8 Правила (и исключения) наследования

В этой главе:

- ✓ Использование наследования и композиции вместе для моделирования систем.
- ✓ Использование встроенных модулей Python для проверки типов объектов.
- ✓ Придание интерфейсам большей строгости с помощью абстрактных базовых классов.

Если вы сами писали классы или использовали фреймворк на основе классов в Python, то скорее всего столкнулись с *наследованием*. Классы могут наследовать от других классов, перенимая поведение родительского класса. Из этой главы вы больше узнаете о наследовании в Python и о том, когда оно уместно или неуместно.

8.1. НАСЛЕДОВАНИЕ РАНЬШЕ

Наследование было задумано в первые дни компьютерного программирования, но люди по-прежнему ведут оживленные споры о том, когда и как его использовать. Большую часть истории объектно-ориентированного программирования наследование было его сутью. Приложения представляли собой модели реального мира с тщательно подобранной иерархией объектов. Так объектно-ориентированное программирование и наследование стали практически неразделимыми.

8.1.1. Серебряная пуля

Хотя наследование иногда становится наилучшим инструментом для достижения цели, оно использовалось на практике как молоток для каждого гвоздя, как неуловимая серебряная пуля. Но как и серебряная пуля, парадигма, отвечающая сразу всем требованиям, — это художественный вымысел.

Разочарование от наследования привело многих разработчиков к отказу от объектно-ориентированного программирования. Это печально, ведь объектная ориентация имеет ряд преимуществ для мысленного моделирования задач, и наследование имеет свое место во время создания правильных иерархий. Мы применим наследование к конкретному набору вариантов использования, о которых вы узнаете из этой главы.

Но сначала несколько слов о том, почему наследование классов привело к такому большому разочарованию.

8.1.2. Трудности иерархий

Объектно-ориентированное программирование всецело касается разделения, инкапсуляции и классификации информации и форм поведения. Я работаю со многими программистами, отвечающими за библиотеки, которые забыли о классификации больше, чем я когда-либо о ней узнаю, — эти люди работают, чтобы выявлять связи между явлениями,

создавая таксономии или даже онтологии[1]. Эти подходы хорошо работают для организации необработанной информации, но вызывают головную боль, как только дело касается поведения ПО. По мере роста ПО становится все труднее поддерживать четкое понимание отношений «родитель — ребенок» между классами.

> **ПРИМЕЧАНИЕ**
>
> Родительские классы в Python (и во многих других языках) называются *суперклассами*. *Дочерние* классы называются *подклассами*. Я буду использовать эту терминологию на протяжении всей остальной части главы.

Класс наследует всю информацию и поведение от своего суперкласса, а затем может переопределить их, чтобы выполнить что-то новое (рис. 8.1). Это, пожалуй, самая тесная сопряженность, которая существует в программировании. Класс является полностью сопряженным со своим суперклассом, потому что все, что он знает и делает по умолчанию, привязано к суперклассу.

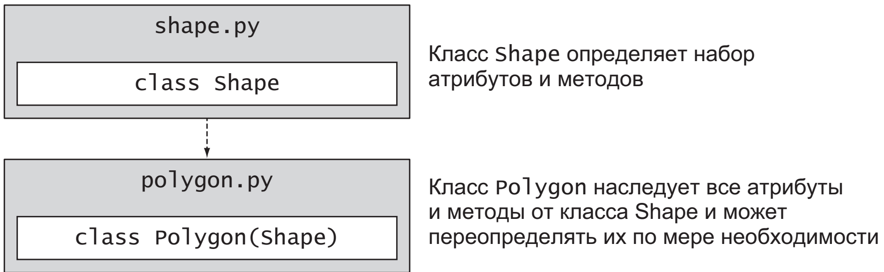

Рис. 8.1. Наследование с одним суперклассом и одним подклассом

Увидеть эту сопряженность *очень трудно*, когда иерархии классов растут, потому что, глядя на конкретный класс, вы не увидите, наследует ли

[1] Подробнее об онтологии в контексте информатики читайте в статье https://ru.wikipedia.org/wiki/Онтология_(информатика).

другой класс от него или нет. Это приводит к ошибкам из-за неожиданных изменений в поведении (рис. 8.2).

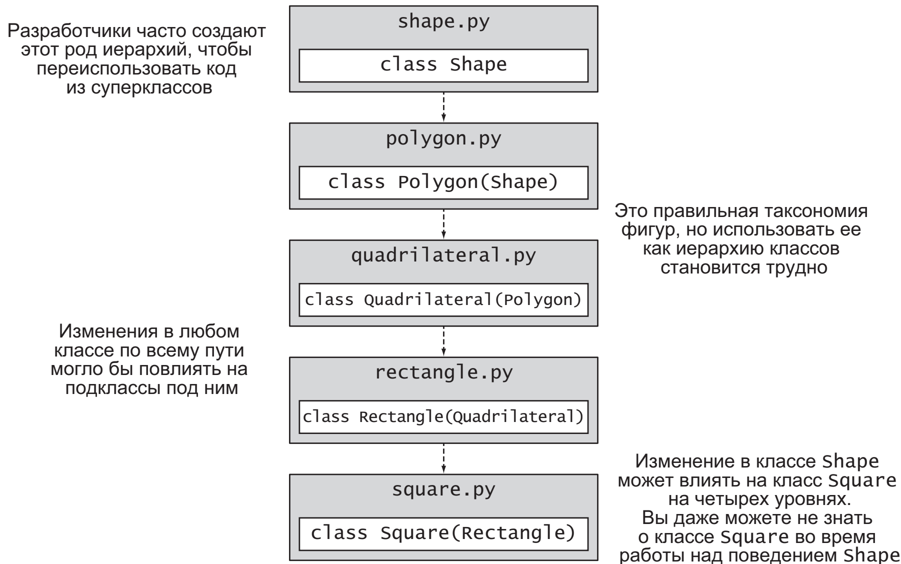

Рис. 8.2. Как глубокая иерархия наследования приводит к увеличению числа ошибок

Если провести аналогию, то в квантовой физике две частицы могут быть *запутаны* таким образом, что изменение одной из них приведет к такому же изменению в другой, независимо от того, как далеко они друг от друга находятся в пространстве. Это «жуткое действие на расстоянии», как называл его Эйнштейн, означает, что вы не можете достоверно определить состояние частицы, потому что оно может измениться в любой момент из-за ее близнеца. В ПО это явление представляет серьезную опасность. Изменяя один класс, вы можете непреднамеренно изменить — или, что еще хуже, — нарушить функциональность в другом подклассе, о котором вы не знали. Как в фильме «Эффект бабочки». (Внимание, спойлер: эта проблема мешает персонажу Эштона Кутчера.)

Иногда наследование применяется для переиспользования кода, но оно создает проблемы в дальнейшем. При наличии глубокой иерархии классы на разных уровнях могут переопределять или дополнять поведение своих

суперклассов. Очень скоро вы обнаружите, что ходите вверх и вниз по классам, пытаясь проследить поток информации. Я уже говорил, что труд разработчика направлен на улучшение понимания кода и уменьшение его когнитивной нагрузки, но глубокие иерархии работают против этой цели. Тогда почему мы все еще используем наследование?

8.2. НАСЛЕДОВАНИЕ СЕЙЧАС

Из-за головной боли, вызываемой сложными иерархиями, наследование приобрело плохую репутацию. Но это зло не было врожденным. Просто наследование использовалось слишком часто и не по назначению.

8.2.1. Зачем нужно наследование?

Многие по-прежнему через наследование хотят переиспользовать код в каком-то классе, но в реальности оно предназначено для *специализации поведения*. Другими словами, боритесь с желанием создать подкласс только ради переиспользования кода. Создавайте подклассы для того, чтобы метод возвращал другое значение либо под капотом работал по-другому.

Рассматривайте подклассы как *специальные случаи* суперкласса. Они будут многократно использовать код из суперкласса, но не повторять его действия.

Когда класс *B* наследует от класса *A*, мы часто говорим *B* «является» *A*. Это подчеркивает, что экземпляры *B* на самом деле являются экземплярами *A* и выглядят так же (подробнее об этом чуть позже). Сравните это с композицией, где, если экземпляр класса *C* использует экземпляр класса *D*, мы говорим, что *C* «имеет» *D*, чтобы подчеркнуть, что *C* состоит из *D* (среди прочего, в потенциальном плане).

Вспомните пример с `Bicycle` из предыдущей главы. Вы ввели несколько типов велосипедных рам, обновив алюминиевую раму `AluminumFrame` до карбоновой рамы `CarbonFiberFrame`, а шину `Tire` — до причудливой шины `FancyTire`. Предположим, что `CarbonFiberFrame` и `FancyTire` наследуют соответственно от `Frame` и `Tire`. Что из нижеследующего можно сказать о том, как вы моделировали велосипеды, используя наследование и композицию?

1. `Tire` имеет `Bicycle`.
2. `Bicycle` имеет `Tire`.
3. `CarbonFiberFrame` является `Tire`.
4. `CarbonFiberFrame` имеет `Frame`.

Поскольку шина не состоит из велосипеда (все наоборот), вариант 1 неверен, тогда как вариант 2 имеет смысл — это композиция. А карбоновая рама *является* рамой (у нее *нет* рамы), и вариант 4 тоже неверен, тогда как вариант 3 имеет смысл — это наследование. Опять же, наследование предназначено для специализации, тогда как композиция — для многократного использования одной формы поведения (рис. 8.3).

Использование наследования для специализации поведения — это только первый шаг. Представьте, что вы заменяете алюминиевую раму на карбоновую. Каждая рама имеет одинаковые точки стыковки, без которых велосипед развалится. Так же и в ПО.

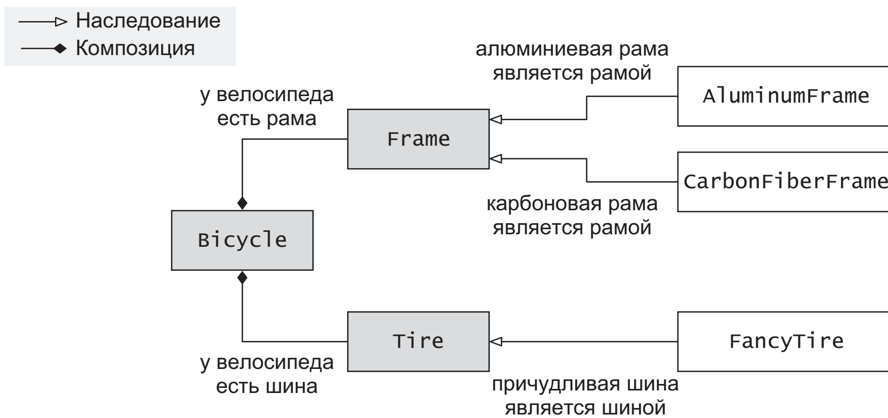

Рис. 8.3. Как наследование и композиция работают вместе

8.2.2. Подстановка

Барбара Лисков, профессор Массачусетского технологического института (MIT), разработала принцип, описывающий концепцию *подстановки* (замещения) в наследовании. Этот принцип гласит, что в программе любой

экземпляр класса должен быть пригодным для замещения экземпляром одного из его подклассов без ущерба для правильности программы.[1] *Правильность* в этом контексте означает, что программа остается безошибочной и достигает тех же базовых результатов, хотя точный результат может быть другим или достигнут другим способом. Подстановка возникает из-за того, что подклассы строго придерживаются интерфейса своих суперклассов.

В Python нетрудно отойти от этого принципа. Рассмотрим следующий листинг, где совершенно правильный код Python моделирует слизней и улиток (две разновидности брюхоногих моллюсков). Класс улиток `Snail` наследует от класса слизняков `Slug` (улитки и слизняки одинаковы во всем, кроме раковины), и можно сказать, что класс `Snail` специализирует, или детализирует, класс `Slug`, добавляя информацию о раковине. Но класс `Snail` нарушает подстановку, потому что программа, использующая класс `Slug`, **не может заменить его классом** `Snail`, не добавив аргумент `shell_size` в метод `__init__`.

Листинг 8.1. Подкласс, нарушающий принцип подстановки

```
class Slug:
    def __init__(self, name):
        self.name = name

    def crawl(self):
        print('slime trail!')


class Snail(Slug):        ◄── Класс Snail наследует от класса Slug
    def __init__(self, name, shell_size):  ◄── Использование сигнатуры создания
    super().__init__(name)                     отличающегося экземпляра часто
    self.name = name                           ведет к нарушению подстановки
    self.shell_size = shell_size


def race(gastropod_one, gastropod_two):   Вы можете создать два экземпляра клас-
    gastropod_one.crawl()                  са Slug и организовать гонки
    gastropod_two.crawl()

                                           Попытка использовать класс Snail
race(Slug('Джеффри'), Slug('Рамона')) ◄── без аргумента shell_size возбуждает
race(Snail('Джеффри'), Snail('Рамона'))  ◄── исключение
```

[1] https: //ru.wikipedia.org/wiki/Принцип_подстановки_Барбары_Лисков.

Можно вытащить из рукава еще больше трюков, чтобы заставить все работать, но учтите, что все это больше подходит для композиции. В конце концов, у улитки *есть* раковина.

Мне нравится сомневаться в *роли*, которую выполняет конкретный набор классов. Если каждый класс в иерархии может выполнять одну и ту же роль, то они пригодны для замещения. Если подкласс изменяет любую свою сигнатуру метода или вызывает исключение в рамках своей специализации, то он может и не выполнять эту роль, но иерархию классов тогда придется переорганизовать.

8.2.3. Идеальный для наследования вариант использования

Сэнди Метц, программист Ruby, который первоначально пришел из сообщества Smalltalk (Smalltalk — язык программирования, частично созданный Аланом Кеем, одним из пионеров объектно-ориентированного программирования), изложил большой набор базовых правил о том, когда следует использовать наследование:[1]

- задача, которую вы решаете, имеет мелкую узкую иерархию;
- подклассы находятся в листьях графа объектов и пользуются другими объектами;
- подклассы используют (специализируют, детализируют) *все* поведение суперкласса.

Я расскажу о каждом из них подробнее.

Мелкая узкая иерархия

Мелкая часть этого правила гласит, что глубоко вложенные иерархии классов приводят к трудностям в управлении ими и вводят дефекты. Поэтому, поддерживая иерархию малой и ограниченной, вам становится понятно, зачем это нужно (рис. 8.4).

---

[1] Подробнее об этом в выступлении Сэнди Метца *All the Little Things* («Все эти мелочи») на RailsConf, 2014 г., www.youtube.com/watch?v=8bZh5LMaSmE.