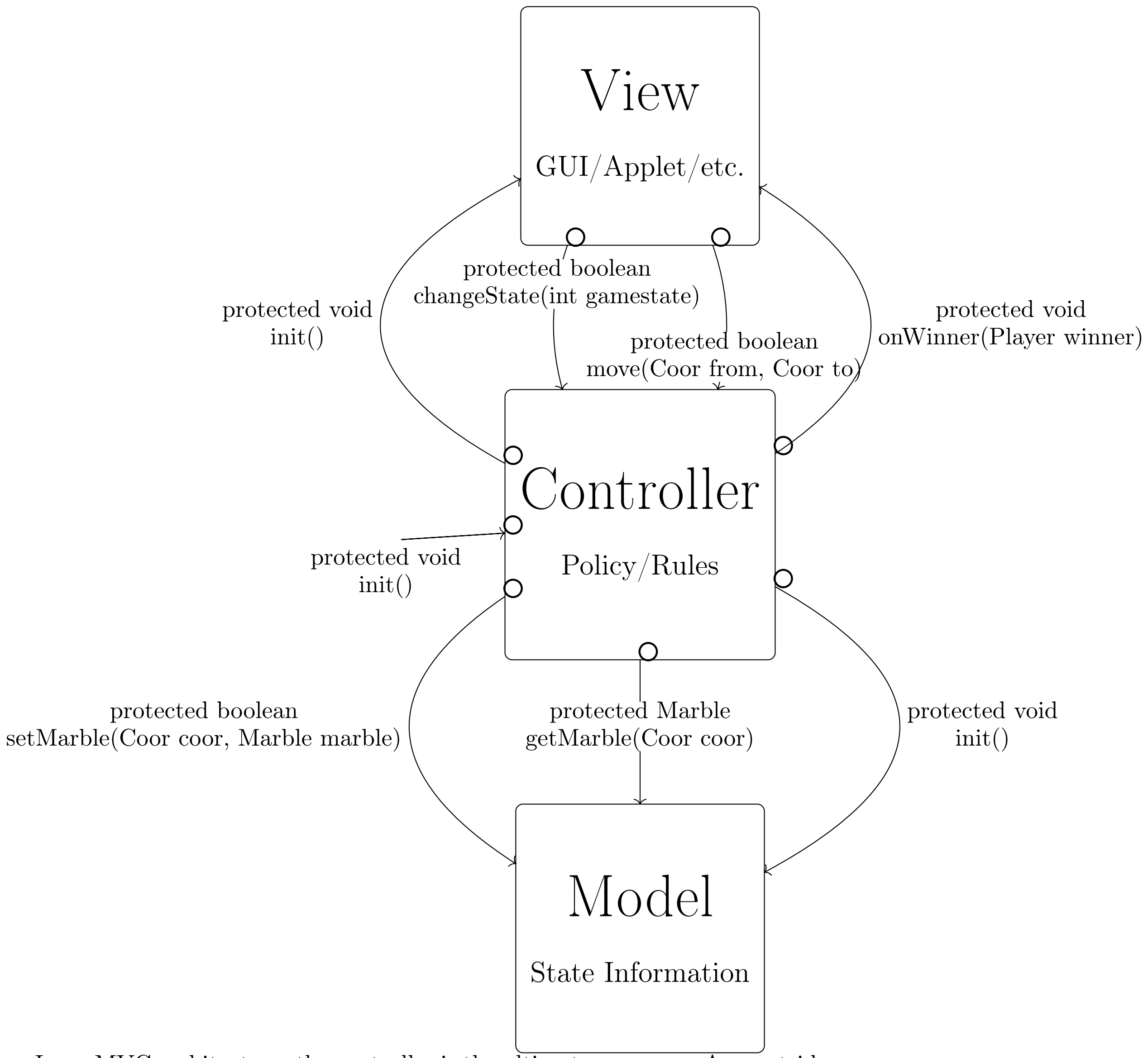
1 Overall Architecture


View
GUI/Applet/etc.
protected void init()
protected boolean changeState(int gamestate)
protected boolean move(Coor from, Coor to)
protected void onWinner(Player winner)
Controller
Policy/Rules
protected void init()
protected boolean setMarble(Coor coor, Marble marble)
protected Marble getMarble(Coor coor)
protected void init()
Model
State Information

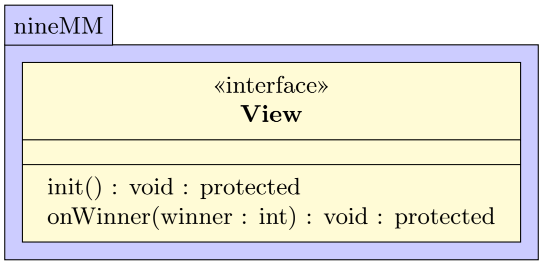
In an MVC architecture, the controller is the ultimate governor. Any outside requests from the view must go through the controller; never is the view allowed to directly contact the model and vice versa. Note that any request sent from the view is just that - a request, which can be denied by the controller, as it is responsible for all policy and security. The model should not have a need to ask the controller for anything for our project, its job is simply to maintain the game state at the order of the controller.

The model, view, and controller are simply interfaces, with a small, but rigidly-defined API. This allows custom classes to be created for different purposes to do those jobs. For example, one class implementing the model could be written to use a two-dimensional array, and another implementation could be written to store its data in a SQL database.

All constants and methods are marked as *protected*, which, if each module compromising the model, view, and controller were placed in the same package, would give the model and view access to call the controller methods and vice versa. If this project were to become part of a larger project, this would prevent other code accessing and calling things they were never meant to. For example, the class containing the main() method which starts the program should not be a part of this package, as it has no need to do much more than create an instance of whichever class implements the controller interface. An unfortunate side effect of this design is that the view can also call methods in the model and vice versa. This can be mitigated by each developer only calling the methods they are supposed to. See diagram above.

2 Individual Components

2.1 View


nineMM
«interface»
View
init() : void : protected
onWinner(winner : int) : void : protected

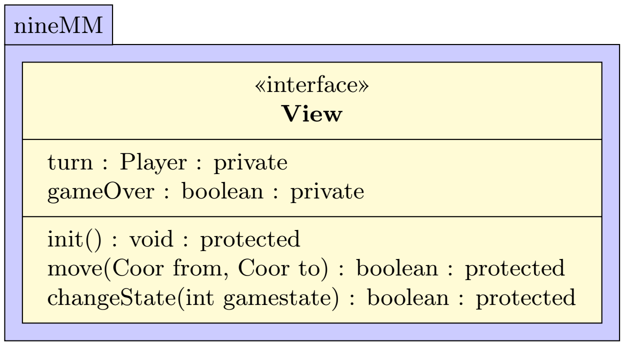
protected void init()

The purpose of this method is to be called by the controller in the event that the game needs to be reset, in which case the view needs to do whatever is necessary to the display (GUI in some cases) to ensure that the marbles (game pieces) are not shown on the game board.

protected void onWinner(Player winner)

onWinner() is a callback to be used when the controller decides to issue an event that the game is over. There is not necessarily a winner, as null may be returned, in which case it is a stalemate. Details of the Player class are further down.

2.2 Controller


nineMM
«interface»
View
turn : Player : private
gameOver : boolean : private
init() : void : protected
move(Coor from, Coor to) : boolean : protected
changeState(int gamestate) : boolean : protected

This architecture suggests that only the view will ask the controller for anything. It should be understood that when calling a method in the controller, it is only a request, which can be denied by the controller for any reason (policies, security, etc.). This interface's methods return a boolean to indicate admitance of an operation (true) or denial (false). However, this does not give very informative error messages to the view. An alternate approach would be to have them return a String. If the return value is null, success is assumed, otherwise the String will hold an informative error message, which the view can use however it wants.

The variables turn and gameOver will be necessary to enforce policy, to keep track of whose turn it is, and whether the game is over or not. Multiple views can simultaneously exist (possibly threaded or networked), and it is assumed that they will ask for operations when it is not allowed (for example, they may try to move a marble when it is not their turn).

protected void init()

init() not only needs to perform whatever functions are necessary to initialize the controller, but also needs to spawn an instance of a model (if one does not already exist, or if a new type of model is needed for whatever reason) as well as spawn as many instances of a view are needed, and call init() on those views. The constructor for the controller needs to have a way to know how many and of what type of views and models are necessary. Note that it would be good practice to only have one model active at a time, having multiple models around leads to state inconsistency.

protected boolean move(Coor from, Coor to)

The view will want to ask the controller at some point to move a marble from one position to another. The controller should allow this if the rules of the game permit (i.e. both the from and to coordinates exist, the to position is empty, etc.). True should be returned if the move is allowed, false otherwise.

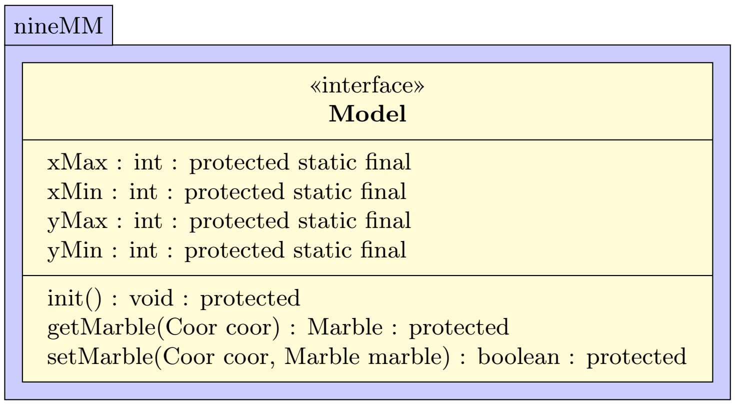
The Coor class is described later.

protected boolean changeState(int gamestate)

From time to time, the view will want to request a change in the overall game state, for example: starting a new game or changing a player color (meaning the order in which turns are taken, the controller could care less about what color is displayed). gamestate is to be a constant defined in the GameState class. This is done because we anticpate more functionality than simply starting a new game later on, as requirements change. True is returned if the operation is permitted, false otherwise.

The GameState class is described later on.

2.3 Model


nineMM
«interface»
Model
xMax : int : protected static final
xMin : int : protected static final
yMax : int : protected static final
yMin : int : protected static final
init() : void : protected
getMarble(Coor coor) : Marble : protected
setMarble(Coor coor, Marble marble) : boolean : protected

It is possible for the game state to be stored in mulitple ways. The simplest way would be for the marbles to be stored in a two-dimensional array. However, it can be as complex as a SQL database server on a networked node far away. As long as this interface is implemented, any model module should serve as a drop-in replacement for any other.

The member variables (xMax, xMin, yMax, yMin) are mostly for the convenience of the Coor class described later.

protected void init()

This should "zero out" the gameboard. Intuitively, this means marking each spot on the gameboard as empty, however that needs to be implemented for a particular model.

protected Marble getMarble(Coor coor)

This should return a Marble object, which holds the status of that marble on the gameboard (essentially the color, but also a special value of empty, i.e. no marble at that spot). Definsive programming tactics can be used where a value of NaM (not a marble) can be returned if the controller asks for something rediculous. It is true that ints could be used instead of a Marble object, but a Marble class is provided in case project requirements change suddenly that requires us to track each game piece individually for whatever reason, we can do so using Object.hash().

The Marble class is described later on.

protected boolean setMarble(Coor coor, Marble marble)

This is provided so the controller can move marbles about however it sees fit. A return type of boolean is provided for the following reasons:

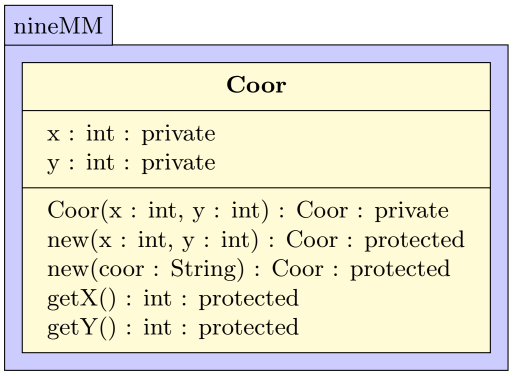
- A particular model (SQL DB for example) may be making networking RPC calls. We need a way to indicate failure in this case.
- The controller may ask to put a marble in a place that doesn't exist.

In lieu of returning a boolean (which does not give informative error messages), a String could be returned in the style described above in the Controller section.

2.4 Coor

While it is true that pure integers can be used to express a coordinate, a class is attractive for the following reasons:

- We (or the user) may want to refer to a coordinate using a different language (i.e. move marble at a5 to spot at b7) in the terminology used with chess boards instead of using an integer x and an integer y.
- Automatic bounds checking. When creating a new Coor, we can automatically return null if the coordinates asked for do not fit the model.


nineMM
Coor
x : int : private
y : int : private
Coor(x : int, y : int) : Coor : private
new(x : int, y : int) : Coor : protected
new(coor : String) : Coor : protected
getX() : int : protected
getY() : int : protected

private Coor(int x, int y)

See immediately below.

protected new(int x, int y) protected new(String coor)

The pseudo-constructor new() has a job to map the input arguments into an integer x and y that represents the coordinate asked for. In the former case, it is self explanatory, but the later case needs some logic to translate a string (i.e. "a5") into x and y coordinates.

Bounds checking should be performed in all cases to ensure that the coordinate asked for fits the model. This is where the xMin, xMax, yMin, and yMax in the model implementation come in handy. null should be returned when out of bounds, an instance of Coor returned otherwise, where the real constructor is called, which sets the x and y coordinates for that object.

Why use a pseudo-constructor? The Java language does not allow us to *return NULL;* in a constructor like some other languages do. Therefore we use a **Factory Method Pattern**, which is an OO design pattern which allows us to do that. Some sample code is shown below:

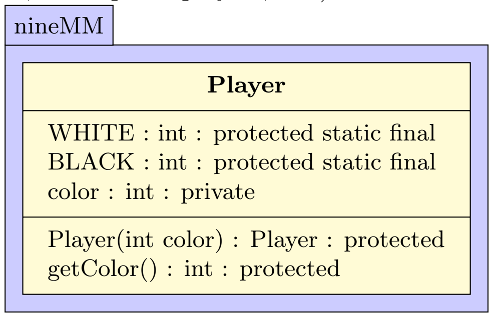
```
protected class Coor
{
        ...

        //private constructor
        private Coor(int x, int y)
        {
                ...
        }

        private boolean bounds\_check\_good(int x, int y)
        {
                ...
        }

        //overloaded pseudoconstructors
        protected Coor new(int x, int y)
        {
                if(bounds\_check\_good(x,y) ) return new Coor(x,y);
                return null; //returned otherwise
        }
        protected Coor new(String coor)
        {
                ...
        }

        ...
}
```

protected int getX() protected int getY()

Getters for the x and y member variables. Setters are not provided so that another module (model, view, etc.) cannot do something nasty to it.

2.5 Player

A concrete representation of a player so that the controller could keep track of them. While possible to use unique integers to tell players apart, a class will provide us with greater flexibility if requirements change later on (i.e. 3-player game, 2 computer players, etc.).


nineMM
Player
WHITE : int : protected static final
BLACK : int : protected static final
color : int : private
Player(int color) : Player : protected
getColor() : int : protected

The member variables are supposed to be unique and have the same value as in the Marble class (i.e. Player.WHITE == Marble.WHITE). The prefered method for doing this is by assignment.

protected Player(int color)

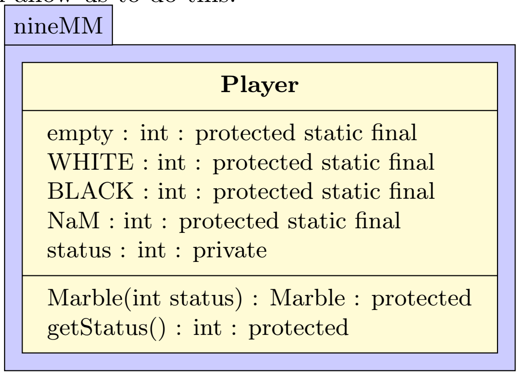
This constructor simply sets the color of the object to the argement.

protected getColor()

A simple getter for the color member variable.

2.6 Marble

Again, a marble could be tracked simply by an integer value indicating color (ownership), but in case project requirements change that require us in some way to track game pieces individually instead of as fungible colors. Object.hashCode() will allow us to do this.


nineMM
Player
empty : int : protected static final
WHITE : int : protected static final
BLACK : int : protected static final
NaM : int : protected static final
status : int : private
Marble(int status) : Marble : protected
getStatus() : int : protected

The *protected static final* member variables are just constants. *status* is meant to be a member variable that indicates which type of marble is at some position on the gameboard. NaM is a special value, which is not empty to indicate a place that does not exist on the game board, or which otherwise does not make sense.

protected Marble(int status)

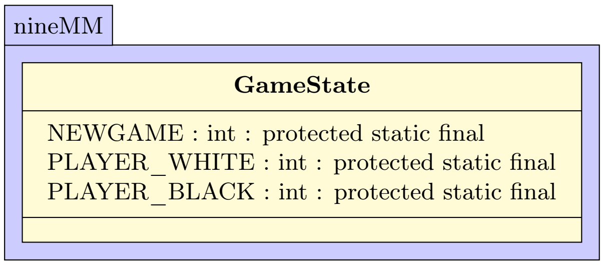
A straightforward constructor that creates a Marble object initialized with the given status.

protected int getStatus()

A simple getter for the *status* member variable. A setter is not provided to prevent other unauthorized modules from making changes.

2.7 GameState

Right now, the GameState class holds constants to be used by changeState() in the controller simply because there did not seem to be a better place to put them.


nineMM
GameState
NEWGAME : int : protected static final
PLAYER_WHITE : int : protected static final
PLAYER_BLACK : int : protected static final

PLAYER_WHITE should hold the same value as Marble.WHITE, which should be the same as Player.WHITE. PLAYER_BLACK should be initialized in a similar manner.

3 Alternate Approaches

This architecture is all in one package, which allows any module to call any *protected* method (or any method with laxer conditions). If possible, a package hiearchy would prevent unauthorized access and calls, but the Java language may not permit this. Further research would be required to verify or expunge this.