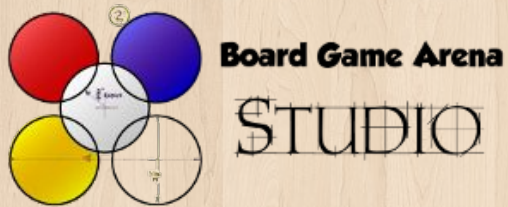


**Board Game Arena**

**STUDIO**

Online boardgaming development platform

**Focus on : BGA Game state machine**



# Why a game state machine?

Because game states will make your life easier.

In all board games, you have to play actions in a specific order : do this is phase 1, then do this in phase 2, then in phase 3 you can do actions A, B or C, ...

With BGA Studio framework you specify a **finite-state machine** for your game. This state machine allows you to structure and to implement the rules of a game very efficiently.

Understanding game states is essential to develop a game on Board Game Arena platform.



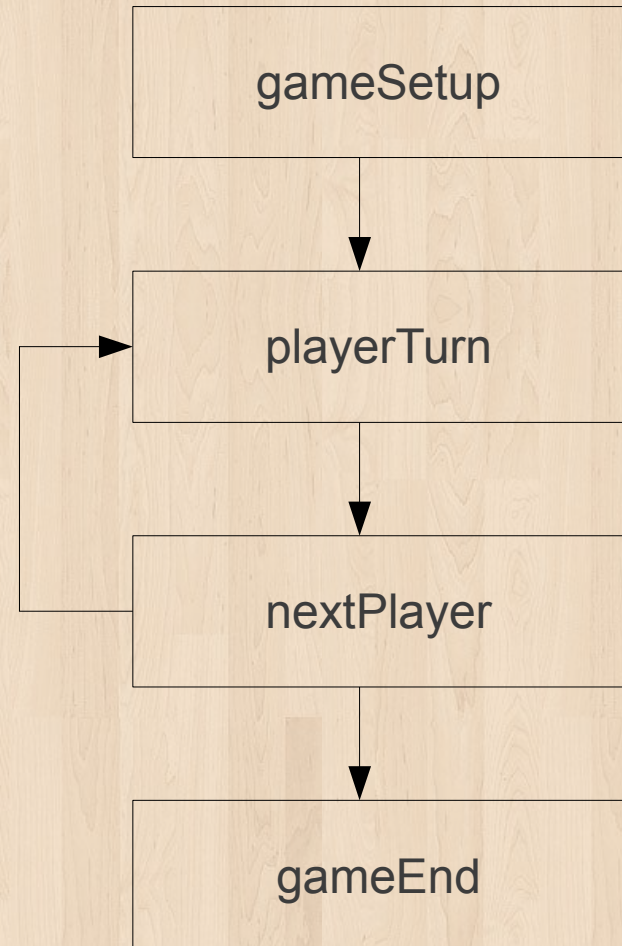
Board Game Arena  
STUDIO

# What is a game state machine?

A simple game state machine diagram with 4 different game states is displayed on the right.

This machine correspond for example to **chess** game, or **reversi** (BGA Studio example) game :

- \_ the game start.
- \_ a player plays one move.
- \_ we jump to the next player (opponent's of the previous player).
- \_ ... we loop until one player can't play, then it is the end of the game





Board Game Arena

STUDIO

# Game state & BGA

Game states is a **central piece** in the BGA framework.

Depending on which game state a specific game is right now, the BGA platform is going to :

- Decide which text is to be displayed in the **status bar**.
- Allow some **player actions** (and forbid some others).
- Determine which players are « **active** » (ie : it is their turn).
- Trigger some automatic **game actions** on server (ex : apply some automatic card effect).
- Trigger some **game interface adaptations** on client side (in Javascript).
- ... and of course, determine what are the **next possible game states**.



# Define your game state machine

You can define the game state machine of your game in your **states.inc.php** file.

Your game state machine is basically a PHP associative array. Each entry correspond to a game state of your game, and each entry key correspond to the **game state id**.

Two game states are particular (and shouldn't be edited) :

- **GameSetup** (id = 1) is the game state active when your game is starting.
- **GameEnd** (id=99) is the game state you have to active when the game is over.

```
45
46 $machinestates = array(
47
48     // The initial state. Please do not modify.
49     1 => array(
50         "name" => "gameSetup",
51         "description" => clienttranslate("Game setup"),
52         "type" => "manager",
53         "action" => "stGameSetup",
54         "transitions" => array( "" => 2 )
55     ),
56
57     // Note: ID=2 => your first state
```



Board Game Arena

STUDIO

# Game state types

There are 4 different game state types :

- **Activeplayer**
- **Multipleactiveplayer**
- **Game**
- **Manager**

When the current game state's type is **activeplayer**, only 1 player is active (it is the turn of only one player, and it is the only one allowed to do some player action).

When the current game state's type is **multipleactiveplayer**, several players are active at the same time, and can perform some player actions simultaneously.

When the current game state's type is **game**, no player is active. This game state is used to perform some automatic action on server side (ex : jump to next player). This is an unstable state : you have to jump to another game state automatically when executing the code associated with this type of state.

You don't have to care about « **Manager** » game state type : only « `gameSetup` » and « `gameEnd` » state are using this type.

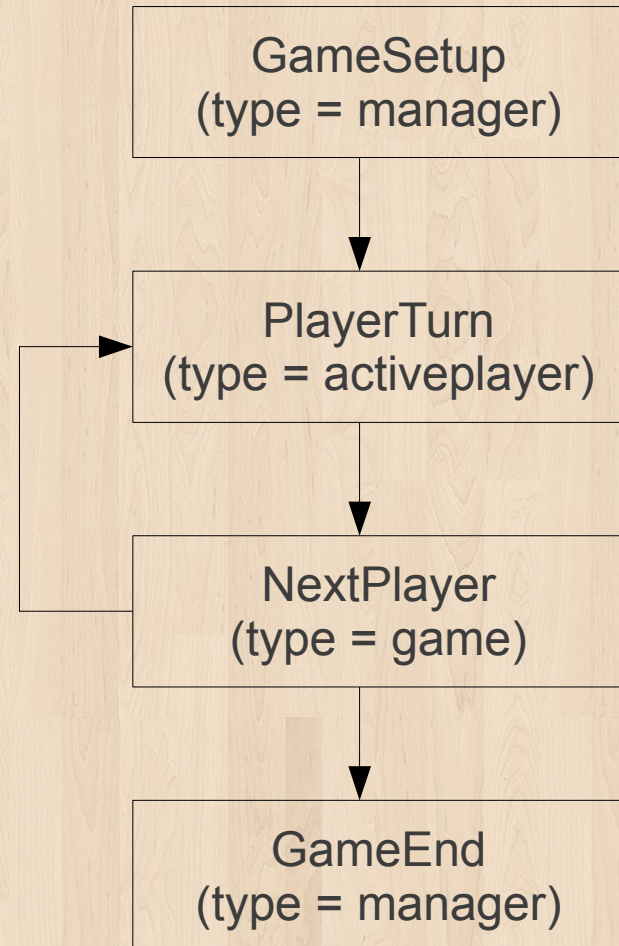


Board Game Arena

STUDIO

# Game state types

Here is the previous example with game state types :





Board Game Arena

STUDIO

# Focus on : Active player

In a board game, most of the time, everyone plays when it's his turn to play. This is why we have in BGA framework the concept of **active player**.

Basically, the active player is the player whose turn it is.

When a player is active, and when the current game state's type is « activeplayer », the following things happened :

- Active player's **icon** in his game panel is changed into a hourglass icon.
- Active player's **time to think** starts decreasing.
- If the active player wasn't active before, the « it's your turn » **sound** is played.



Pay attention : **active player != current player**

In BGA framework, the **current player** is the player who played the current player action (= player who made the AJAX request). Most of the time, the current player is also the active player, but in some context this could be wrong.





# Game state machine example : Reversi

At first, here's the « states.inc.php » file for Reversi, that correspond to the diagram on the right. We're going to explain it now.

```

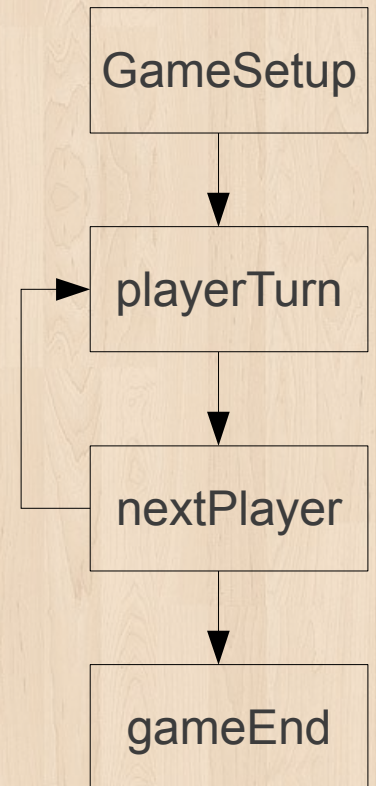
1 => array(
  "name" => "gameSetup",
  "description" => clienttranslate("Game setup"),
  "type" => "manager",
  "action" => "stGameSetup",
  "transitions" => array( "" => 10 )
),

10 => array(
  "name" => "playerTurn",
  "description" => clienttranslate('${actplayer} must play a disc'),
  "descriptionmyturn" => clienttranslate('${you} must play a disc'),
  "type" => "activeplayer",
  "args" => "argPlayerTurn",
  "possibleactions" => array( 'playDisc' ),
  "transitions" => array( "playDisc" => 11, "zombiePass" => 11 )
),

11 => array(
  "name" => "nextPlayer",
  "type" => "game",
  "action" => "stNextPlayer",
  "updateGameProgression" => true,
  "transitions" => array( "nextTurn" => 10, "cantPlay" => 11, "endGame" => 99 )
),

99 => array(
  "name" => "gameEnd",
  "description" => clienttranslate("End of game"),
  "type" => "manager",
  "action" => "stGameEnd",
  "args" => "argGameEnd"
)

```





Board Game Arena

STUDIO

# Game state function 1/6 : transitions

« **Transition** » argument specify « paths » between game states, ie how you can jump from one game state to another.

Example with «nextPlayer» state is Reversi :

```
11 => array(
  "name" => "nextPlayer",
  "type" => "game",
  "action" => "stNextPlayer",
  "updateGameProgression" => true,
  "transitions" => array( "nextTurn" => 10, "cantPlay" => 11, "endGame" => 99 )
),
```

In the example above, if the current game state is « nextPlayer », the next game state could be one of the following : 10 (playerTurn), 11 (nextPlayer, again), or 99 (gameEnd).

The strings used as keys for the transitions argument defines the name of the **transitions**. The transition name is used when you are jumping from one game state to another in your PHP.

Example : if current game state is « nextPlayer » and if you do the following in your PHP :

```
$this->gamestate->nextState( 'nextTurn' );
```

... you are going to use transition « nextTurn », and then jump to game state 10 (=playerTurn) according to the game state definition above.



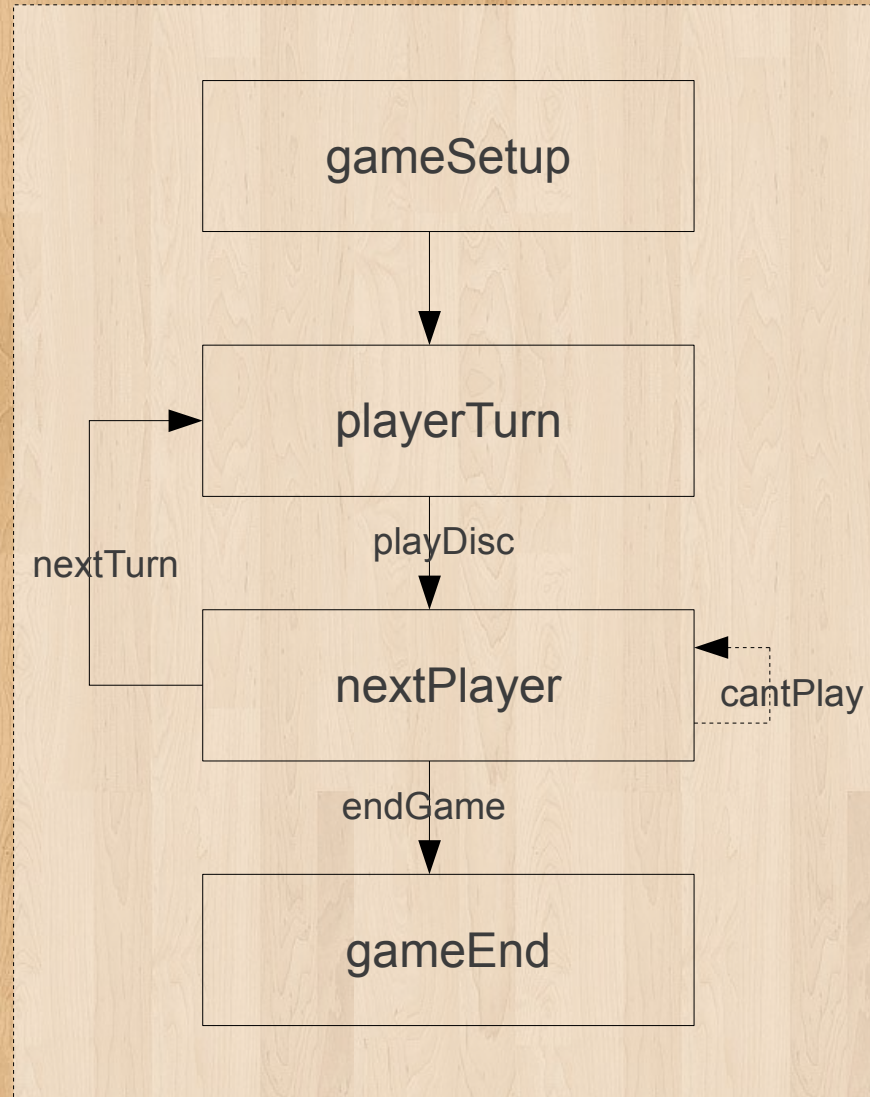
Board Game Arena  
STUDIO

# Game state function 1/6 : transitions

You can see on the right the transitions displayed on the diagram.

If you design your game states and game state transitions carefully, you are going to save a lot of time programming your game.

Once a good state machine is defined for a game, you can start to concentrate yourself on each of the specific state of the game, and let BGA Framework take care of navigation between game states.





Board Game Arena

STUDIO

# Game state function 2/6 : description

When you specify a « **description** » argument in your game state, this description is displayed in the status bar.

Example with « **playerTurn** » state is Reversi :

```
10 => array(  
  "name" => "playerTurn",  
  "description" => clienttranslate('${actplayer} must play a disc'),  
  "descriptionmyturn" => clienttranslate('${you} must play a disc'),  
  "type" => "activeplayer",  
  "args" => "argPlayerTurn",  
  "possibleactions" => array( 'playDisc' ),  
  "transitions" => array( "playDisc" => 11, "zombiePass" => 11 )  
)
```

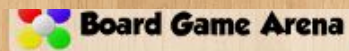


Table n°245624 Coup n° Progression: 50%

**Sourisdudesert must play a disc**





Board Game Arena

STUDIO

# Game state function 2/6 : descriptionmyturn

When you specify a « **descriptionmyturn** » argument in your game state, this description takes over the « description » argument in the status bar when the current player is active.

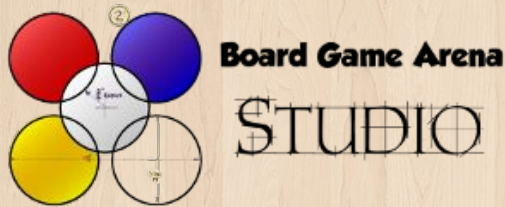
Example with « playerTurn » state is Reversi :

```
10 => array(  
  "name" => "playerTurn",  
  "description" => clienttranslate('${actplayer} must play a disc'),  
  "descriptionmyturn" => clienttranslate('${you} must play a disc'),  
  "type" => "activeplayer",  
  "args" => "argPlayerTurn",  
  "possibleactions" => array( 'playDisc' ),  
  "transitions" => array( "playDisc" => 11, "zombiePass" => 11 )  
)
```

n°245652 Coup n°1 Progression: 0%  
Vous devez réfléchir un peu, merci

4:52

**Vous must play a disc**



# Game state function 3/6 : game interface adaptations

Game states are especially useful on server side to implement the rules of the game, but they are also useful on client side (game interface) too.

3 Javascript methods are directly linked with gamestate :

- « onEnteringState » is called when we jump into a game state.
- « onLeavingState » is called when we are leaving a game state.
- « onUpdateActionButtons » allows you to add some player action button in the status bar depending on current game state.

Example 1 : you can show some `<div>` element (with `dojo.style` method) when entering into a game state, and hide it when leaving.

Example 2 : in Reversi, we are displaying possible move when entering into « playerTurn » game state (we'll tell you more about this example later).

Example 3 : in Dominion, we add a « Buy nothing » button on the status bar when we are in the « buyCard » game state.

# Game state function 4/6 : possibleactions

« **possibleactions** » defines what game actions are possible for the active(s) player(s) during the current game state. Obviously, « possibleactions » argument is specified for activeplayer and multipleactiveplayer game states.

Example with « playerTurn » state is Reversi :

```
10 => array(
  "name" => "playerTurn",
  "description" => clienttranslate('${actplayer} must play a disc'),
  "descriptionmyturn" => clienttranslate('${you} must play a disc'),
  "type" => "activeplayer",
  "args" => "argPlayerTurn",
  ""possibleactions"" => array( 'playDisc' ),
  "transitions" => array( "playDisc" => 11, "zombiePass" => 11 )
),
```

Once you define an action as « possible », it's very easy to check that a player is authorized to do some game action, which is essential to ensure your players can't cheat :

From your PHP code :

```
function playDisc( $x, $y )
{
  // Check that this player is active and that this action is possible at this moment
  self::checkAction( 'playDisc' );
```

From your Javascript code :

```
if( this.checkAction( 'playDisc' ) ) // Check that this action is possible at this moment
{
  this.ajaxcall( "/reversi/reversi/playDisc.html" );
```





Board Game Arena

STUDIO

# Game state function 4/6 : action

«**action**» defines a PHP method to call each time the game state become the current game state. With this method, you can do some automatic actions.

In the following example with the « nextPlayer » state in Reversi, we jump to the next player :

```
11 => array(
    "name" => "nextPlayer",
    "type" => "game",
    ""action" => "stNextPlayer",
    "updateGameProgression" => true,
    "transitions" => array( "nextTurn" => 10, "cantPlay" => 11, "endGame" => 99 )
),
```

In reversi.game.php :

```
function stNextPlayer()
{
    // Active next player
    $player_id = self::activeNextPlayer();
    ...
}
```

A method called with a « action » argument is called « **game state reaction method** ». By convention, game state reaction method are prefixed with « st ».



Board Game Arena

STUDIO

# Game state function 4/6 : action

In Reversi example, the stNextPlayer method :

- Active the next player
- Check if there is some free space on the board. If there is not, it jumps to the gameEnd state.
- Check if some player has no more disc on the board (=> instant victory => gameEnd).
- Check that the current player can play, otherwise change active player and loop on this gamestate (« cantPlay » transition).
- If none of the previous things happened, give extra time to think to active player and go to the « playerTurn » state (« nextTurn » transition).

```
function stNextPlayer()
{
  // Active next player
  $player_id = self::activeNextPlayer();

  // Check if both player has at least 1 discs, and if there are free squares to play
  $player_to_discs = self::getCollectionFromDb( "SELECT board_player, COUNT( board_x )
                                               FROM board
                                               GROUP BY board_player", true );

  if( ! isset( $player_to_discs[ null ] ) )
  {
    // Index 0 has not been set => there's no more free place on the board !
    // => end of the game
    $this->gamestate->nextState( 'endGame' );
    return ;
  }
  else if( ! isset( $player_to_discs[ $player_id ] ) )
  {
    // Active player has no more disc on the board => he loses immediately
    $this->gamestate->nextState( 'endGame' );
    return ;
  }
}
```



Board Game Arena

STUDIO

# Game state function 5/6 : args

From time to time, it happens that you need some information on the client side (ie : for your game interface) only for a specific game state.

Example 1 : for Reversi, the list of possible moves during playerTurn state.

Example 2 : in Caylus, the number of remaining king's favor to choose in the state where the player is choosing a favor.

Example 3 : in Can't stop, the list of possible die combination to be displayed to the active player in order he can choose among them.

In such a situation, you can specify a method name as the « args » argument for your game state. This method must get some piece of information about the game (ex : for Reversi, the possible moves) and return them.

Thus, this data can be transmitted to the clients and used by the clients to display it.

Let's see a complete example using args with « Reversi » game :



Board Game Arena

STUDIO

# Game state function 5/6 : args

In states.inc.php, we specify some « args » argument for gamestate « playerTurn » :

```
10 => array(
    "name" => "placeWorkers",
    "description" => clienttranslate('${actplayer} must place some workers'),
    "descriptionmyturn" => clienttranslate('${you} must place some workers'),
    "type" => "activeplayer",
    "args" => "argPlaceWorkers",
    "possibleactions" => array( "placeWorkers" ),
    "transitions" => array( "nextPlayer" => 11, "nextPhase" => 12, "zombiePass" => 11 )
),
```

It corresponds to a « argPlaceWorkers » method in our PHP code (reversi.game.php):

```
function argPlayerTurn()
{
    return array(
        'possibleMoves' => self::getPossibleMoves()
    );
}
```

Then, when we enter into « playerTurn » game state on the client side, we can highlight the possible moves on the board using information returned by argPlayerTurn :

```
onEnteringState: function( stateName, args )
{
    console.log( 'Entering state: '+stateName );

    switch( stateName )
    {
        case 'playerTurn':
            this.updatePossibleMoves( args.args.possibleMoves );
            break;
    }
}
```



Board Game Arena

STUDIO


# Game state function 6/6 : updateGameProgression

When you specify « updateGameProgression » in a game state, you are telling the BGA framework that the « game progression » indicator must be updated each time we jump into this game state.

Consequently, your « getGameProgression » method will be called each time we jump into this game state.

Example with « nextPlayer » state is Reversi :

```
11 => array(  
  "name" => "nextPlayer",  
  "type" => "game",  
  "action" => "stNextPlayer",  
  "updateGameProgression" => true,  
  "transitions" => array( "nextTurn" => 10, "cantPlay" => 11, "endGame" => 99 )  
)
```



15652 Coup n°4 Progression: 5%  
mais réfléchir un peu, merci

You must play a die.



Board Game Arena

STUDIO

# Game states tips

Specifying a good state machine for your game is **essential** while working with BGA Framework. Once you've designed your state machine and after you've added corresponding « st », « arg » and « player action » methods on your code, your source code on server side is well organized and you are ready to concentrate yourself on more detailed part of the game.

Usually, it's better to start designing a state machine on paper, in order to visualize all the transitions. A good practice is to read the game rules while writing on a paper all possible different player actions and all automatic actions to perform during a game, then to check if it correspond to the state machine.

Defining a new game state is very simple to do : it's just configuration! Consequently, don't hesitate to add as many game states as you need. In some situation, add one or two additional game states can simplify the code of your game and make it more reliable. For example, if there are a lot of operations to be done at the end of a turn with a lot of conditions and branches, it is probably better to split these operations among several « game » type's game state in order to make your source code more clear.



**Board Game Arena**

**STUDIO**

# Summary

- You know how to design and how to specify a **game state machine** for your game.
- You know how to use game states to **structure and organize your source code**, in order you can concentrate yourself on each part.
- You know how to use all game states built-in functionalities to manage active players, status bar and game progression.