Ознаки – код для повторного використання

Хоча Pharo пропонує тільки просте наслідування класів, воно підтримує механізм, який називається *Traits, Ознаки*[[1]](#footnote-1), для спільного використання коду, який описує поведінку і стан, різними непов’язаними між собою класами. Ознака містить набір методів, які можуть працювати в різних класах без обмежень щодо відношення наслідування.

За допомогою ознак можна поширювати код серед різних класів без дублювання. Це полегшує для класів повторне використання певної поведінки, інкапсульованої в одному місці.

Як побачимо, ознаки пропонують спосіб компонування та вирішення конфліктів імен строго визначеним способом. У використанні ознак не обов’язково перемагає останній завантажений метод, як це відбувається в інших мовах. У Pharo чи клас, чи ознака завжди беруть до уваги пріоритетність і вирішують у власному контексті, як усунути конфлікт: методи можна вилучити з компонування, або зробити їх доступними під новим іменем.

## Проста ознака

Наведений нижче код визначає ознаку за допомогою надсилання повідомлення *named:uses:package:* класові *Trait*. Частина *uses:* повідомлення містить порожній масив, що засвідчує те, що ця ознака не містить інших ознак.

Trait named: #TFlyingAbility

uses: {}

package: 'Traits-Example'

Ознаки можуть визначати методи. Ознака *TFlyingAbility* визначає один метод *fly*.

TFlyingAbility >> fly

^ 'I''m flying!'

Ознака не призначена для створення екземплярів – її має використати клас. Цей клас створить екземпляри, які зможуть відповідати на реалізовані в ній повідомлення.

Тепер визначимо клас *Bird*. Він використовує ознаку *TFlyingAbility*. Завдяки цьому клас містить метод *fly*.

Object subclass: #Bird

uses: TFlyingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Екземпляри класу *Bird* уміють відповідати на повідомлення *fly*.

Bird new fly

>>> 'I''m flying!'

## Виклик необхідного методу

Методи не ознаки не мусять визначати повністю всю поведінку. Метод ознаки може викликати методи, які стануть доступними тільки в класі, який використає ознаку.

У прикладі метод *greeting* ознаки *TGreetable* викликає метод *name*, не визначений в ознаці. В такому випадку клас, який використає ознаку, має реалізувати такий *необхідний* метод.

Trait named: #TGreetable

uses: {}

package: 'Traits-Example'

TGreetable >> greeting

^ 'Hello ', self name

Зверніть увагу, що *self* в методі ознаки представляє отримувача повідомлення. Тут діють такі самі правила, як у методах класу.

Object subclass: #Person

uses: TGreetable

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Визначено клас *Person*, який використовує ознаку *TGreetable*. Потрібно визначити метод *name* в класі *Person*. Його викликатиме метод *TGreetable >> greeting*.

Person >> name

^ 'Bob'

Person new greeting

>>> 'Hello Bob'

## У методі ознаки *self* вказує на отримувача

Можна засумніватися, на що вказує *self* у методі ознаки. Проте немає різниці між вико­ристанням *self* у методі, визначеному в класі, та в методі, визначеному в ознаці: *self* завжди представляє отримувача повідомлення. Місце визначення методу: чи це клас, чи ознака – ніяк не впливає на *self*.

На підтвердження визначено невелику ознаку, її метод *whoAmI* лише повертає *self*.

Trait named: #TInspector

uses: {}

package: 'Traits-Example'

TInspector >> whoAmI

^ self

Object subclass: #Foo

uses: TInspector

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Наведений нижче фрагмент коду демонструє, що *self* – це отримувач, навіть, якщо повертається з методу ознаки.

| foo |

foo := Foo new.

foo whoAmI == foo

>>> true

## Стан ознаки

Починаючи з Pharo 7.0, в ознаках можна оголошувати змінні екземпляра. У новому прикладі ознака *TCounting* визначає змінну екземпляра, що називається *count*.

Trait named: #TCounting

instanceVariableNames: 'count'

package: 'Traits-Example'

Ознака може ініціалізувати свій стан спеціальним методом, чий селектор, за домов­леністю, будують зі слова «*initialize*» та імені ознаки слідом. Далі ознака *TCounting* визначає метод *initializeTCounting* і метод збільшення значення змінної.

TCounting >> initializeTCounting

count := 0

TCounting >> increment

count := count + 1.

^ count

Клас *Counter* використовує ознаку *TCounting*, тому його екземпляри матимуть змінну *count*.

Object subclass: #Counter

uses: TCounting

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'.

Щоб правильно ініціалізувати екземпляр класу *Counter*, метод *Counter >> initialize* мав би викликати визначений в ознаці метод *initializeTCounting*.

Counter >> initialize

self initializeTCounting

У наступному фрагменті створено екземпляр класу *Counter*. Видно, що його змінну екземпляру ініціалізовано правильно.

Counter new increment; increment

>>> 2

## Клас може використати кілька ознак

Клас не обмежено використанням лише однієї ознаки. Він може використати їх кілька. Раніше було оголошено ознаку *TFlyingAbility*, а тепер оголосимо ще одну – *TSpeaking­Ability*.

Trait named: #TSpeakingAbility

uses: {}

package: 'Traits-Example'

Вона визначає метод *speak*.

TSpeakingAbility >> speak

^ 'I''m speaking!'

Тепер клас *Duck* може використати обидві ознаки: і *TFlyingAbility*, і *TSpeakingAbility*.

Object subclass: #Duck

uses: TFlyingAbility + TSpeakingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Екземпляр класу *Duck* отримав поведінку з обох ознак.

| d |

d := Duck new.

d speak

>>> 'I''m speaking!'

d fly

>>> 'I''m flying!'

## Перевизначений метод має вищий пріоритет, ніж метод ознаки

Метод, родом з ознаки, діє так само, ніби його визначили в класі (чи ознаці), що вико­ристовує її. Користувач ознаки (це може бути клас або інша ознака) завжди може перевизначити метод, який походить з неї, і перевизначений метод матиме вищий пріоритет, ніж метод ознаки.

Проілюструємо це. Можна було б перевизначити метод *speak* у класі *Duck* так, щоб він робив щось інше, наприклад, надсилав повідомлення *quack*.

Duck >> quack

^ 'QUACK'

Duck >> speak

^ self quack

Це означає, що:

* метод *TSpeakingAbility >> speak* більше не доступний з класу *Duck*, і
* замість нього використовується новий метод, навіть тими методами, які надсилають повідомлення *speak*.

Duck new speak

>>> 'QUACK'

Додамо до ознаки *TSpeakingAbility* оголошення ще одного методу.

TSpeakingAbility >> doubleSpeak

^ 'I double: ', self speak, ' ', self speak

Легко переконатися, що визначений в класі *Duck* метод *speak* має вищий пріоритет, ніж його однойменний конкурент з ознаки *TSpeakingAbility*.

Duck new doublespeak

>>> 'I double: QUACK QUACK'

## Доступ до перевантаженого методу ознаки

Іноді виникає потреба перевизначити метод ознаки і водночас зберегти доступ до нього. Це можливо за допомогою створення псевдоніма перевизначеного методу в частині оголошення використання ознаки операторами @ і -> як у прикладі.

Object subclass: #Duck

uses: TFlyingAbility + TSpeakingAbility @ {#originalSpeak -> #speak}

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Стрілка означає, що новий метод це те саме, що й старий, тільки оголошено нове ім’я. Тут сказано, що *originalSpeak* нове ім’я методу *speak*.

Щоб доступитися до перевантаженого методу, надсилають повідомлення з його новим іменем так, ніби це ім’я звичайного методу. Далі оголошено новий метод *differentSpeak*, який надсилає повідомлення *originalSpeak*.

Duck >> differentSpeak

^ self originalSpeak, ' ', self speak

Duck new differentSpeak

>>> 'I''m speaking! QUACK'

Будьте уважні, бо оголошення псевдоніма не змінює імені методу. Справді, якщо йдеться про рекурсивний метод, то він не звертатиметься за новим іменем, а за старим. Псевдонім додає нове ім’я наявному методу, але не змінює його визначення: тіло методу залишиться незмінним. Отже, псевдонім оголошують, так би мовити, для зовнішнього використання.

## Опрацювання конфлікту

Може трапитися, що дві ознаки, використані в одному класі, визначають однойменний метод. Така ситуація призводить до конфлікту. Щоб вирішити його, можна використа­ти дві різні стратегії.

1. За допомогою оператора вилучення (–) можна вилучити конфліктуючий метод з однієї ознаки. Тоді в класу залишиться інший.
2. Перевизначити конфліктуючий метод у класі. У цьому випадку клас матиме доступ тільки до нового методу, відповідні методи ознак стануть недоступними, і конфлікт буде вичерпано. Зрозуміло, що доступ до обох методів ознак можна зберегти за допомогою псевдонімів, як було пояснено раніше.

Розглянемо приклад. Визначимо нову ознаку *THighFlyingAbility*.

Trait named: #THighFlyingAbility

instanceVariableNames: ''

package: 'Traits-Example'

Вона також визначає метод *fly*.

THighFlyingAbility >> fly

^ 'I''m flying high'

Якщо тепер оголосити клас *Eagle*, який використовує обидві ознаки, і *THighFlyingAbility*, і *TFlyingAbility*, то отримаємо конфлікт під час опрацювання повідомлення *fly* до екзем­пляра класу, бо система не знатиме, котрий з методів виконати.

Object subclass: #Eagle

uses: THighFlyingAbility + TFlyingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Eagle new fly

>>> 'A class or trait does not properly resolve a conflict between

multiple traits it uses.'

## Вирішення конфлікту – вилучити метод

Щоб вирішити конфлікт під час компонування, можна вилучити метод *fly* з ознаки *TFlyingAbility*:

Object subclass: #Eagle

uses: THighFlyingAbility + (TFlyingAbility - #fly)

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Тепер у класу тільки один метод *fly*, отриманий з *THighFlyingAbility*.

Eagle new fly

>>> 'I''m flying high'

## Вирішення конфлікту – перевизначити метод

Інший спосіб вирішити конфлікт – перевизначити конфліктуючий метод у класі, який використовує ознаки.

Object subclass: #Eagle

uses: THighFlyingAbility + TFlyingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Eagle >> fly

^ 'Flying and flying high'

Тепер доступний тільки один метод *fly* – визначений у класі.

Eagle new fly

>>> 'Flying and flying high'

Щоб доступитися до перевантажених методів *THighFlyingAbility >> fly* і *TFlyingAbility >> fly*, можна оголосити і використати їхні псевдоніми, як описано в параграфі 11.7.

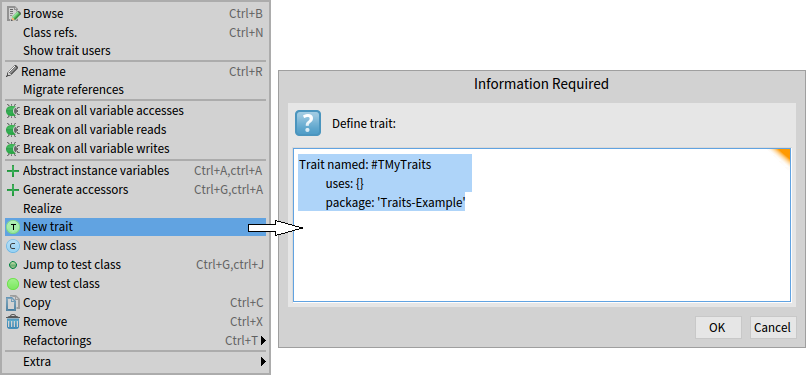


Рис. .. Діалог для створення ознаки

## Ознаки та наслідування

Ознаки визначають методи та змінні екземпляра. Їх треба розглядати як різновид *фрагментів* класу. Тому не можна створити екземпляр ознаки. Ознаку використовують класи, екземпляри яких отримують стан і поведінку ознаки.

Крім того, ознака не призначена для оголошення підкласів. Наслідування ознаки не спрацює. Проте ознаку можуть повторно використати інші ознаки. Кажуть, що її можна скласти з інших ознак.

Отже, ми отримуємо світ, де:

* класи можна розширювати шляхом наслідування. Клас може використовувати ознаки;
* ознаки повторно використовують класи й інші ознаки. Ознаки не наслідують, а тільки компонують.

Якщо два класи, які використовують різні ознаки, що визначають метод з однаковою назвою, як-от *fly* у нашому прикладі, перебувають у відношенні наслідування, пошук методу працює нормально: він працює так, ніби ознаки не існують, а методи визначені безпосередньо в класах.

Уявімо, що *Bird* використовує *TFlyingAbility*, *Eagle* використовує *THighFlyingAbility* та наслідує *Bird*. Екземпляр *Eagle* виконає метод *fly* з *THighFlyingAbility* або локально перевизначений в класі, якщо він там є.

## Про що не було сказано[[2]](#footnote-2)

Окремі важливі моменти ненавмисно випали з пояснень. Поговоримо про них зараз.

### Створення ознаки

У розділі було доволі прикладів. Сподіваємося, ви вже здогадалися, як оголошувати ознаки, щоб випробувати їхнє використання. Ви могли б скористатися одним з таких способів.

**Оглядач класів** можна використовувати для оголошення і нових класів, і нових ознак. За замовчуванням Оглядач пропонує шаблон оголошення класу. Щоб отримати шаблон ознаки, скористайтеся командою «*New trait*» контекстного меню панелі класів Огляда­ча (див. рис. 11.1). Вона відкриває окреме вікно, в якому можна уточнити деталі нової ознаки. Після натискання на кнопку «*ОК*» ознака буде збережена і відкомпільована.

Зверніть увагу на різні піктограми класу та ознаки в контекстному меню. Такі ж Оглядач відображає на панелі класів (імена ознак з’являються тут) і панелі методів.

Якщо ви розпочали роботу зі створення нового пакета класів, то відкрити контекстне меню на порожній панелі класів не вдасться. Тоді можна вручну замінити у вікні редак­тора коду Оглядача шаблон оголошення класу на оголошення ознаки та зберегти його.

**Робоче вікно** також може стати в пригоді. Адже це – Pharo! Тут усе можна виконати програмно. Наберіть у вікні, наприклад, «*Trait named: #TMyTraits uses: {} package: 'Traits-Example'*» і виконайте командою «*Do it*» – ви побачите в Оглядачі, що і пакет створено, й ознаку в ньому.

### Екземпляри ознаки

Повторимо ще раз: ознаки – не класи, створювати екземпляри ознак не можна. Ви можете спробувати виконати, наприклад, «*TGreetable new*». Тоді у відповідь отримаєте вікно з повідомленням про помилку «*Traits should not be instantiated!*» і роз’яснення, що ознаки не наслідують *Object*, тому не розуміють повідомлень до нього.

Ознаки – дуже специфічні об’єкти. Їх варто трактувати як будівельні блоки для конст­руювання класів. Перенесіть подумки оголошення методу з ознаки до класу, який її використовує, і все стане на свої місця: і значення *self*, і можливість викликати методи класу, і перевантаження методів. Нагадаємо також, що метод з ознаки можуть запози­чити кілька користувачів – класів і інших ознак. Зміна визначення такого методу вплине на кожного користувача.

### Приклад наслідування

Пояснимо співвідношення ознак і наслідування на прикладі. Для цього нагадаємо деякі з оголошених раніше ознак і класів.

Trait named: #TFlyingAbility

uses: {}

package: 'Traits-Example'

TFlyingAbility >> fly

^ 'I''m flying!'

Trait named: #THighFlyingAbility

instanceVariableNames: ''

package: 'Traits-Example'

THighFlyingAbility >> fly

^ 'I''m flying high'

Object subclass: #Bird

uses: TFlyingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Потім ми оголошували клас *Duck*, підклас *Object*. Але логічно було б оголосити його підкласом *Bird*! Так і зробимо.

Bird subclass: #Duck

uses: TSpeakingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Як і раніше, качка вміє і літати, і говорити.

| d |

d := Duck new.

d speak

>>> 'I''m speaking!'

d fly

>>> 'I''m flying!'

Для польотів клас *Duck* використовує успадкований метод *fly*, запозичений класом *Bird* з ознаки *TFlyingAbility*. «Будівельний блок» ознаки став частиною *Bird*, тому все працює.

Продовжимо експерименти з наслідуванням і змінимо оголошення класу *Eagle*.

Bird subclass: #Eagle

uses: THighFlyingAbility

instanceVariableNames: ''

classVariableNames: ''

package: 'Traits-Example'

Раніше ми отримали конфлікт однойменних методів *fly*, отриманих класом *Eagle* з ознак *THighFlyingAbility* і *TFlyingAbility*. Тепер цього не відбудеться. Клас *Eagle* отримає метод з *THighFlyingAbility* і так перевантажить метод надкласу. Вибір однозначний – у класі *Eagle* доступний тільки *THighFlyingAbility >> fly*.

### Термінологія

Слово *Trait* використовують у С++, коли йдеться про використання шаблонів і характеристики шаблонів. Нехай зовнішня схожість не вводить вас в оману: ознаки Pharo не мають нічого спільного з характеристиками С++. За призначенням ознаки найближчі до підмішаних класів у Python.

## Підсумок розділу

Ознаки – це групи методів і станів, які можна повторно використовувати різним користувачам (класам або ознакам), підтримуючи різновид множинного наслідування в контексті мови з простим наслідуванням. Клас або ознака може використовувати кілька ознак. Ознаки можуть визначати змінні екземпляра та методи. Коли ознаки, які використовуються в класі, визначають метод з однаковою назвою, це призводить до конфлікту.

Конфлікт можна вирішити двома способами:

* користувач (клас або ознака) може перевизначити конфліктуючий метод: метод класу має вищий пріоритет, ніж метод ознаки;
* користувач може вилучити конфліктуючий метод.

Перевантажені методи можна викликати за допомогою псевдонімів, які визначають у частині оголошення використання ознаки.

Не можна створювати екземпляри ознак.

Ознаки, використані для побудови класу, доступні в його підкласах.

1. Дослівний переклад слова *traits* – типаж, характерні риси – не дуже доречний у контексті можливостей цього механізму, тому в книзі використано близький варіант *ознака*. Traits (у Pharo) – колекція методів, які можна залучати для побудови класу. Уміння виконувати певні методи – ознака класу, тому, на думку перекладача, термін *ознака* підходить найкраще. [↑](#footnote-ref-1)
2. Цей параграф написав перекладач книги. [↑](#footnote-ref-2)