Урок Классы № 2

Проектирование и разработка классов. Часть 2

1. **Ходы на доске с другими фигурами и взятие фигур. Варианты проектирования**
2. **Реализация взятия фигур. Проверка того, что фигура не проходит сквозь другие**
3. **Создание и использование своих модулей**

Аннотация

На этом уроке мы продолжим проектирование и реализацию шахмат, которые не закончили в прошлый раз. Попробуем реализовать движение фигур по доске, на которой есть другие фигуры. Например, ладья, как и другие фигуры (за исключением коня), не может двигаться сквозь другие фигуры.

1. Ходы на доске с другими фигурами и взятие фигур. Варианты проектирования

Естественнее всего реализовать проверку возможности хода в методе `can_move`, однако так не получится, поскольку в данный момент фигура знает только свои текущие координаты и координаты назначения. Информации о других фигурах на доске у нее нет! При проектировании мы приняли неудачное решение о том, как хранить информацию о положении фигуры на доске — в результате придется дорабатывать не только код метода `can_move`, но и тот метод класса `Board`, который вызывает `can_move`.

Возможно несколько проектных решений по передаче информации о положении фигуры на доске. Давайте их рассмотрим.

1. Хранение информации о координатах фигуры и в экземпляре доски, и в экземпляре фигуры (сейчас мы выбрали этот вариант).

В экземпляре `Board` информация хранится в силу того, что для получения ссылки на фигуру `self.field[row][col]` нужно знать ее индексы `row` и `col`. В самой фигуре координаты хранятся как свойства. Их нужно передавать при инициализации и внимательно следить, чтобы при любом передвижении фигуры в `self.field` вызывался метод `set_position` для обновления координатных свойств внутри фигуры.

Достоинства: всегда просто узнать координаты фигуры.

Недостатки:

– Нужно следить за актуальностью координатных свойств фигур

– У фигуры нет доступа ко всей доске

2. Хранить ссылку на доску внутри фигуры.

Достоинства:

– У фигуры есть информация о других фигурах

– Не нужно поддерживать актуальность двух представлений, координаты хранятся только в экземпляре `Board`

Недостатки:

– Чтобы фигура могла узнать свои координаты, ей придется искать себя на доске в циклах по `row` и `col`

– Кольцевая ссылка (фигура ссылается на доску, доска на фигуру) в старых версиях Python (до 3.6) мешает работе сборщика мусора. Если не сделать специальный метод для разрыва кольцевых ссылок перед завершением работы с доской или забыть его вызвать, произойдет утечка памяти. Не во всех языках программирования сборщики мусора умеют корректно работать с кольцевыми ссылками

3. Передавать координаты фигуры в вызовы тех методов, которым они нужны.

Достоинства:

– Не нужно поддерживать соответствие двух наборов координат

Недостатки:

– Сложнее вызывать методы

– Нет доступа ко всей доске

4. Передавать объект доски в вызовы методов фигур.

Достоинства:

– Не нужно поддерживать соответствие двух наборов координат

Недостатки:

– Для получения координат нужно искать фигуру на доске

5. Передавать и координаты фигуры, и объект доски.

Достоинства:

– Не нужно поддерживать соответствие двух наборов координат

– Не нужно искать на доске фигуру — координаты уже есть

– Никаких кольцевых ссылок

Недостатки:

– Более сложный вызов метода

Вообще говоря, вариантов еще больше. Например, вместо двумерного поля, в ячейки которого

записываются фигуры, можно использовать список фигур, каждая из которых хранит свои координаты. Для краткости мы не рассматриваем такое представление.

Для удобства все рассмотренные варианты представлены в таблице.

Остановимся на передаче в метод `can_move` и текущих координат фигуры, и экземпляра доски (вариант 5).

2. Реализация взятия фигур. Проверка того, что фигура не проходит сквозь другие

Код функций `main` и `print_board` оставляем без изменений. В классе `Board` изменим `__init__` так, чтобы она расставляла на доске полный набор фигур. Интересно, что классы фигур теперь спроектированы так, что фигура не хранит никаких свойств, кроме цвета. Таким образом, например, восемь ссылок на одну белую пешку могут работать не хуже, чем восемь различных белых пешек. Впрочем, такая особенность в нашем случае не принесет большой экономии памяти. К тому же она помешает реализации некоторых возможностей (рокировке, взятию на проходе), поэтому все ссылки будут указывать на различные объекты.

Метод `move_piece` класса `Board` теперь будет смотреть, что расположено в поле назначения. Если поле пусто, он попытается сделать ход на эту клетку; если оно занято фигурой противника — попытаться съесть эту фигуру.

```
class Board:
    def __init__(self):
        self.color = WHITE
        self.field = []
        for row in range(8):
            self.field.append([None] * 8)
        self.field[0] = [
            Rook(WHITE), Knight(WHITE), Bishop(WHITE), Queen(WHITE),
            King(WHITE), Bishop(WHITE), Knight(WHITE), Rook(WHITE)
        ]
        self.field[1] = [
            Pawn(WHITE), Pawn(WHITE), Pawn(WHITE), Pawn(WHITE),
            Pawn(WHITE), Pawn(WHITE), Pawn(WHITE), Pawn(WHITE)
        ]
        self.field[6] = [
            Pawn(BLACK), Pawn(BLACK), Pawn(BLACK), Pawn(BLACK),
            Pawn(BLACK), Pawn(BLACK), Pawn(BLACK), Pawn(BLACK)
        ]
        self.field[7] = [
            Rook(BLACK), Knight(BLACK), Bishop(BLACK), Queen(BLACK),
            King(BLACK), Bishop(BLACK), Knight(BLACK), Rook(BLACK)
        ]

    # ... методы current_player_color и cell без изменений ...

    def move_piece(self, row, col, row1, col1):
```

```
        '''Переместить фигуру из точки (row, col) в точку (row1, col1).
        Если перемещение возможно, метод выполнит его и вернёт True.
        Если нет --- вернёт False'''

        if not correct_coords(row, col) or not correct_coords(row1, col1):
            return False
        if row == row1 and col == col1:
            return False  # нельзя пойти в ту же клетку
        piece = self.field[row][col]
        if piece is None:
            return False
        if piece.get_color() != self.color:
            return False
        if self.field[row1][col1] is None:
            if not piece.can_move(self, row, col, row1, col1):
                return False
        elif self.field[row1][col1].get_color() == opponent(piece.get_color()):
            if not piece.can_attack(self, row, col, row1, col1):
                return False
        else:
            return False
        self.field[row][col] = None  # Снять фигуру.
        self.field[row1][col1] = piece  # Поставить на новое место.
        self.color = opponent(self.color)
        return True
```

Теперь можно легко запрограммировать проверку того, что при движении пешка и ладья не могут проходить сквозь другие фигуры. Проверку на то, что в конечной клетке нет фигуры, делает сама доска, поэтому достаточно реализовать проверку промежуточных клеток. В классе ладьи для перебора клеток мы воспользуемся циклом следующего вида:

```
step = 1 if (b >= a) else -1
for i in range(a + step, b, step):
    do_something()
```

Этот цикл проходит от `a` (не включая) до `b` (не включая) как при `a > b`, так и при `a < b`. При `a == b` тело цикла не выполнится ни разу.

Нужно также реализовать отдельный метод `can_attack` для проверки возможности съесть фигуру в клетке назначения. Так как у пешки траектория нападения отличается от траектории движения, приходится делать два отдельных метода. Для остальных фигур метод `can_attack` будет просто возвращать результат работы `can_move`.

Сам код классов выглядит так, как показано ниже.

Класс «Ладья»

```
class Rook:
    def __init__(self, color):
        self.color = color
```

```
    def get_color(self):
        return self.color


    def char(self):
        return 'R'

    def can_move(self, board, row, col, row1, col1):
        # Невозможно сделать ход в клетку,
        # которая не лежит в том же ряду или столбце клеток.
        if row != row1 and col != col1:
            return False

        step = 1 if (row1 >= row) else -1
        for r in range(row + step, row1, step):
            # Если на пути по вертикали есть фигура
            if not (board.get_piece(r, col) is None):
                return False

        step = 1 if (col1 >= col) else -1
        for c in range(col + step, col1, step):
            # Если на пути по горизонтали есть фигура
            if not (board.get_piece(row, c) is None):
                return False
        return True

    def can_attack(self, board, row, col, row1, col1):
        return self.can_move(board, row, col, row1, col1)
```

Как видно, нам еще нужно добавить в класс `Board` метод `get_piece`, чтобы получать фигуру по координатам, не нарушая инкапсуляции. Впрочем, классы фигур и доски связаны настолько сильно, что не предполагают использования друг без друга. В таких случаях инкапсуляцией иногда пренебрегают, но только между сильно связанными классами, при взаимодействии с остальными она должна присутствовать. Поскольку мы строго следовали сокрытию внутренних данных объектов с самого начала, поступим так же и здесь.

Класс «Пешка»

```
class Pawn:
    def __init__(self, color):
        self.color = color

    def get_color(self):
        return self.color

    def char(self):
        return 'P'

    def can_move(self, board, row, col, row1, col1):
        # Пешка может ходить только по вертикали
```

```
        # "взятие на проходе" не реализовано
        if col != col1:

            return False

        # Пешка может сделать из начального положения ход на 2 клетки
        # вперёд, поэтому поместим индекс начального ряда в start_row.
        if self.color == WHITE:
            direction = 1
            start_row = 1
        else:
            direction = -1
            start_row = 6

        # ход на 1 клетку
        if row + direction == row1:
            return True

        # ход на 2 клетки из начального положения
        if (row == start_row
                and row + 2 * direction == row1
                and board.field[row + direction][col] is None):
            return True
        return False

    def can_attack(self, board, row, col, row1, col1):
        direction = 1 if (self.color == WHITE) else -1
        return (row + direction == row1
                and (col + 1 == col1 or col - 1 == col1))
```

Подведем итоги: мы реализовали основу для шахмат. Используя написанный каркас и следуя принятым решениям об интерфейсах классов, можно реализовать классы остальных фигур. Для полноценной программы для двух игроков нужно еще реализовать рокировку, взятие на проходе и превращение пешки и запретить ходы, не уводящие короля из-под шаха или ставящие его под шах. К сожалению, объем и время урока ограничены. Возможно, вам будет интересно когда-нибудь сделать это самостоятельно.

В выложенном коде программы chess_v2.py определены все классы фигур, но классы коня, слона, ферзя и короля определены как «заглушки». Метод `can_move` в них просто всегда возвращает истину и позволяет фигуре ходить в любую клетку доски. Такие заглушки, которые частично реализуют функциональность, часто применяются при проектировании сверху вниз. Это позволяет писать и отлаживать программы постепенно, не реализуя сразу огромных объемов кода. Главное — не забыть заменить все заглушки на рабочий код в финальной версии программы. Спроектируйте свои классы с максимальным применением механизма наследования от базового класс `Figure`.

3. Создание и использование своих модулей

Мы уже неоднократно импортировали дополнительные модули из стандартной библиотеки в свою программу. Но что, если нам хочется разделить исходный код своей программы на несколько

частей? Например, потому что кода стало слишком много и в нем тяжело ориентироваться либо захотелось использовать уже готовую функцию или класс в другой своей программе.

Тут все работает точно так же, как и с импортом любых других модулей. Давайте рассмотрим самую простую ситуацию: у нас есть есть файл с именем my_module с функция `print_hello`, доступ к которой мы хотим получить из другого нашего файла с кодом с именем main.py, и оба этих файла лежат в одной папке на компьютере. В этом случае код наших файлов будет выглядеть вот так:

```
#  my_module.py


def print_hello():
    print("hello!")
```

```
#  main.py


from my_module import print_hello

print_hello()
```

main.py может выглядеть и вот так:

```
#  main.py


import my_module

my_module.print_hello()
```

Не пугайтесь, если среда разработки будет подчеркивать строчки с импортом ваших модулей, это происходит потому, что мы их не оформили как пакеты, но все будет работать.

Давайте немного усложним ситуацию. Пусть первоначально исходный код my_module выглядит так:

```
def print_hello():
    print("hello!")


print("Проверяем функцию print_hello")
print_hello()
```

Тогда, если мы захотим импортировать модуль целиком и вызвать функцию, у нас возникнут некоторые проблемы:

```
import my_module

my_module.print_hello()
```

```
Проверяем функцию print_hello
hello!
hello!
```

Это происходит из-за того, что при таком импорте (а такую запись используют очень часто) Python импортирует не только все функции и классы модуля, но и выполняет весь код, который находится на верхнем уровне.

Библиотек, которые не содержат кода, не предназначенного для использования внешними потребителями, почти не бывает. Это может быть код, например для проверки каких-то функций модуля, а бывает и такое, что модуль может использоваться не только в составе других программ, но и сам по себе представляет полноценное приложение.

Так как же отделить код для внешних потребителей от того, который нужен только непосредственно для самого модуля? Для этого в Python есть устоявшийся прием: весь код верхнего уровня помещается в функцию `main`, а на верхнем уровне добавляется одно условие. Давайте посмотрим на примере нашего модуля my_module (добавим в функцию `print_hello` вывод служебной переменной для демонстрации):

```
def print_hello():
    print(__name__)
    print("hello!")


def main():
    print("Проверяем функцию print_hello")
    print_hello()


if __name__ == '__main__':
    main()
```

```
Проверяем функцию print_hello
__main__
hello!
```

При запуске на исполнение модуля my_module мы увидим, что переменная `__name__` равна «`__main__`». Это верно для любого модуля, который мы запустили непосредственно, а не импортировали в другой модуль. Посмотрим, что произойдет при импорте, запустим модуль main без изменений:

```
my_module
hello!
```

Когда мы импортируем модуль в другой, `__name__` импортируемого модуля будет равно названию этого модуля, а значит, условие, запускающее функцию `main`, не выполнится.

Как мы уже упоминали, такое оформление своих программ является хорошим тоном, и мы настоятельно рекомендуем вам делать именно так.