* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Классы 2](https://lyceum.yandex.ru/courses/123/groups/568/lessons/635)

**Проектирование и разработка классов. Часть №2**

**План урока**

1

[Ходы на доске с другими фигурами и взятие фигур. Варианты проектирования](https://lyceum.yandex.ru/courses/123/groups/568/lessons/635/materials/1732#1)

2

[Реализация взятия фигур. Проверка того, что фигура не проходит «сквозь» другие](https://lyceum.yandex.ru/courses/123/groups/568/lessons/635/materials/1732#2)

**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 неё нет! При проектировании мы приняли неудачное решение о том, как хранить информацию о положении фигуры на доске — в результате придётся 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 её 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 придётся искать себя на доске в циклах по row и col.
* Кольцевая ссылка (фигура ссылается на доску, доска на фигуру) мешает работе сборщика мусора. Если не сделать специальный метод для разрыва кольцевых ссылок перед завершением работы с доской или забыть его вызвать, произойдёт утечка памяти. В некоторых языках сборщики мусора умеют корректно работать с кольцевыми ссылками, но сборщик Python — нет.

**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 ещё 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](https://yastatic.net/s3/lyceum/content/image/classes_table.png).

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 принесёт большой экономии памяти. К тому же она помешает реализации некоторых возможностей (рокировка, взятие 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. Также нужно 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:

*# 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, ещё нам 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.

*# 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))

Подведём 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 ещё реализовать рокировку, взятие на проходе и превращение пешки, а также запретить ходы, не уводящие короля из-под шаха или ставящие его под шах. К сожалению, объём и время урока ограничены. Возможно, вам будет интересно когда-нибудь сделать это самостоятельно. В выложенном коде программы [chess\_v2.py](https://yastatic.net/s3/lyceum/content/resources/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 объёмов 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.

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты