KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

4. Dědičnost

verze z 17. října 2024

1 Motivace

Programy vytvářené v předchozí přednášce trpěly problémem opakujícího se kódu, kterému by se dobrý programátor měl vyhnout. Podívejme se blíže na jeden příklad. Jednoduchá definice třídy `Entry` by vypadala následovně.

```
class Entry:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.change_handler = None

    def get_x(self):
        return self.x

    def set_x(self, x):
        self.x = x
        return self

    def get_y(self):
        return self.y

    def set_y(self, y):
        self.y = y
        return self

    def get_change_handler(self):
        return self.change_handler

    def set_change_handler(self, change_handler):
        self.change_handler = change_handler
        return self
```

Kód záměrně zjednodušujeme. Například vynecháváme kontroly při nastavování hodnot vlastností. Třída definuje atributy a vlastnosti `x`, `y` a `change_handler`. Pro jednoduchost neuvažujeme text pole. Zárodek třídy `Button` je podobný třídě `Entry`:

```
class Button:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.click_hander = None

    def get_x(self):
        return self.x

    def set_x(self, x):
        self.x = x
        return self

    def get_y(self):
        return self.y

    def set_y(self, y):
        self.y = y
        return self

    def get_click_hander(self):
        return self.click_hander

    def set_click_hander(self, click_hander):
        self.click_hander = click_hander
        return self
```

Obě třídy se shodují v definicích atributů a vlastností `x` a `y`.

2 Hierarchie tříd

Problém s opakujícím se kódem vyřešíme tak, že umožníme sdílet definice atributů a metod mezi třídami. Třída je bude dědit po svých předcích. **Přímého předka** třídy můžeme uvést v definici třídy:

```
class Class(ParentClass):
    def __init__(self):
        super().__init__()
        ⋮
    ⋮
```

Class: jméno nové třídy
ParentClass: jméno existující třídy

Třída `AtomicWidget` bude obsahovat společné definice atributů a metod tříd `Entry` a `Button`:

```
class AtomicWidget:
    def __init__(self):
        self.x = 0
        self.y = 0

    def get_x(self):
        return self.x

    def set_x(self, x):
        self.x = x
        return self

    def get_y(self):
        return self.y

    def set_y(self, y):
        self.y = y
        return self
```

Třídy `Entry` a `Button` budou mít za přímého předka třídu `AtomicWidget`:

```
class Entry(AtomicWidget):
    def __init__(self):
        super().__init__()
        self.change_handler = None

    def get_change_handler(self):
        return self.change_handler

    def set_change_handler(self, change_handler):
        self.change_handler = change_handler
        return self

class Button(AtomicWidget):
    def __init__(self):
        super().__init__()
        self.click_hander = None

    def get_click_hander(self):
        return self.click_hander

    def set_click_hander(self, click_hander):
        self.click_hander = click_hander
        return self
```

Pomocí vztahu přímého předka můžeme definovat následující tři vztahy.

1. Třída C je **předkem** třídy D, pokud je C přímým předkem D nebo C je přímým předkem třídy E a třída E je předkem třídy D.
2. Třída D je **přímý potomek** třídy C, pokud třída C je přímý předek třídy D.
3. Třída D je **potomek** třídy C, pokud třída C je předkem třídy D.

Třída získá (říkáme, že **zdědí**) definice všech atributů a metod svých předků.

Tedy třídy `Entry` a `Button` zdědily od třídy `AtomicWidget` definice atributů `x`, `y` a metod `get_x`, `set_x`, `get_y` a `set_y`. Například dostáváme:

```
>>> entry = Entry()
>>> entry.set_x(5)
<Entry object at 0x10b24ba10>
>>> entry.get_x()
5
>>> button = Button()
>>> button.set_x(5)
<Button object at 0x10ee116d0>
>>> button.get_x()
5
```

Pomocí vztahu přímého předka vzniká hierarchie tříd, která musí splňovat následující pravidlo nazývané **is-a**.

> Je-li třída D potomkem třídy C, pak věta „každé D je C“ musí dávat smysl.

Například třída `Entry` může být potomkem třídy `AtomicWidget`, protože každé textové pole je atomický ovládací prvek. Anglicky: *Every entry is an atomic widget.* Třída `Window` nemůže být potomkem třídy `AtomicWidget`, protože existuje okno, které není atomickým ovládacím prvkem. Dokonce žádné okno není atomickým ovládacím prvkem.

Rozšíříme si vztah přímé instance mezi objekty a třídami. Objekt O je **instancí** třídy C, pokud O je přímou instancí třídy C nebo O je přímou instancí třídy D a třída D je potomek třídy C. Vestavěný predikát `isinstance` rozhoduje, zda je objekt instancí třídy:

```
>>> isinstance(entry, AtomicWidget)
True
>>> isinstance(button, AtomicWidget)
True
>>> isinstance(button, Entry)
False
```

Jak jsme viděli v případě tříd `Entry` a `Button`, tak potomci tříd mohou definovat nové atributy a metody. Například třída `Entry` přidává vlastnost `change_handler`.

3 Přepisování metod

Uvažujme jednoduchou třídu `Group` pro skupiny ovládacích prvků:

```
class Group:
    def __init__(self):
        self.items = []

    def get_items(self):
        return self.items

    def set_items(self, items):
        self.items = items
        return self
```

a jejího potomka `ButtonGroup` pro skupiny tlačítek:

```
class ButtonGroup(Group):
    def __init__(self):
        super().__init__()

    def set_items(self, items):
        for item in items:
            if not isinstance(item, Button):
                raise TypeError("item is not a button")
        self.items = items
        return self
```

V instanci třídy `ButtonGroup` nyní existují dvě metody jména `set_items`. První je metoda přidaná třídou `Group` a druhá třídou `ButtonGroup`. Nastává otázka, jakou metodu vybrat pro obsluhu zprávy `set_items`.

Při zaslání zprávy ***`message`*** objektu se k obsluze vybere metoda objektu jména ***`message`*** přidaná třídou C pro kterou platí, že neexistuje metoda jména ***`message`*** přidaná třídou D, kde třída D by byla potomkem třídy C. Tedy k obsluze zprávy `set_items` se vybere metoda přidaná třídou `ButtonGroup`.

Pokud třída definuje metodu, kterou zdědila, říkáme, že ji **přepisuje**. Například třída `ButtonGroup` přepisuje metodu `set_items`.

Pokud metoda M přepisuje metodu N, můžeme v těle metody M zavolat metodu N následovně.

Výraz

```
super().message(arg1, arg2, ...) => value

  message: zpráva
  value, arg1, arg2, ...: hodnoty
```

v těle M zavolá metodu N s argumenty *receiver*, *arg1*, *arg2*, ..., kde *receiver* je příjemce zprávy. Hodnota *value* je návratová hodnota metody N. Říkáme, že uvedený výraz **volá přepsanou metodu**.

Upravíme metodu `set_items` třídy `ButtonGroup` tak, že zavolá přepsanou metodu:

```
def set_items(self, items):
    for item in items:
        if not isinstance(item, Button):
            raise TypeError("item is not a button")
    super().set_items(items)
    return self
```

Výraz `super().set_items(items)` zavolá metodu třídy `Group`:

```
def set_items(self, items):
    self.items = items
    return self
```

Již víme, že definice inicializace instance (`__init__`) je metoda. Pokud metodu `__init__` přepisujeme, musíme nejprve zavolat přepisovanou metodu výrazem `super().__init__()`. Metoda `__init__` ve třídě `ButtonGroup`:

```
def __init__(self):
    super().__init__()
```

pouze volá přepsanou metodu. Metodu `__init__` tedy můžeme odstranit:

```
class ButtonGroup(Group):
    def set_items(self, items):
        for item in items:
            if not isinstance(item, Button):
                raise TypeError("item is not a button")
        super().set_items(items)
        return self
```

4 Reakce na uživatele

Díky dědičnosti můžeme obsluhu kliknutí na tlačítko provést přímo v obsluze k tomu určené zprávy. Změníme definici třídy `Button`:

```
class Button(AtomicWidget):
    def button_clicked(self):
        return self
```

Vidíme, že metoda `button_clicked` pouze vrací příjemce. V potomcích třídy však můžeme obsluhu změnit. Například:

```
class MyButton(Button):
    def button_clicked(self):
        super().button_clicked()
        print("Kliknuto na tlačítko")
        return self
```

Nyní klik na tlačítko způsobí tisk:

```
>>> button = MyButton()
>>> button.button_clicked()
Kliknuto na tlačítko
<MyButton object at 0x1021d3620>
```

Podobně změníme definici třídy `Entry`:

```
class Entry(AtomicWidget):
    def entry_change(self):
        return self
```

Zpráva `entry_change` je určena k tomu, aby se zaslala poli při změně jeho textu. Můžeme ji v potomkovi přepsat:

```
class MyEntry(Entry):
    def entry_change(self):
        super().entry_change()
        print("Obsah pole se změnil")
        return self
```

a tím docílit reakce na změnu textu:

```
>>> entry = MyEntry()
>>> entry.entry_change()
Obsah pole se změnil
<MyEntry object at 0x1021d3f20>
```

5 Čistě objektové uživatelské rozhraní

Přiložený soubor `omw.pdf` obsahuje příručku k čistě objektové knihovně Object Micro Widget (`omw`) na tvorbu uživatelského rozhraní. Knihovna je podobná knihovně

`pomw`. Objekty neobsahují vlastnost `data` a reakce na uživatelský vstup je zařízena přepisováním vhodných zpráv.

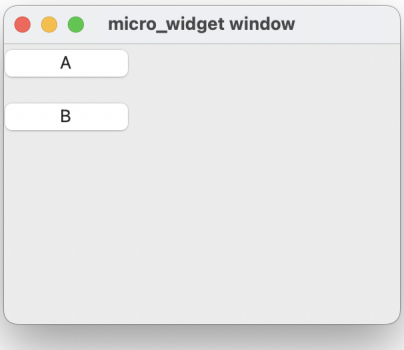
Třídy pro skupinu tlačítek bychom vytvořily jako přímého potomka třídy `Group`, u kterého přepíšeme metodu kontrolující, zda je prvek skupiny tlačítko:

```
class ButtonGroup(Group):
    def check_item(self, item):
        if not isinstance(item, Button):
            raise TypeError("items of a button group have to be buttons")
        return self
```

Program:

```
window = Window()
button1 = Button().set_text("A")
button2 = Button().set_text("B").move(0, 40)
group = ButtonGroup().set_items([button1, button2])
window.set_widget(group)
```

vytvoří okno s dvěma tlačítky:



Při změně prvků skupina provede kontrolu každého prvku tak, že pošle sobě zprávu `check_item` s prvkem jako argumentem. Proto pokus o nastavení prvků skupiny tlačítek obsahující popisek vyvolá výjimku:

```
>>> group.set_items([Label()])
TypeError: items of a button group have to be buttons
```

Definujeme si třídy přepisující obsluhy:

```
from omw import *

class MyButton(Button):
```

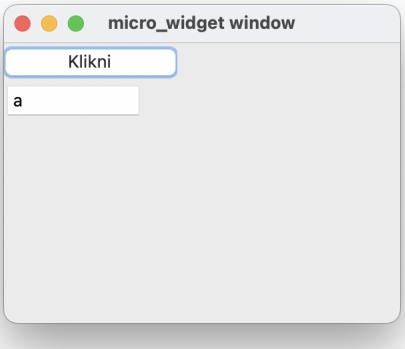
```
    def button_clicked(self):
        super().button_clicked()
        print("Kliknuto na tlačítko")
        return self

class MyEntry(Entry):
    def entry_change(self):
        super().entry_change()
        print("Obsah pole se změnil")
        return self
```

použijeme:

```
window = Window()
button = MyButton().set_text("Klikni")
entry = MyEntry().move(0, 30)
group = Group().set_items([button, entry])
window.set_widget(group)
```

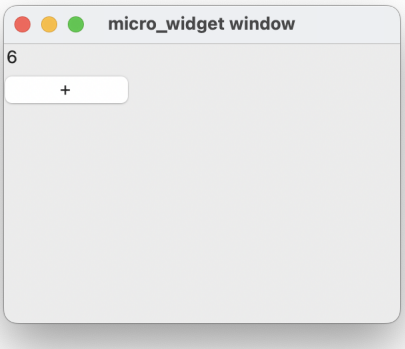
v zobrazeném okně změníme text pole a klikneme na tlačítko:



Vytiskne se:

```
Obsah pole se změnil
Kliknuto na tlačítko
```

Na závěr se podívejme, jak bychom vytvořili jednoduché počítadlo:



Vytvoříme jej jako potomka třídy `Group`. Každé počítadlo bude skupinou ovládacích prvků.

```
class Counter(Group):
    def __init__(self):
        super().__init__()
        self.value = 0
        label = Label()
        button = Button().move(0, 20).set_text("+")
        self.set_items([label, button])
        self.ensure_label_text()
```

Zaslání zprávy `ensure_label_text` zajistí, že popisek zobrazuje aktuální hodnotu počítadla. Definice obsluhy je uvedena níže. Přidáme definice vlastností pro čtení `label` a `button`.

```
    def get_label(self):
        return self.get_items()[0]

    def get_button(self):
        return self.get_items()[1]
```

Přidáme definici vlastnosti `value` určující hodnotu počítadla.

```
   def get_value(self):
        return self.value

    def set_value(self, value):
        self.value = value
        self.ensure_label_text()
        return self
```

Při nastavení hodnoty počítadla je potřeba zajistit, aby ji popisek zobrazoval.

```
    def ensure_label_text(self):
        label = self.get_label()
        value = self.get_value()
        label.set_text(str(value))
        return self
```

Na závěr přidáme metodu na inkrementaci počítadla.

```
    def inc_value(self):
        return self.set_value(self.get_value() + 1)
```

Vytvoříme počítadlo z výše uvedeného obrázku.

```
window = Window()
counter = Counter()
window.set_widget(counter)
counter.set_value(6)
```

Inkrementaci počítadla je nutné provést zasláním zprávy `inc_value`.

```
counter.inc_value()
```

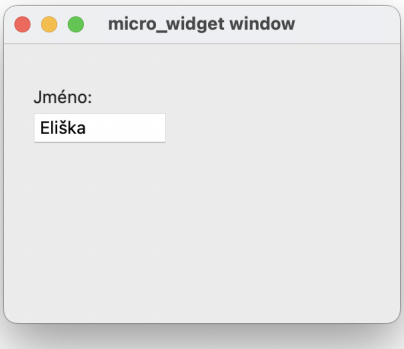
Jak správně zařídit, aby se počítadlo inkremetovalo stiskem tlačítka, se dozvíte na příští přednášce.

Všimněme si, že přestože je počítadlo skupinou, není vhodné, aby mu jeho uživatel nastavoval prvky zasláním zprávy `set_items`. Mohl by tak snadno počítadlo uvést do nekonzistentního stavu:

```
>>> counter.set_items([])
<Counter object at 0x106bf4170>
>>> counter.get_label()
IndexError: list index out of range
```

Otázky a úkoly na cvičení

1. Vytvořte třídu `EntryField` pro položky formuláře tvořené textovým polem s popiskem. Třída by měla být přímým potomkem vhodné třídy z knihovny Object Micro Widget. Zkontrolujte, zda nově přidaná třída splňuje pravidlo *is-a*. Příklad položky formuláře přidané do okna:



2. Přidejte do třídy `EntryField` definice vlastností pro čtení `label` a `entry`, jejichž hodnoty budou popisek a textové pole položky formuláře. Položku formuláře z předchozího snímku by tedy bylo možné vytvořit následovně.

```
ef = EntryField()
ef.get_label().set_text("Jméno:")
ef.get_entry().set_text("Eliška")
```

3. Zařiďte, aby položky formuláře měly vlastnost `value`. Hodnota vlastnosti se rovná textu v textovém poli. Například můžeme zjednodušit nastavení textu položky zasláním zprávy `set_value`:

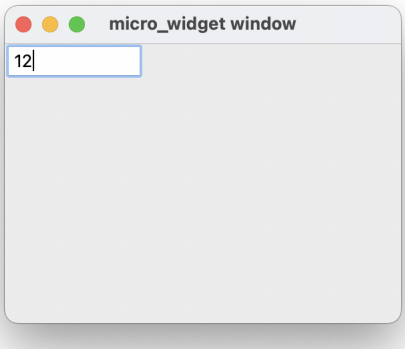
```
el.set_value("Eliška")
```

   Zasláním zprávy `get_value` položce získáme text zadaný uživatelem.

4. Podobně přidejte instancím třídy `EntryField` vlastnost `label_text` určující text popisku. Položku formuláře z příkladu můžeme pak úsporně vytvořit takto:

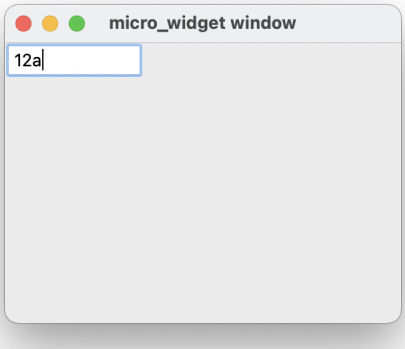
```
ef = EntryField().set_label_text("Jméno:")
ef.set_value("Eliška")
```

5. Napište třídu `IntegerEntry` pro celočíselné pole s vlastností `value`, jejíž hodnota je číslo zobrazené v poli. Například číselné pole:



má hodnotu vlastnosti `value` rovnou dvanáct.

6. Získání hodnoty vlastnosti `value` číselného pole musí vyvolat výjimku v případě, že zobrazený text není celé číslo. Například zaslání zprávy `get_value` poli:



musí vyvolat výjimku.

7. Přidejte číselnému poli obsluhu zprávy `is_valid` zasílané bez argumentů, která rozhodne, zda je text v poli zápisem celého čísla. Napište si k tomu pomocnou proceduru `is_decimal`:

```
>>> is_decimal("12")
True
>>> is_decimal("12a")
False
```

8. Napište třídu `IntegerEntryField`. Přímé instance třídy jsou podobné instancím `EntryField` až na to, že pole slouží k zadání celých čísel.

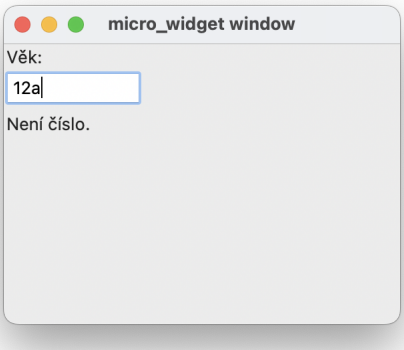
9. Napište pro třídu `IntegerEntryField` metodu `validate`, která zobrazí pod polem validační hlášku v případě, že zadaný text není celé číslo. Například kód:

   ```
   ief = IntegerEntryField()
   ief.set_label_text("Věk:")
   w = Window().set_widget(ief)
   ```

   vytvoří pole, do kterého zadáme `12a`. Zaslání zprávy:

   ```
   ief.validate()
   ```

   podá validační hlášení:



   Samozřejmě po smazání písmena `a` a opětovném zaslání zprávy `validate` chybová hláška zmizí.

10. Vytvořte textové pole, do kterého půjdou zadat jen cifry.

11. Vytvořte počítadlo, které bude tvořit tlačítko zobrazující jeho hodnotu. Stisk tlačítka hodnotu počítadla inkrementuje.