the Python programming language to design and demonstrate code generation techniques with the use of inheritance patterns. The analysis is based on the design of inheritance implementation patterns from our previous work. Our design shows how can be conceptual-level inheritance implemented efficiently in software systems. The proposed design is demonstrated with Python (sometimes called "executable pseudocode"). However, it can be implemented with other object-oriented programming languages, even those that do not support multiple inheritance. The resulting design and prototype of expanders for inheritance patterns are ready for application in practice and further use within different technology stacks.**

***Index Terms*—Multiple Inheritance; Python 3; Evolvability; Method Resolution Order; Composition Over Inheritance.**

I. Introduction

This paper extends the previous conference paper [1] where inheritance in Python has been analysed, and conceptual-level inheritance patterns implementation in Python has been introduced. It provides more detailed information about the design and implementation by both adding new sections and extending the existing ones. Moreover, it includes code generation design and prototype previously outlined as future work.

Inheritance in the software engineering discipline is a commonly used technique for 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, furniture, and flammable object. On the other hand, in Object-Oriented Programming (OOP) software implementations, inheritance is typically used or even misused to re-use methods and attributes purely. In OOP, it causes so-called ripple effects violating evolvability of software [2]–[4].

The Python programming language is the most popular general-purpose and multi-paradigm programming language [5]. It is dynamically typed but allows multiple inheritance and also type hints [6]. Sometimes, Python is also called "executable pseudocode" thanks to its easy-to-read indentation-based syntax and versatility [7]. It is widely used for prototyping due to its flexibility and then rewritten into enterprise-ready implementation (e.g., using Java). For the implementation of conceptual-level inheritance in OOP, there are several patterns suggested in our previous work [3]. Although the patterns are compared and evaluated, implementation examples and empirical proofs are not yet provided.

Whenever there is a repeated pattern in software implementation, it strives for either re-use via generalisation, code generation or a combination of both. The code generation techniques instantiate patterns using code templates. The produces code can be then used in a more extensive code base and further customised. Normalised Systems use a code generation technique called expansions (the templates are called "Expanders"). One of the core advantages is the produces code's evolvability as it supports extensions using features and custom code insertions. Moreover, the design and provided tooling is verified by large-scale and real-world use cases. We strive to implement Expanders for the proposed inheritance patterns in Python. It should prove that the patterns can be used to implement conceptual-level inheritance without combinatorial effects with utilizing code generation.

In this extended paper, we design 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 designed to compare and demonstrate the additional complexity versus complexity caused by combinatorial effects. Then, with the patterns prepared, we design and prototype the expanders. First, Section II outlines the Design Science Research methodology that is followed within this research. Section III acquaints the reader with terminology and the overall context. In Section IV, we analyse how traditional inheritance works in Python and then describe the design and implementation of each pattern. Subsequently, Section V describes the patterns expansion design, and Section VI briefly

describes the implemented prototype and results. Finally, Section VII summarises and evaluates the expanders in contrast to traditional inheritance, describes the new experience with developing expanders for Python, and suggests future research.

II. Methodology

This research follows the Design Science Research (DSR) methodology [8]. It designs the artefacts based on a three-cycle view [9]. The knowledge base represents the NS theory and expanders, previous work on inheritance, and conceptual-level and software-level inheritance. On the other side, the environment is the Normalized Systems development, where we want to deliver the artefacts to allow modelling with inheritance. However, without causing combinatorial effects in the implementation – those are the requirements according to DSR.

The artefacts need to be evaluated and improved in design cycle iterations. The evaluation considers the measure of eliminations of combinatorial effects as well as usability (clarity, versatility, and extensibility). In this paper, we present the final artefacts of this procedure; however, we also mention key improvements done during the iterations.

We set the following research goals that the designed artefact will address:

- G1 – Design and implementation of the conceptual-level inheritance patterns [3].
- G2 – Design a code generation for the proposed implementation.
- G3 – Prove that the designed code generation is feasible and versatile (i.e., can produce usable source code with the possibility to extend it).

Whereas G1 and G2 can be considered as traditional design artefacts, G3 is related to evaluation and actually demonstrating the value of artefacts supporting G1 and G2. Although we use Python in our research, the design must be done language-independent. If someone designs implementation of the patterns in other languages following our steps, the same outcomes should be achieved.

The structure of this paper follows the workflow for achieving the set goals in the natural order. First, we design the known patterns for implementation for G1. That gives us reference implementations that we strive to code-generate for G2, i.e., remove the burden of manual coding of complex structures for implementation where conceptual-level inheritance is natural and straightforward. Finally, with a designed code generation technique, we can prove and demonstrate the feasibility and versatility for achieving G3.

III. Related Work and Terminology

This section briefly introduces the related research and terminology required for our approach and provides an overview of the knowledge base in terms of DSR. It focuses on Normalized Systems and Expanders, inheritance on conceptual-level, and Python programming language that we selected for prototyping due to its versatility and other capabilities..

A. Normalized Systems Theory

Normalized Systems Theory [2] (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 [2] 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 [2] 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 [2] [3] [10].

B. Normalized Systems Expanders

The code templates, together with mapping from NS models, are called Expanders. It allows the generation of any textual files from NS models. For software development, expanders to produce enterprise information systems in Java are used. However, one can also develop expanders to produce technical documentation of the system or graphical representation using SVG. The expanded code base is expected to be enhanced by adding custom code fragments called "craftings" to implement functionality that cannot capture using elements [11].

To avoid overwriting a system's craftings upon re-generation (or so-called rejuvenation), they should be harvested. The harvesting procedure basically goes through the code base and stores both insertions and extensions in the designed location. Then, when a system is rejuvenated, it takes the selected expanders, NS model of Elements, technical details and harvested craftings to generate the codebase. There might be several reasons for re-generation: a change in the model, updated expanders, or a different underlying framework (cross-cutting concern) [2], [11], [12].

Finally, the expanders can also be variable using *features*. It aims to include pieces of code to multiple expanders. Then, a feature can be enabled by an option from the NS model (e.g., by specifying a particular data option for a data element) or another condition. It typically adds some logic to multiple artefacts at once to extract cross-cutting concerns from the expanders (decoupling). As expanders are very variable and

can become complex, there is also a prepared way to test them easily [11].

C. 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 [13] 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 [3] [14].

D. 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 [15]. 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 [13] [16].

Both single and multiple inheritance can be used for reuse of source code. In [13], a clear explanation between essential and accidental (i.e., purely for reuse) use of inheritance is made. Moreover, [17] shows how multiple inheritance leverages reuse of code in OOP, including its consequences. According to [18], Python programs use widely (multiple) inheritance and it is comparable to use of inheritance in Java programs of the similar sample set.

E. 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 [6] [18].

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 [19], 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 [6] [20].

"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 [6] [16].

IV. Python Inheritance Analysis

In this section, we analyse how conceptual-level inheritance implementation patterns proposed in [3] 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. The proposed implementation of the patterns fulfils our goal G1.

For demonstration, we use a conceptual model depicted in Figure 1 using OntoUML [14]. 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.

Note that Figure 1 is a conceptual model and not an implementation model. For example, one may object that Man and Woman can be just an attribute and enumeration. That would be true only for traditional programming. In the conceptual model, that approach is not correct as Man and Woman are types of Person, not property. In programming, when considering evolvability, the reason is that enumeration does not allow evolution in a sense where items have own properties and change over time. Therefore, in Normalized Systems, there is no such concept of enumeration.

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.,

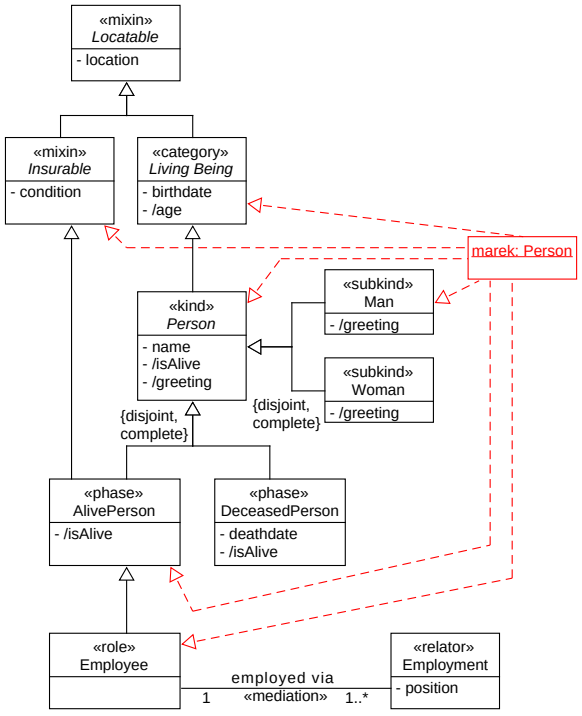

Fig. 1. Diagram of OntoUML example model with instance

```
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", ...)
```

Fig. 2. Part of the traditional inheritance implementation

`__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 Figure 2. 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`.

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 Figure 3. 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 Figure 3, 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`.

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 Figure 4. 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

```
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", ...)
```

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

`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. Figure 4 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.

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

```
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

# ...
```

Fig. 4. Part of union pattern implementation

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

# ...
```

Fig. 5. Part of composition pattern implementation

The example in Figure 5 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 Figure 6 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)

# ...
```

Fig. 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.

V. Inheritance Expanders Design

In this section, we describe the common design of the inheritance patterns code generation using Normalized Systems Expanders and related tooling. We do so for all of the patterns and again using Python programming language.

A. Capturing Inheritance in Models

The input models for expanders describe all the entities with their properties and relations. Although creating a custom meta-model for using expanders is possible, that is not part of our goals. Therefore, we use the NS meta-model of so-called Elements. Our focus is on Data Elements, which can be seen as counterparts to classes or entities from structural conceptual modelling. For data elements, we are interested in their fields – link fields representing relationships between data elements and value fields that are attributes with a data type.

Due to the evolvability issues, there is no way how to model inheritance directly with the current NS meta-model. Nevertheless, it is required to capture this somehow in a model, e.g., that a data element Person is a specialisation of a data element Living Being. For the same reason, there is no way to model an abstract class (or entity) as it makes no sense without inheritance. Another complication comes with generalisation sets that are groups of inheritance relations.

NS meta-model provides a way to encode additional information using so-called options flexibly. Various options are based on the construct to which they are attached, e.g., field options are related to a certain field. We use these options to encode inheritance-related information in NS models:

- **Abstract Element** – Indication that a data element is abstract, i.e., must be further specialised. It is captured using a single data option on a data element – if present, it is an abstract data element.
- **Inheritance** – Indication that a link between two data elements forms an inheritance relation. There might be a field option stating that the target is a generalisation or specialisation on a link field.
- **Generalisation Set** – Instead of creating groups of relations and then stating if those are disjoint, complete, or both, we capture these atomically. On a link field with an inheritance indication, a field option might add if the relation is disjoint with other such fields (a field is uniquely identified by the name and the data element name).

B. Code Templates and Mapping

With the inheritance-related information encoded in NS models, the expanders can be designed. First, it is required to prepare a mapping that queries the information from a model and supplies it to corresponding code templates. Then, a code template creates a file based on these supplied inputs. The high-level design is shown in Figure 7.

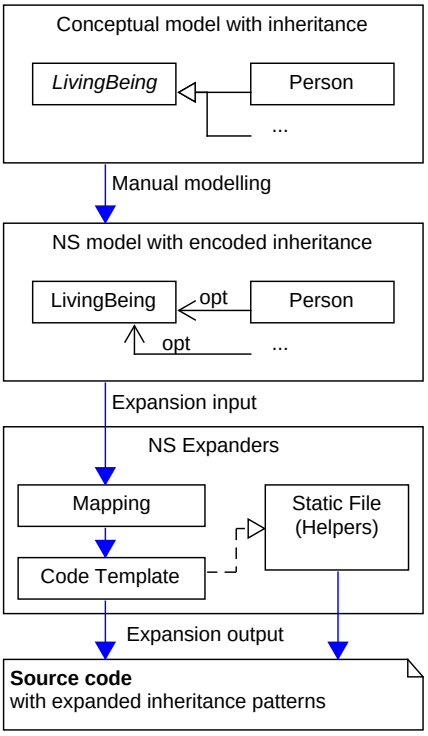

Fig. 7. High-level design of inheritance patterns expansion

For all data elements in a component, the template needs all data options, fields, and fields options. Other model parts are out of our scope. Data elements are turned into classes, with fields turned into instance attributes based on a specific inheritance pattern used. The patterns designed in the previous section are a basis for this step. All model-specific names are variable and queried from an input model.

There might be some additional static constructs, e.g., `Delegation` class or `init_bases` function, that requires no inputs from a model. Those common definitions are separated into static files, which are just copied. Code templates may rely on such definitions – import them from a known location and use them. That is depicted by the dashed arrow in Figure 7.

As our purpose is to generate class hierarchies in Python based on the design presented in the previous section, there

will be one expander per pattern. An expander, in our case, produces a single Python file with all classes, including hierarchies for a given input model. It simplifies both generation and manipulation with the result as we are comparing different inheritance patterns. Moreover, it is natural for Python to have all data classes of a single component inside one file in contrast to Java with file-per-class.

C. Anchors in Expanders

Anchors are an essential part of expanders. We follow the best practices in using anchors. It delimits generated fragments based on a particular entity, e.g., when a class is formed from a data element, it is delimited by anchor comments. Similarly, it is done for a block of fields assignments or method parameters that are generated from a model.

Another type of anchor is related to custom code. Wherever it may be necessary to add custom code, there should be such an anchor delimiter. Then, expanded code can be easily enhanced with insertions that can be later harvested and eventually re-injected upon rejuvenation. These anchors are added on module, class, and method levels. To maintain code readability after generation, there are multiple anchors for each level, e.g., at the top of a module for imports, before class declarations, and finally, at the end after all class declarations.

D. Features in Expanders

Similarly to anchors, expanders may also contain so-called features. It can be used as optional parts of expanders or for potential future extensions of expanders. We follow the same principles and add anchors so that future extensions may add code fragments to any part of the output source code. That will simplify the future prototyping of enhancements and adding optionally generated code. However, we did not identify any practical use of features for this work as we aim to generate specific inheritance patterns described in the previous section.

VI. Expanding Inheritance Patterns

The design described above outlines the common design for code generation of all inheritance patterns as part of G2. In this section, we provide additional details in the design specific to a certain pattern. It also addresses G3 as we implement and demonstrate the expanders. The implementation also serves evaluation purposes according to the DSR methodology.

An expander takes a model of NS elements together with technical details (configuration) as inputs. A mapping (part of expander) defines what is extracted from the model and eventually transformed (e.g., new derived variables). Finally, a code template (part of expander) is filled based on the mapping with data originating in the input model and files are produced and stored according to the expander and project configuration. The input model can be seen in an XML representation, but that is just a projection for serialization; from our perspective, the syntax does not matter; we care about semantics.

A. Traditional Inheritance Expanders

The first presented pattern uses the traditional inheritance. In the first stage, the inheritance is done using the multiple inheritance with MRO that is standard in Python. That will serve as the reference for other patterns which should "work the same" but without using the inheritance directly for evolvability reasons. Moreover, at this stage, the expander is the most straightforward to design:

1) All data elements become classes.
2) For each data element, all superclasses must be resolved by checking link fields for the specialisation option.
3) For each data element, all value fields and non-inheritance link fields are instance attributes (using initialisation `__init__`).
4) For each data element, all calculated fields become a method with property decorator and anchors for implementing the calculation.

The second stage is to generate classes representing all possible combinations of classes (e.g., `EmployeeMan` from Figure 2). It clearly shows the potential of a combinatorial explosion. As an optimisation, we exclude the unnecessary combinations with disjoint classes for generalisation sets:

1) For each class in a single hierarchy, list all combinations for its specialisations.
2) Remove combinators where any two classes are disjoint.
3) Create classes from the list, with the corresponding superclasses.
4) List all properties for the class and remove duplicates.
5) Create initialisation that calls correct initialisation of the parents.

When using the proposed '`init_bases`' function, the two list steps are significantly reduced. Only own fields are initialised directly and other are passed to this function. It makes both expanders and resulting code simpler and shorter.

B. Union Pattern Expanders

Expansion with the union pattern has the opposite procedure when compared to the previous one. It requires resolving the "folding" of class hierarchy into a single one. It must solve several edge cases such as hierarchies with multiple ultimate parents or the creation of discriminator attributes. Then all fields are merged into a single union class based on the previous expander design.

1) Find a root in a hierarchy; if there is not a single root, add an artificial one.
2) Create data element for each hierarchy.
3) Create a discriminator instance attribute for every disjoint generalisation set in the hierarchy.
4) Add all value, link, and calculated fields as for traditional inheritance into the single class.

The logic about solving the potential issue with the hierarchy and folding it to a single class must be robust, and it is resulting in quite complex code in expanders and mapping. Moreover, the resulting classes are also very complex and hard to use, as we discuss the results further in the next

section. We did not use the `Delegation` feature here as it adds complexity for methods, but at this point, there are no generated methods – only attributes.

C. Composition Pattern Expanders

The composition pattern that follows the "composition over inheritance" rule with the use of delegation can be expanded more easily than the union pattern. It creates a class per data element as for traditional inheritance, but instead of using inheritance, it uses the `Delegation` descriptor. For parents and children, there are additional special attributes just as shown in Figure 5. The steps are as follows:

1) All data elements become classes.
2) For each data element, all superclasses are represented as `_p_`-prefixed attributes.
3) For each data element, all subclasses are represented as `_c_`-prefixed attributes.
4) For each data element, all value fields and non-inheritance link fields are instance attributes (using initialisation `__init__`).
5) For each data element, all calculated fields become a method with property decorator and anchors for implementing the calculation.
6) For each data element, all inherited fields are added using `Delegation` description on the class level.

Steps 1, 4, and 5 are the same as for the traditional inheritance expanders. It shows that only the inheritance part is replaced by delegation and necessary preparations (of the linking attributes). Also, note that this pattern does not take into account disjoint nor complete constraints of generalisation sets.

D. Generalisation Set Pattern Expanders

As explained, the generalisation set pattern can be seen as an extension to the composition pattern. However, in terms of expansion, there is a significant overhead needed to find the generalisation sets in the input model. It must group the inheritance relations where disjoint, and union constraints through options are used. Based on these groups, additional `GS` classes can be generated as shown in Figure 6.

It creates classes from data elements as for composition pattern but uses `GS` class for `_p_`-prefixed and `_c_`-prefixed where it exists. The added complexity in both expanders and resulting code turned to be very significant. One of the main issues was the different handling of single-class generalisation sets and multiple-class generation sets. Therefore, we create `GS` classes also for single-class sets to keep consistency across source code.

E. Expanders Implementation

With the design described in the previous section and steps for each pattern expander, the implementation did not require any additional knowledge. The steps are represented in "code" of mappings that queries and transforms the information from an input model to desired form, e.g., a list of data elements in a component or a list of superclasses for a data element. Although the expressiveness of XML mappings in expanders is limited, it was entirely sufficient for our implementation. Figure 8 shows a basic mapping common for all of our expanders.

```
<mapping>
  <value name="componentName"
  ↪  eval="component.name"/>

  <list name="dataElements"
  ↪  eval="component.dataElements"
  ↪  param="dataElement">
    <value name="name" eval="dataElement.name"/>

    <list name="valueFields"
    ↪  eval="dataElement.fields" param="field"
    ↪  filter="field.valueField neq null">
      <value name="name" eval="field.name"/>
      <value name="type" eval="field.valueField.
      ↪  valueFieldType.name"/>
    </list>

    <list name="linkFields"
    ↪  eval="dataElement.fields" param="field"
    ↪  filter="field.linkField neq null">
      <value name="name" eval="field.name"/>
      <value name="targetElement"
      ↪  eval="field.linkField.
      ↪  targetElement.name"/>
      <value name="lnType" eval="field.linkField.
      ↪  linkFieldType.name"/>
    </list>
   </list>

   <!-- ... -->
</mapping>
```

Fig. 8. Fragment of a basic mapping

Whereas the mapping file queries and transforms the information to be supplied into a code template, the code template itself may contain a more complex transformation. It uses the String Templates well-known from Java that are capable of using conditions, loops, functions, and many other constructs. For example, filtering the non-inheritance link fields has been done in templates when generating the corresponding code (instance attributes). In summary, the implementation based on the design described above turned to be straightforward without any unexpected issues. A fundamental code template fragment from traditional inheritance expanders is shown in Figure 9.

F. Expanders Comparison

Both the design of the expanders and their implementation provided valuable knowledge about the patterns and verification of the theoretical comparison made in the previous work. In terms of complexity (computed based on logic and mathematical operations needed in mappings and templates), the composition pattern shows the best results even when compared to the traditional inheritance. The leading cause has been identified in omitting the disjoint and complete constraints.

The resulting code varies a lot based on the input model and hierarchies. Generally, traditional inheritance suffers from

```
delimiters "$", "$"

base() ::= <<
# Generated traditional inheritance patterns for
# component "$componentName$"
import Delegration from .helpers
@anchor:imports
# anchor:custom-imports:start
# anchor:custom-imports:end

# anchor:dataElements:start
$dataElements:dataElementEntry()$
# anchor:dataElements:end

# anchor:custom-moduleEnd:start
# anchor:custom-moduleEnd:end
>>

dataElementEntry(de) ::= <<

class $de.name$(
    # anchor:de-$de.name$-bases:start
    $deSuperclasses(de)$
    # anchor:de-$de.name$-bases:end
):

    $deInitializer(dataElement)$

    @anchor:methods
    # anchor:custom-$de.name$-methods:start
    # anchor:custom-$de.name$-methods:end

    ...


>>
```

Fig. 9. Fragment of a code template

a combinatorial explosion. Union pattern encapsulates the explosion inside a single class which is unmaintainable when it "folds" a more complex hierarchy with several generalisation sets. It may be helpful for smaller and mostly linear hierarchies. The code with generalisation sets pattern is the most complex in terms of additional classes and navigation between subclasses and superclasses.

VII. Evaluation and Discussion

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. Initial Prototype Implementation

We demonstrated our implementation of all four previously designed patterns as set in the goal G1. 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. Expanders Prototype

To also fulfil the goals G2 and G3, we designed code generation for all of the inheritance patterns (in Python programming language) and proved its feasibility by implementing them with the use of Normalized Systems Expanders. It revealed several implementation issues and helped to refine the design based on the DSR design cycle. It verified the hypothesis that union and generalisation set patterns create over-complex source code that is hard to maintain and extend. The most suitable composition pattern keeps flexibility, and additional constraints can be supplied using custom insertions and extra logic where needed.

The expanders clearly mitigate the burden of additional complexity brought by inheritance implementation patterns as that is something generated and managed only in the code templates. A change in the inheritance can be easily projected into code by re-generating. The burden that remains is related to custom code, where a programmer must use the navigation between superclasses and subclasses that may not be natural, but the patterns were designed to make this as effortless as possible.

C. 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).

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

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)

regardless of what technologies are used, as a lot of repetition is needed.

D. Future Work – Inheritance Modelling for NS

As explained and practically demonstrated, we had to encode the inheritance relations in an NS model using fields options. A software analyst must create a link between data elements and then mark it as an inheritance in terms of modelling. In visualisations of such a model, it still looks like a regular relation. A potential next step would be to propose an extension to the NS Elements meta-model to directly model inheritance relations. That would require enhancement in the meta-model as well as incorporation in existing expanders.

E. Future Work – Production-Ready Technologies

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.

F. Future Work – Inheritance in UI/UX

Another step or aspect of implementing conceptual-level inheritance lies in creating a related user interface compatible with data and behaviour encapsulation. After there is MDD-based generation of code from a model for backend service, it should also be considered how to generate UI for related frontend. To elevate user experience, special relations made for implementing inheritance should look different than others. Moreover, the user should be able to easily create an object with a selection of possible subclasses closer to the model rather than to the implementation on the backend.

VIII. Conclusions

This extended paper proposes code generation design for conceptual-level inheritance patterns that minimise the issues caused by combinatorial effects. We analysed and demonstrated the evolvability of inheritance in Python using already-designed implementation patterns. Due to Python's flexibility and ability to redefine core constructs, we managed to implement the conceptual-level inheritance implementation patterns while maintaining code readability and maintainability. Further, we designed code generation of those constructs from models using Normalized Systems Expanders and existing tooling. It proved the feasibility of implementing the patterns in larger codebases. Although we used Python for this work, it can be used as a basis for implementation in other object-oriented languages. Finally, future work has been outlined. The following challenges are identified in the generation of user interface fragments, e.g., forms or details, for entities with inheritance.

Acknowledgment

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

References

[1] Marek Suchánek and Robert Pergl, "Evolvability Analysis of Multiple Inheritance and Method Resolution Order in Python," in *PATTERNS 2020, The Twelfth International Conference on Pervasive Patterns and Applications*, H. Mannaert, I. Pu, and J. Daykin, Eds. IARIA, 2020, pp. 19–24, Accessed: 24 Nov 2021. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=patterns_2020_2_10_79

[2] 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.

[3] 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, Accessed: 24 Nov 2021. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=patterns_2019_1_10_78001

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

[5] Pierre Carbonnelle, "PYPL: PopularitY of Programming Language," Feb 2020, accessed: 15 Nov 2021. [Online]. Available: http://pypl.github.io/PYPL.html

[6] Python Software Foundation, "Python 3.8.0 Documentation," 2019, [online]. Accessed: 12 Nov 2021. [Online]. Available: https://docs.python.org/3.8/#

[7] 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

[8] A. R. Hevner, "Design science research," in *Computing Handbook, Third Edition: Information Systems and Information Technology*, H. Topi and A. Tucker, Eds. CRC Press, 2014, pp. 22: 1–23.

[9] ——, "The three cycle view of design science," *Scand. J. Inf. Syst.*, vol. 19, no. 2, p. 4, 2007. [Online]. Available: http://aisel.aisnet.org/sjis/vol19/iss2/4

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

[11] NSX. Prime Radiant Online. Accessed: 2 Nov 2021. [Online]. http://primeradiant.stars-end.net/foundation/.

[12] H. Mannaert, P. De Bruyn, and J. Verelst, "On the interconnection of cross-cutting concerns within hierarchical modular architectures," *IEEE Transactions on Engineering Management*, 2020.

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

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

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

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

[17] 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, pp. 109–116, 2017.

[18] 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.

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

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