План выполнения задания

Решать задачу будем последовательной реализацией нужных классов. Во многих случаях правильное разбиение кода на классы делает ваш код хорошо читаемым и экономит очень много времени.
Можно выделить две группы классов:

1. Внутренняя логика игры - корабли, игровая доска и вся логика связанная с ней.
2. Внешняя логика игры - пользовательский интерфейс, искусственный интеллект, игровой контроллер, который считает побитые корабли.

В начале имеет смысл написать классы исключений, которые будет использовать наша программа. Например, когда игрок пытается выстрелить в клетку за пределами поля, во внутренней логике должно выбрасываться соответствующее исключение `BoardOutException`, а потом отлавливаться во внейшней логике, выводя сообщение об этой ошибке пользователю.

Далее нужно релизовать **класс `Dot`** - класс точек на поле. Каждая точка описывается параметрами:

1. Координата по оси `x`.
2. Координата по оси `y`.

В программе мы будем часто обмениваться информацией о точках на поле, поэтому имеет смысле сделать отдельный тип данных дня них. Очень удобно будет реализовать в этом классе метод `__eq__`, чтобы точки можно было проверять на равенство. Тогда, чтобы проверить, находится ли точка в списке, достаточно просто использовать оператор `in`, как мы делали это с числами .

Следущим идёт **класс `Ship`** - корабль на игровом поле, который описывается параметрами:

1. Длина.
2. Точка, где размещён нос корабля.
3. Направление корабля (вертикальное/горизонтальное)
4. Количеством жизней (сколько точек корабля еще не подбито).

И имеет методы:

1. Метод `dots`, который возвращает список всех точек корабля.

Самый важный класс во внутренней логике - **класс `Board`** - игровая доска. Доска описывается параметрами:

1. Двумерный списк, в котором хранятся состояния каждой из клеток.
2. Список кораблей доски.
3. Параметр `hid` типа `bool` - информация о том, нужно ли скрывать корабли на доске (для вывода доски врага), или нет (для своей доски).
4. Количество живых кораблей на доске.

И имеет методы:

1. Метод `add_ship`, который ставит корабль на доску (если ставить не получается, выбрасываем исключения).
2. Метод `contour`, который обводит корабль по контуру. Он будет полезен и в ходе самой игры, и в при расстановке кораблей (помечает соседние точки, где корабля по правилам быть не может).
3. Метод, который выводит доску в консоль в зависимости от параметра `hid`.
4. Метод `out`, который для точки (объекта класса `Dot`) возвращает `True`, если точка выходит за пределы поля, и `False`, если не выходит.
5. Метод `shot`, который делает выстрел по доске (если есть попытка выстрелить за пределы и в использованную точку, нужно выбрасывать исключения).

Теперь нужно заняться внешней логикой:
Класс `Player` - класс игрока в игру (и AI, и пользователь). Этот класс будет родителем для классов с AI и с пользователем. Игрок описывается параметрами:

1. Собственная доска (объект класса `Board`)
2. Доска врага.

И имеет следующие методы:

1. `ask` - метод, который “спрашивает” игрока, в какую клетку он делает выстрел. Пока мы делаем общий для AI и пользователя класс, этот метод мы описать не можем. Оставим этот метод пустым. Тем самым обозначим, что потомки должны реализовать этот метод.
2. `move` - метод, который делает ход в игре. Тут мы вызываем метод `ask`, делаем выстрел по вражеской доске (метод `Board.shot`), отлавливаем исключения, и если они есть, пытаемся повторить ход. Метод должен

возвращать `True`, если этому игроку нужен повторный ход (например если он выстрелом подбил корабль).

Теперь нам остаётся унаследовать **классы `AI` и `User` от `Player`** и переопределить в них метод `ask`. Для `AI` это будет выбор случайной точка, а для `User` этот метод будет спрашивать кординаты точки из консоли.

После создаём наш главный класс - **класс `Game`**. Игра описывается параметрами:

1. Игрок-пользователь, объект класса `User`.
2. Доска пользователя.
3. Игрок-компьютер, объект класса `Ai`.
4. Доска компьютера.

И имеет методы:

1. `random_board` - метод генерирует случайную доску. Для этого мы просто пытаемся в случайные клетки изначально пустой доски расставлять корабли (в бесконечном цикле пытаемся поставить корабль в случайную току, пока наша попытка не окажется успешной). Лучше расставлять сначала длинные корабли, а потом короткие. Если было сделано много (несколько тысяч) попыток установить корабль, но это не получилось, значит доска неудачная и на неё корабль уже не добавить. В таком случае нужно начать генерировать новую доску.
2. `greet` - метод, который в консоли приветствует пользователя и рассказывает о формате ввода.
3. `loop` - метод с самим игровым циклом. Там мы просто последовательно вызываем метод `mode` для игроков и делаем проверку, сколько живых кораблей осталось на досках, чтобы определить победу.
4. `start` - запуск игры. Сначала вызываем `greet`, а потом `loop`.

И останется просто создать экземпляр класса `Game` и вызвать метод `start`.

По ходу написания кода полезно проверять свой прогресс, тестируя написанные классы по отдельности. Для этого можно моделировать различные ситуации, например, создать список кораблей, добавить их на доску и попробовать сделать выстрел в разные точки. Для проверки функционала класса не обязательно иметь весь написанный код.