Evolvability Analysis of Multiple Inheritance and Method Resolution Order in Python

The Twelfth International Conference on Pervasive Patterns and Applications
PATTERNS 2020
October 25, 2020 to October 29, 2020 - Nice, France

Marek Suchánek (FIT CTU in Prague, FBE UA)
marek.suchanek@fit.cvut.cz

Robert Pergl (FIT CTU in Prague)
Introduction and Outline

• Speaker: Marek Suchánek
  • PhD student at FIT CTU in Prague and FBE University of Antwerp (joint degree)
  • Member of CCMi (Centre for Conceptual Modelling and Implementation)

• Presentation
  1. Concept of Inheritance
  2. Inheritance in Python 3
  3. Inheritance Implementation Patterns
  4. Conclusions and future work
Inheritance in Real-World

• Natural concept in real-world
• Key to evolution (passing properties to next generations)
• Taxonomies (common properties of species)
• Allows us to form abstractions and relate them
• Everyone understands how it works
Inheritance in Conceptual Models

• Conceptual models are used to describe reality
• ... including inheritance as a key principle
• Different languages provide different/various ways
  • OntoUML
  • UML (class diagram)
  • OWL (owl:subclass)
  • ER (IS-A hierarchies)
• Subtyping, subclassing, etc.
• May be „detached“ from reality, harder to understand
Inheritance in Software Implementation

- In OOP intended to reflect real-world inheritance
- Often abused or mis-used purely for re-use
- DRY principle but with combinatorial effects
- “Composition over inheritance”
- Various implementations and behavior in different languages
- Single inheritance, multiple inheritance, prototyping, interfaces
- Hard to understand (and use correctly)

```python
class AlivePerson(Person, Insurable):
    def __init__(self, name, birthdate, condition):
        Person.__init__(self, name, birthdate)
        Insurable.__init__(self, condition)

@property
def is_alive(self):
    return True
```
Multiple Inheritance in Python 3

- Python 3 = language suitable for prototyping
- Allows to re-define behaviour of almost everything
- Method Resolution Order
- Concept of metaclasses
- Easy transition to other OOP languages in „production-ready“ stacks such as Java
Inheritance Implementation Patterns

• We described patterns how to implement inheritance based on related work:
  • Union pattern
  • Composition pattern
  • Generalization Set pattern

• Evaluated only on the conceptual-level

• Next goal was to investigate how to use them in implementation:
  • Elimination of combinatorial effects
  • Ease of implementation (overhead)
  • Flexibility for various use cases
Inheritance Implementation Patterns

- **Traditional Inheritance** (multiple with MRO)
- Order of subclassing matters, but the order is usually not modelled
- Changing a class affects all the (direct and indirect) subclasses
- Object can be instance only of a single class
- Imminent combinatorial effect (leading to „combinatorial explosion“)

```python
class AlivePerson(Person, Insurable):
    def __init__(self, name, birthday, location, condition):
        Person.__init__(self, name, birthday, location)
        Insurable.__init__(self, condition)

class Woman(Person):
    @property
def greeting(self):
        return f'Mrs. {self.name}'
```
Inheritance Implementation Patterns

- **Union Pattern**
  - Merges whole hierarchy into a single class
  - The "order" is captured while merging
  - Changes contained in the class
  - Discriminators to toggle subclasses
  - To share behavior, delegation can be used
  - Can be generated but causes issues when union classes should be split or merged

```python
class Person:
    def __init__(self, name, birthdate, location, condition):
        self.location = location
        self.condition = condition
        self.birthday = 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
```
Inheritance Implementation Patterns

- **Composition Pattern**
- Replaces inheritance with delegation
- Easy to generate from a model (may need order)
- Additional rules for generalization sets must be handled using custom code

```python
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'parent_{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'p_{self.p_name}')
        setattr(p, self.a_name, value)

class Person:

    condition = Delegation('insurable', 'condition')
    age = Delegation('living_being', 'age')

    def __init__(self, *, name, **kwargs):
        self._p_living_being = LivingBeing(_c_person=self, **kwargs)
        self._c_man = None
        self._c_woman = None
        self.name = name
```
Inheritance Implementation Patterns

- Generalization Set Pattern
  - Strives to include GS with composition
  - For a set of classes, special GS class may be added
  - It holds additional GS constraints and maintains integrity
  - Special treatment of overlapping generalization sets
  - More complex navigation for delegation
## Comparison

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Classes*</th>
<th>Extra constructs</th>
<th>CE-handling</th>
<th>Issue(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>$N + 2^N$</td>
<td>none</td>
<td>none</td>
<td>initialization, order of superclasses, uncontrolled change propagation</td>
</tr>
<tr>
<td>Traditional + init_bases</td>
<td>$N + 2^N$</td>
<td>Init_bases function</td>
<td>shared initialization</td>
<td>shared attributes across hierarchy, order of superclasses, uncontrolled change propagation</td>
</tr>
<tr>
<td>Union</td>
<td>2</td>
<td>Delegation class</td>
<td>shared class (merged)</td>
<td>Separation of Concerns violated, maintainability, discriminators</td>
</tr>
<tr>
<td>Composition</td>
<td>$N$</td>
<td>Delegation class</td>
<td>shared initialization, delegation</td>
<td>manual handling of GS constraints, added complexity (for humans)</td>
</tr>
<tr>
<td>GS</td>
<td>$N + 1$</td>
<td>Delegation class, GS helpers</td>
<td>shared initialization, delegation</td>
<td>added complexity (for humans)</td>
</tr>
</tbody>
</table>

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

• We revisited and prototyped the inheritance implementation patterns

• Focused on generation from model and maintainability

• Avoid order-related combinatorial effects by solving it upon transformation

• Other change-related combinatorial effect are partially avoided by delegation

Future work:

• Prototype and test expansion for inheritance with production-ready stack

• Inheritance in UI/UX (i.e., how to create instances for hierarchy)
Questions & Discussion