Паттерн MVC на примере Сапёра

18.06.2015
01:13

Введение

Паттерн Model-View-Controller (MVC) является крайне полезным при создании приложений со сложным графическим интерфейсом или поведением. Но и для более простых случаев он также подойдет. В этой заметке мы создадим игру сапер, спроектированную на основе этого паттерна. В качестве языка разработки выбран Python, однако особого значения в этом нет. Паттерны не зависят от конкретного языка программирования, и вы без труда сможете перенести получившуюся реализацию на любую другую платформу.

Коротко о паттерне MVC

Как следует из названия, паттерн MVC включает в себя 3 компонента: Модель, Представление и Контроллер. Каждый из компонентов выполняет свою роль и является взаимозаменяемым. Это значит, что компоненты связаны друг с другом лишь некими четкими интерфейсами, за которыми может лежать любая реализация. Такой подход позволяет подменять и комбинировать различные компоненты, обеспечивая необходимую логику работы или внешний вид приложения. Разберемся с теми функциями, которые выполняет каждый компонент.

Модель

Отвечает за внутреннюю логику работы программы. Здесь мы можем скрыть способы хранения данных, а также правила и алгоритмы обработки информации.

Например, для одного приложения мы можем создать несколько моделей. Одна будет отладочной, а другая рабочей. Первая может хранить свои данные в памяти или в файле, а вторая уже задействует базу данных. По сути это просто паттерн Стратегия.

Представление

Отвечает за отображение данных Модели. На этом уровне мы лишь предоставляем интерфейс для взаимодействия пользователя с Моделью. Смысл введения этого компонента тот же, что и в случае с предоставлением различных способов хранения данных на основе нескольких Моделей.

Например, на ранних этапах разработки мы можем создать простое консольное представление для нашего приложения, а уже потом добавить красиво оформленный GUI. Причем, остается возможность сохранить оба типа интерфейсов.

Кроме того, следует учитывать, что в обязанности Представления входит лишь своевременное отображение состояния Модели. За обработку действий пользователя отвечает Контроллер, о которым мы сейчас и поговорим.

Контроллер

Обеспечивает связь между Моделью и действиями пользователя, полученными в результате взаимодействия с Представлением. Координирует моменты обновления состояний Модели и

Представления. Принимает большинство решений о переходах приложения из одного состояния в другое.

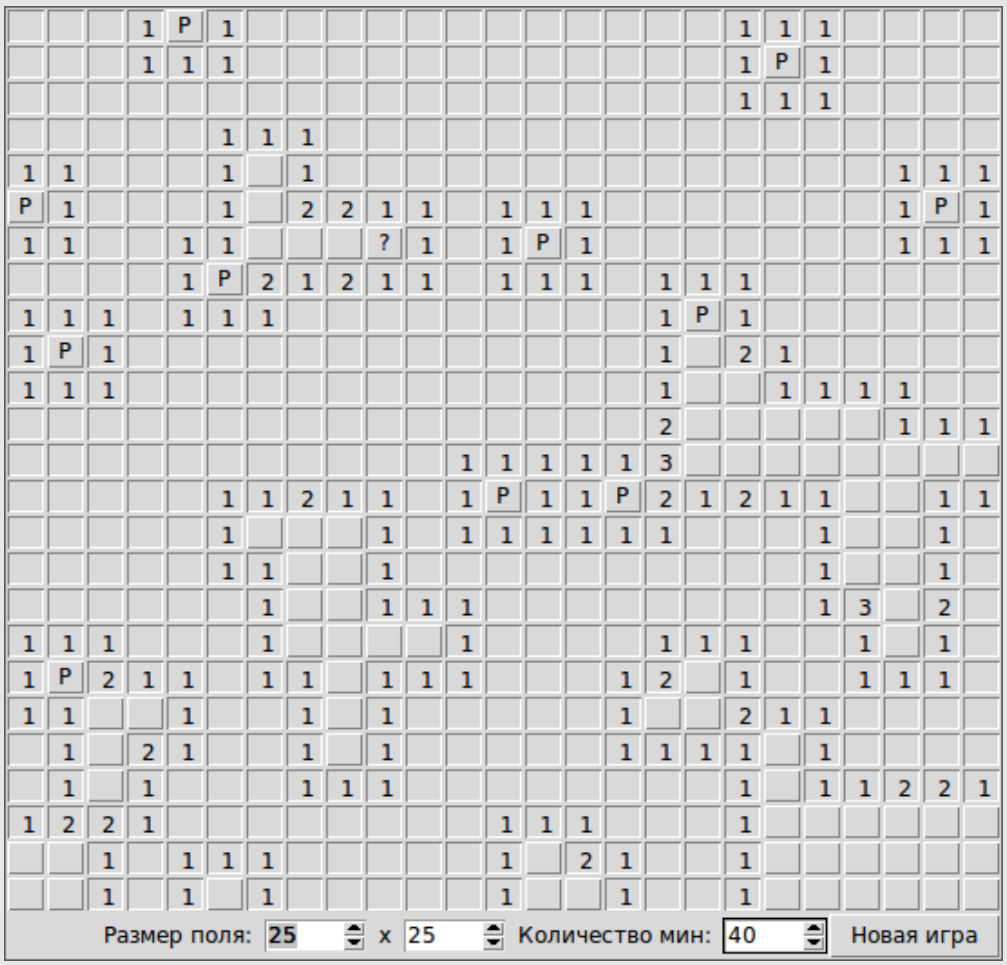
Фактически на каждое действие, которое может сделать пользователь в Представлении, должен быть определен обработчик в Контроллере. Этот обработчик выполнит соответствующие манипуляции над моделью и в случае необходимости сообщит Представлению о наличии изменений.

Спецификации игры Сапер

Достаточно теории. Теперь перейдем к практике. Для демонстрации паттерна MVC мы напишем несложную игру: Сапер. Правила игры достаточно простые:

1. Игровое поле представляет собой прямоугольную область, состоящую из клеток. В некоторых клетках случайным образом расположены мины, но игрок о них не знает;
2. Игрок может щелкнуть по любой клетке игрового поля левой или правой кнопками мыши;
3. Щелчок левой кнопки мыши приводит к тому, что клетка будет открыта. При этом, если в клетке находится мина, то игра завершается проигрышем. Если в соседних клетках, рядом с открытой, расположены мины, то на открытой клетке отобразится счетчик с числом мин вокруг. Если же мин вокруг открытой клетки нет, то каждая соседняя клетка будет открыта по тому же принципу. То есть клетки будут открываться до тех пор, пока либо не упрутся в границу игрового поля, либо не дойдут до уже открытых клеток, либо рядом с ними не окажется мина;
4. Щелчок правой кнопки мыши позволяет делать пометки на клетках. Щелчок на закрытой клетке помечает ее флажком, который блокирует ее состояние и предотвращает случайное открытие. Щелчок на клетке, помеченной флажком, меняет ее пометку на вопросительный знак. В этом случае клетка уже не блокируется и может быть открыта левой кнопкой мыши. Щелчок на клетке с вопросительным знаком возвращает ей закрытое состояние без пометок;
5. Победа определяется состоянием игры, при котором на игровом поле открыты все клетки, за исключением заминированных.

Пример того, что у нас получится приведен ниже:



UML-диаграммы игры Сапер

Прежде чем перейти к написанию кода неплохо было бы заранее продумать архитектуру приложения. Она не должна зависеть от языка реализации, поэтому для наших целей лучше всего подойдет UML.

Диаграмма Состояний игровой клетки

Любая клетка на игровом поле может находиться в одном из 4 состояний:

1. Клетка закрыта;
2. Клетка открыта;
3. Клетка помечена флажком;
4. Клетка помечена вопросительным знаком.

Здесь мы определили лишь состояния, значимые для Представления. Поскольку мины в процессе игры не отображаются, то и в базовом наборе соответствующего состояния не предусмотрено. Определим возможные переходы из одного состояния клетки в другое с помощью UML Диаграммы Состояний:

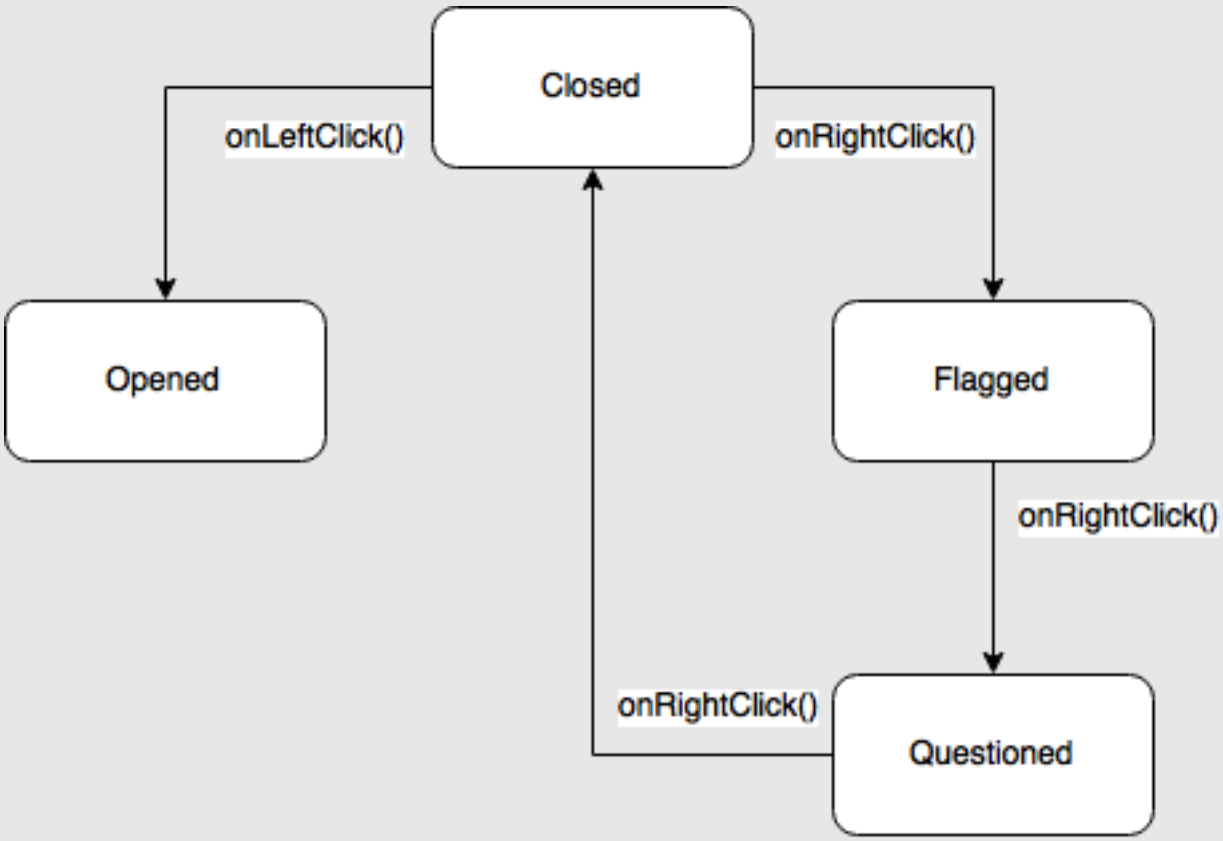
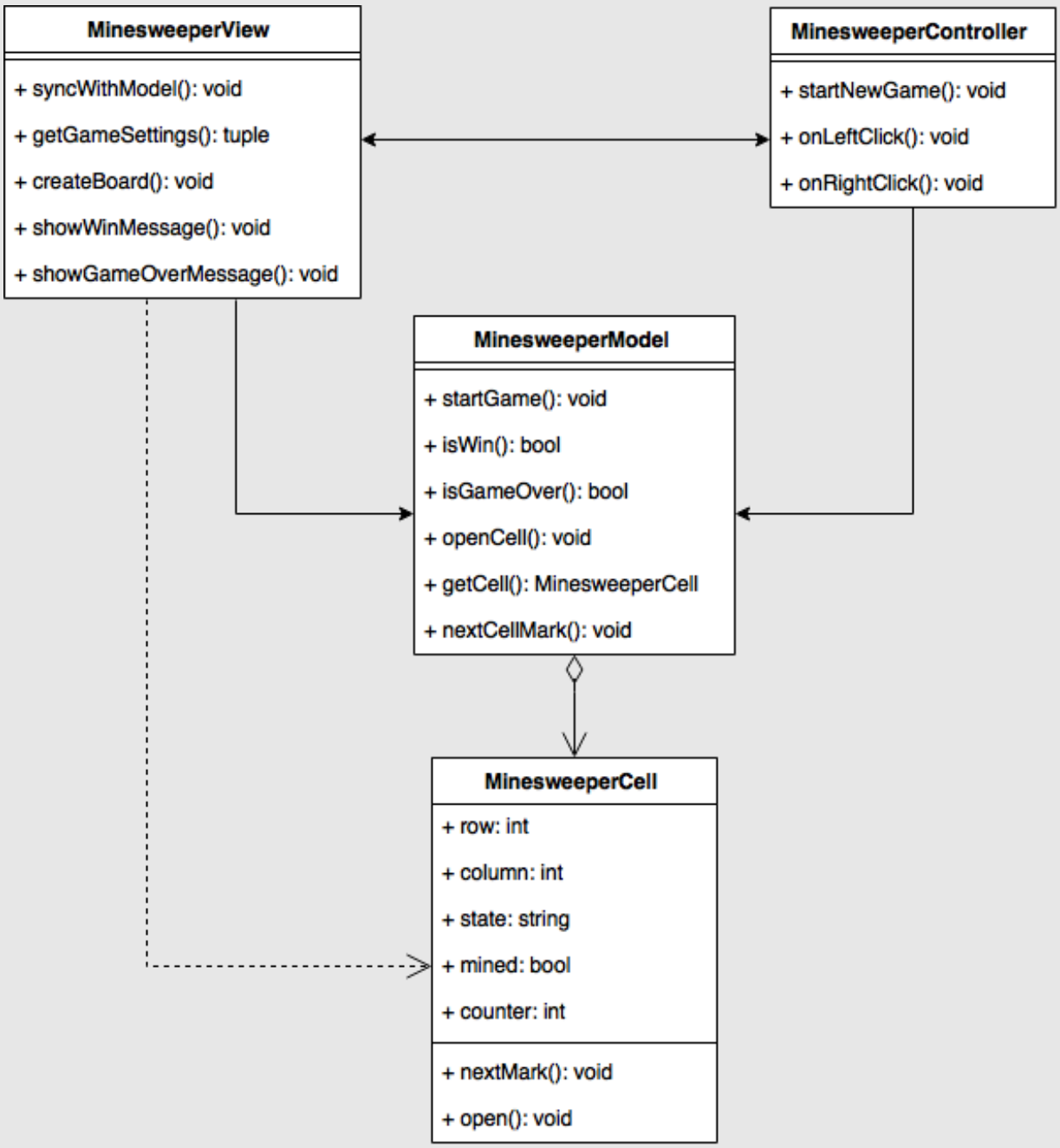


Диаграмма Классов игры Сапер

Поскольку мы решили создавать наше приложение на основе паттерна MVC, то у нас будет три основных класса: `MinesweeperModel`, `MinesweeperView` и `MinesweeperController`, а также вспомогательный класс `MinesweeperCell` для хранения состояния клетки. Рассмотрим их диаграмму классов:



Организация архитектуры довольно проста. Здесь мы просто распределили задачи по каждому классу в соответствии с принципами паттерна MVC:

1. В самом низу иерархии расположен класс игровой клетки `MinesweeperCell`. Он хранит позицию клетки, определяемую рядом `row` и столбцом `column` игрового поля; одно из состояний `state`, которые мы описали в предыдущем подразделе; информацию о наличии мины в клетке (`mined`) и счетчик мин в соседних клетках `counter`. Кроме того, у него есть два метода: `nextMark()` для циклического перехода по состояниям, связанным с пометками, появляющимися в результате щелчка правой кнопкой мыши, а также `open()`, который обрабатывает событие, связанное с щелчком левой кнопкой мыши;
2. Чуть выше расположен класс Модели `MinesweeperModel`. Он является контейнером для игровых клеток `MinesweeperCell`. Его первый метод `startGame()` подготавливает игровое поле для начала игры. Метод `isWin()` делает проверку игрового поля на состояние выигрыша и возвращает истину, если игрок победил, иначе возвращается ложь. Для проверки проигрыша предназначен аналогичный метод `isGameOver()`. Методы `openCell()` и `nextCellMark()` всего лишь

делегируют действия соответствующим клеткам на игровом поле, а метод `getCell()` возвращает запрашиваемую игровую клетку;
3. Класс Представления `MinesweeperView` включает следующие методы: `syncWithModel()` - обеспечивает перерисовку Представления для отображения актуального состояния игрового поля в Модели; `getGameSettings()` - возвращает настройки игры, заданные пользователем; `createBoard()` - создает игровое поле на основе данных Модели; `showWinMessage()` и `showGameOverMessage()` соответственно отображают сообщения о победе и проигрыше;
4. И наконец класс Контроллера `MinesweeperController`. В нем определено всего три метода на каждое возможное действие игрока: `startNewGame()` отвечает за нажатие на кнопке "Новая игра" в интерфейсе Представления; `onLeftClick()` и `onRightClick()` обрабатывают щелчки по игровым клеткам левой и правой кнопками мыши соответственно.

Реализация игры Сапер на Python

Пришло время заняться реализацией нашего проекта. В качестве языка разработки выберем Python. Тогда класс Представления будем писать на основе модуля `tkinter`.

Но начнем с Модели.

Модель `MinsweeperModel`

Реализация модели на языке Python выглядит следующим образом:

```
MIN_ROW_COUNT = 5

MAX_ROW_COUNT = 30


MIN_COLUMN_COUNT = 5

MAX_COLUMN_COUNT = 30


MIN_MINE_COUNT = 1

MAX_MINE_COUNT = 800


class MinesweeperCell:

    # Возможные состояния игровой клетки:

    #   closed - закрыта

    #   opened - открыта

    #   flagged - помечена флажком

    #   questioned - помечена вопросительным знаком


    def __init__( self, row, column ):

        self.row = row

        self.column = column

        self.state = 'closed'

        self.mined = False

        self.counter = 0
```

```
    markSequence = [ 'closed', 'flagged', 'questioned' ]
    def nextMark( self ):
        if self.state in self.markSequence:
            stateIndex = self.markSequence.index( self.state )
            self.state = self.markSequence[ ( stateIndex + 1 ) % len( self.markSequence ) ]

    def open( self ):
        if self.state != 'flagged':
            self.state = 'opened'



class MinesweeperModel:
    def __init__( self ):
        self.startGame()

    def startGame( self, rowCount = 15, columnCount = 15, mineCount = 15 ):
        if rowCount in range( MIN_ROW_COUNT, MAX_ROW_COUNT + 1 ):
            self.rowCount = rowCount

        if columnCount in range( MIN_COLUMN_COUNT, MAX_COLUMN_COUNT + 1 ):
            self.columnCount = columnCount

        if mineCount < self.rowCount * self.columnCount:
            if mineCount in range( MIN_MINE_COUNT, MAX_MINE_COUNT + 1 ):
                self.mineCount = mineCount
        else:
            self.mineCount = self.rowCount * self.columnCount - 1

        self.firstStep = True
        self.gameOver = False
        self.cellsTable = []
        for row in range( self.rowCount ):
            cellsRow = []
            for column in range( self.columnCount ):
                cellsRow.append( MinesweeperCell( row, column ) )
            self.cellsTable.append( cellsRow )

    def getCell( self, row, column ):
        if row < 0 or column < 0 or self.rowCount <= row or self.columnCount <= column:
            return None

        return self.cellsTable[ row ][ column ]
```

```
    def isWin( self ):
        for row in range( self.rowCount ):
            for column in range( self.columnCount ):
                cell = self.cellsTable[ row ][ column ]
                if not cell.mined and ( cell.state != 'opened' and cell.state != 'flagged' ):
                    return False

        return True

    def isGameOver( self ):
        return self.gameOver

    def openCell( self, row, column ):
        cell = self.getCell( row, column )
        if not cell:
            return

        cell.open()

        if cell.mined:
            self.gameOver = True
            return

        if self.firstStep:
            self.firstStep = False
            self.generateMines()

        cell.counter = self.countMinesAroundCell( row, column )
        if cell.counter == 0:
            neighbours = self.getCellNeighbours( row, column )
            for n in neighbours:
                if n.state == 'closed':
                    self.openCell( n.row, n.column )

    def nextCellMark( self, row, column ):
        cell = self.getCell( row, column )
        if cell:
            cell.nextMark()

    def generateMines( self ):
        for i in range( self.mineCount ):
            while True:
                row = random.randint( 0, self.rowCount - 1 )
```

```
                column = random.randint( 0, self.columnCount - 1 )
                cell = self.getCell( row, column )
                if not cell.state == 'opened' and not cell.mined:
                    cell.mined = True
                    break


    def countMinesAroundCell( self, row, column ):
        neighbours = self.getCellNeighbours( row, column )
        return sum( 1 for n in neighbours if n.mined )


    def getCellNeighbours( self, row, column ):
        neighbours = []
        for r in range( row - 1, row + 2 ):
            neighbours.append( self.getCell( r, column - 1 ) )
            if r != row:
                neighbours.append( self.getCell( r, column ) )
            neighbours.append( self.getCell( r, column + 1 ) )


        return filter( lambda n: n is not None, neighbours )
```

В верхней части мы определяем диапазон допустимых настроек игры:

```
MIN_ROW_COUNT = 5
MAX_ROW_COUNT = 30


MIN_COLUMN_COUNT = 5
MAX_COLUMN_COUNT = 30


MIN_MINE_COUNT = 1
MAX_MINE_COUNT = 800
```

Вообще, эти настройки можно было сделать тоже частью Модели. Однако размеры поля и количество мин достаточно статичная информация и вряд ли будет часто меняться.

Затем мы определили класс игровой клетки `MinesweeperCell`. Она оказалась достаточно простой. В конструкторе класса происходит инициализация полей клетки значениями по умолчанию. Далее для упрощения реализации циклических переходов по состояниям мы используем вспомогательный список `markSequence`. Если клетка находится в состоянии `'opened'`, которое не входит в этот список, то в методе `nextMark()` ничего не произойдет, иначе клетка попадает в следующее состояние, причем, из последнего состояния `'questioned'` она "перепрыгивает" в начальное состояние `'closed'`. В методе `open()` мы проверяем состояние клетки, и если оно не равно `'flagged'`, то клетка переходит в открытое состояние `'opened'`.

Далее следует определение класса Модели `MinesweeperModel`. Метод `startGame()` осуществляет компоновку игрового поля по переданным ему параметрам `rowCount`, `columnCount` и `mineCount`. Для каждого из параметров происходит проверка на попадание в допустимый диапазон значений. Если переданное значение находится вне диапазона, то сохраняется то значение параметра игрового поля не меняется. Следует отметить, что для числа мин предусмотрена дополнительная проверка. Если переданное количество мин превышает размер поля, то мы ограничиваем его количеством клеток без единицы. Хотя, конечно, такая игра особого смысла

не имеет и будет закончена в один шаг, поэтому вы можете придумать какое-нибудь свое правило на такой случай.

Игровое поле хранится в виде списка списков клеток в переменной `cellsTable`. Причем, обратите внимание, что в методе `startGame()` у клеток устанавливается лишь значение позиции, но мины еще не расставляются. Зато определяется переменная `firstStep` со значением `True`. Это нужно для того, чтобы убрать элемент случайности из первого хода и не допускать мгновенный проигрыш. Мины будут расставляться после первого хода в оставшихся клетках.

Метод `getCell()` просто возвращает клетку игрового поля по строке `row` и столбцу `column`. Если значение строки или столбца неверно, то возвращается `None`.

Метод `isWin()` возвращает `True`, если все оставшиеся не открытые клетки игрового поля заминированы, то есть в случае победы, иначе вернется `False`. А метод `isGameOver()` просто возвращает значение атрибута класса `gameOver`.

В методе `openCell()` происходит делегирование вызова `open()` объекту игровой клетки, которая расположена на игровом поле в позиции, указанной в параметрах метода. Если открытая клетка оказалось заминированной, то мы устанавливаем значение `gameOver` в `True` и выходим из метода. Если игра еще не окончена, то мы смотрим, а не первый ли это ход, проверяя значение `firstStep`. Если ход и правда первый, то произойдет расстановка мин по игровому полю с помощью вспомогательного метода `generateMines()`, о которой мы поговорим немного позже. Далее мы подсчитываем количество заминированных соседних клеток и устанавливаем соответствующее значение атрибута `counter` для обрабатываемой клетки. Если счетчик `counter` равен нулю, то мы запрашиваем список соседних клеток с помощью метода `getCellNeighbours()` и осуществляем рекурсивный вызов метода `openCell()` для всех закрытых "соседей", то есть для клеток со статусом `'closed'`.

Метод `nextCellMark()` всего лишь делегирует вызов методу `nextMark()` для клетки, расположенной на переданной позиции.

Расстановка мин происходит в методе `generateMines()`. Здесь мы просто случайным образом выбираем позицию на игровом поле и проверяем, чтобы клетка на этой позиции не была открыта и не была уже заминирована. Если оба условия выполнены, то мы устанавливаем значение атрибута `mined` равным `True`, иначе продолжаем поиск другой свободной клетки. Не забудьте, что для того, чтобы использовать на Python модуль `random` нужно явным образом его импортировать командой `import random`.

Метод подсчета количества мин `countMinesAroundCell()` вокруг некоторой клетки игрового поля полностью основывается на методе `getCellNeighbours()`. Запрос "соседей" клетки в методе `getCellNeighbours()` тоже реализован крайне просто. Не думаю, что у вас возникнут с ним проблемы.

Представление `MinesweeperView`

Теперь займемся представлением. Код класса `MinesweeperView` на Python представлен ниже:

```
class MinesweeperView( Frame ):

    def __init__( self, model, controller, parent = None ):

        Frame.__init__( self, parent )

        self.model = model

        self.controller = controller

        self.controller.setView( self )

        self.createBoard()


        panel = Frame( self )

        panel.pack( side = BOTTOM, fill = X )
```

```
        Button( panel, text = 'Новая игра', command = self.controller.startNewGame ).pack( side = RIGHT )

        self.mineCount = StringVar( panel )
        self.mineCount.set( self.model.mineCount )
        Spinbox(
            panel,
            from_ = MIN_MINE_COUNT,
            to = MAX_MINE_COUNT,
            textvariable = self.mineCount,
            width = 5
        ).pack( side = RIGHT )
        Label( panel, text = ' Количество мин: ' ).pack( side = RIGHT )

        self.rowCount = StringVar( panel )
        self.rowCount.set( self.model.rowCount )
        Spinbox(
            panel,
            from_ = MIN_ROW_COUNT,
            to = MAX_ROW_COUNT,
            textvariable = self.rowCount,
            width = 5
        ).pack( side = RIGHT )

        Label( panel, text = ' x ' ).pack( side = RIGHT )

        self.columnCount = StringVar( panel )
        self.columnCount.set( self.model.columnCount )
        Spinbox(
            panel,
            from_ = MIN_COLUMN_COUNT,
            to = MAX_COLUMN_COUNT,
            textvariable = self.columnCount,
            width = 5
        ).pack( side = RIGHT )
        Label( panel, text = 'Размер поля: ' ).pack( side = RIGHT )

    def syncWithModel( self ):
        for row in range( self.model.rowCount ):
            for column in range( self.model.columnCount ):
                cell = self.model.getCell( row, column )
                if cell:
                    btn = self.buttonsTable[ row ][ column ]
```

```
                if self.model.isGameOver() and cell.mined:
                    btn.config( bg = 'black', text = '' )

                if cell.state == 'closed':
                    btn.config( text = '' )
                elif cell.state == 'opened':
                    btn.config( relief = SUNKEN, text = '' )
                    if cell.counter > 0:
                        btn.config( text = cell.counter )
                    elif cell.mined:
                        btn.config( bg = 'red' )
                elif cell.state == 'flagged':
                    btn.config( text = 'P' )
                elif cell.state == 'questioned':
                    btn.config( text = '?' )

    def blockCell( self, row, column, block = True ):
        btn = self.buttonsTable[ row ][ column ]
        if not btn:
            return

        if block:
            btn.bind( '<Button-1>', 'break' )
        else:
            btn.unbind( '<Button-1>' )

    def getGameSettings( self ):
        return self.rowCount.get(), self.columnCount.get(), self.mineCount.get()

    def createBoard( self ):
        try:
            self.board.pack_forget()
            self.board.destroy()

            self.rowCount.set( self.model.rowCount )
            self.columnCount.set( self.model.columnCount )
            self.mineCount.set( self.model.mineCount )
        except:
            pass

        self.board = Frame( self )
        self.board.pack()
```

```
        self.buttonsTable = []
        for row in range( self.model.rowCount ):
            line = Frame( self.board )
            line.pack( side = TOP )
            self.buttonsRow = []
            for column in range( self.model.columnCount ):
                btn = Button(
                    line,
                    width = 2,
                    height = 1,
                    command = lambda row = row, column = column: self.controller.onLeftClick( row, column
),
                    padx = 0,
                    pady = 0
                )
                btn.pack( side = LEFT )
                btn.bind(
                    '<Button-3>',
                    lambda e, row = row, column = column: self.controller.onRightClick( row, column )
                )
                self.buttonsRow.append( btn )

            self.buttonsTable.append( self.buttonsRow )

    def showWinMessage( self ):
        showinfo( 'Поздравляем!', 'Вы победили!' )

    def showGameOverMessage( self ):
        showinfo( 'Игра окончена!', 'Вы проиграли!' )
```

Наше Представление основано на классе `Frame` из модуля `tkinter`, поэтому не забудьте выполнить соответствующую команду импорта: `from tkinter import *`. В конструкторе класса передаются Модель и Контроллер. Сразу же вызывается метод `createBoard()` для компоновки игрового поля из клеток. Скажу заранее, что для этой цели мы будем использовать обычные кнопки `Button`. Затем создается `Frame`, который будет выполнять роль нижней панели для указания параметров игры. На эту панель мы последовательно помещаем кнопку "Новая игра", обработчиком которой становится наш Контроллер с его методом `startNewGame()`, а затем три счетчика `Spinbox` для того, чтобы игрок мог указать размер игрового поля и число мин.

Метод `syncWithModel()` просто проходит в двойном цикле по каждой игровой клетке и изменяет соответствующим образом вид кнопки, которая представляет ее в нашем графическом интерфейсе. Для простоты я использовал текстовые символы для вывода обозначений, однако не так сложно поменять текст на графику из внешних графических файлов.

Кроме того, обратите внимание, что для представления открытой клетки мы используем стиль кнопки `SUNKEN`. А в случае проигрыша открываем местоположение всех мин на игровом поле, показывая соответствующие кнопки черным цветом, а кнопку, отвечающую последней открытой клетке с миной, выделяем красным цветом:

Следующий метод `blockCell()` выполняет вспомогательную роль и позволяет контроллеру устанавливать состояние блокировки для кнопок. Это нужно для предотвращения случайного открытия игровых клеток, помеченных флажком, и достигается путем установки пустого обработчика щелчка левой кнопки мыши.

Метод `getGameSettings()` всего лишь возвращает значения размещенных в нижней панели счетчиков с размером игрового поля и количеством мин.

Создание представления игрового поля осуществляется в методе `createBoard()`. В первую очередь идет попытка удаления старого игрового поля, если оно существовало, а также мы пробуем установить значения счетчиков из панели в соответствии с текущей конфигурацией Модели. Затем создается новый `Frame`, который мы назовем `board`, для представления игрового поля. Таблицу кнопок `buttonsTable` мы компонуем по тому же принципу, что и игровые клетки в Модели с помощью двойного цикла. Обработчики каждой кнопки привязываются к методам Контроллера `onLeftClick()` и `onRightClick()` для щелчка левой и правой кнопок мыши соответственно.

Последние два метода `showWinMessage()` и `showGameOverMessage()` всего лишь отображают диалоговые окна с соответствующими сообщениями с помощью функции `showinfo()`. Для того, чтобы ей воспользоваться вам понадобится импортировать еще один модуль: `from tkinter.messagebox import *`.

Контролер `MinesweeperController`

Вот мы и дошли до реализации Контроллера:

```
class MinesweeperController:

    def __init__( self, model ):

        self.model = model


    def setView( self, view ):

        self.view = view


    def startNewGame( self ):

        gameSettings = self.view.getGameSettings()

        try:

            self.model.startGame( *map( int, gameSettings ) )

        except:

            self.model.startGame( self.model.rowCount, self.model.columnCount, self.model.mineCount )


        self.view.createBoard()


    def onLeftClick( self, row, column ):

        self.model.openCell( row, column )

        self.view.syncWithModel()

        if self.model.isWin():

            self.view.showWinMessage()

            self.startNewGame()

        elif self.model.isGameOver():

            self.view.showGameOverMessage()

            self.startNewGame()


    def onRightClick( self, row, column ):

        self.model.nextCellMark( row, column )

        self.view.blockCell( row, column, self.model.getCell( row, column ).state == 'flagged' )

        self.view.syncWithModel()
```

Для привязки Представления к Контроллеру мы добавили метод `setView()`. Это объясняется тем, что если бы мы хотели передать Представление в конструктор, то это Представление должно было бы уже существовать до момента создания Контроллера. А тогда подобное решение с дополнительным методом для привязки просто перешло бы от Контроллера к Представлению, в которым бы появился метод `setController()`.

Метод-обработчик для нажатия на кнопке "Новая игра" `startNewGame()` сначала запрашивает параметры игры, введенные в Представление. Параметры игры возвращаются в виде кортежа из трех компонент, которые мы пытаемся преобразовать в `int`. Если все пройдет нормально, то мы передаем эти значения в метод Модели `startGame()` для построения игрового поля. Если же что-то пойдет не так, то мы просто пересоздадим игровое поле со старыми параметрами. А в завершении мы направляем запрос на создание нового отображения игрового поля в Представлении с помощью вызова метода `createBoard()`.

Обработчик `onLeftClick()` сначала указывает Модели на необходимость открыть игровую клетку в выбранной игроком позиции. Затем сообщает Представлению о том, что состояние

Модели изменилось и предлагает все перерисовать. Затем происходит проверка Модели на состояние победы или проигрыша. Если что-то из этого произошло, то сначала в Представление направляется запрос на отображение соответствующего уведомления, а затем происходит вызов обработчика `startNewGame()` для начала новой игры.

Щелчок правой кнопкой мыши обрабатывается в методе `onRightClick()`. В первой строке происходит вызов метода Модели `nextCellMark()` для циклической смены метки выбранной игровой клетки. В зависимости от нового состояния клетки Представлению отправляется запрос на установку или снятие блокировки на соответствующую кнопку. А в конце вновь обеспечивается обновление вида Представления для отображения актуального состояния Модели.

Комбинируем Модель, Представление и Контроллер

Теперь осталось лишь соединить все элементы в рамках нашей реализации Сапера на основе паттерна MVC и запустить игру:

```
model = MinesweeperModel()

controller = MinesweeperController( model );

view = MinesweeperView( model, controller )

view.pack()

view.mainloop()
```

Заключение

Вот мы и рассмотрели паттерн MVC. Коротко прошлись по теории. А потом по шагам создали полноценное игровое приложение, пройдя путь от постановки задачи и проектирования архитектуры до реализации на языке программирования Python с использованием графического модуля `tkinter`.