

Toddtris - UML Model

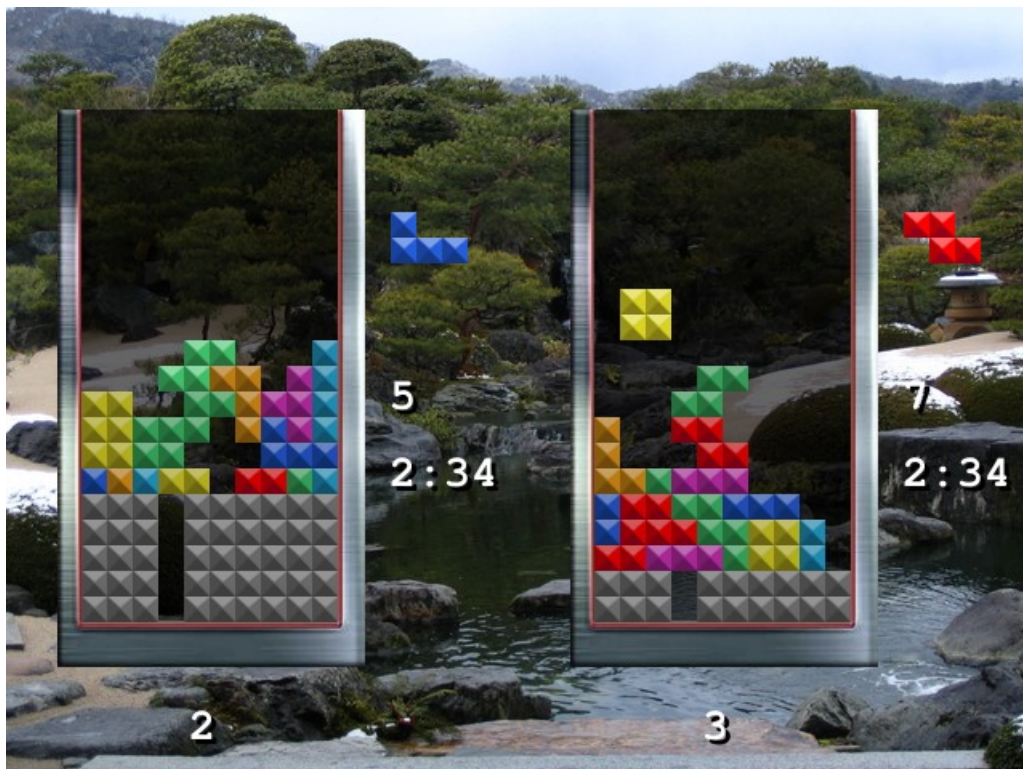


Table of Contents

1 Model.....	3
1.1 Block.....	3
1.2 Well.....	3
1.3 Randomizer.....	4
1.4 KickOffset.....	4
1.5 TwoPlayerGame.....	4
1.6 WellListener / WellListenerList.....	4
2 State Machine.....	4
2.1 Implementation of State Machine.....	5
2.2 AppState.....	5
2.3 AppStateMachine.....	6
2.4 InputReader / InputState.....	7
2.5 main.cpp / init.cpp.....	7
3 Well Controller.....	7
4 Rendering.....	8
4.1 SplashRenderer.....	8
4.2 MenuRenderer.....	8
4.3 OnePlayerRenderer.....	8
4.4 TwoPlayerRenderer.....	8
4.5 WellRenderer.....	8
4.6 Renderer Implementations.....	9
5 Platform Specific Code.....	10
6 Namespaces.....	11

1 Model

The 'model' namespace includes the classes that model the physical state of the game. The model classes have been designed to minimise object creation and memory usage owing to the code's origin in a Java ME game.

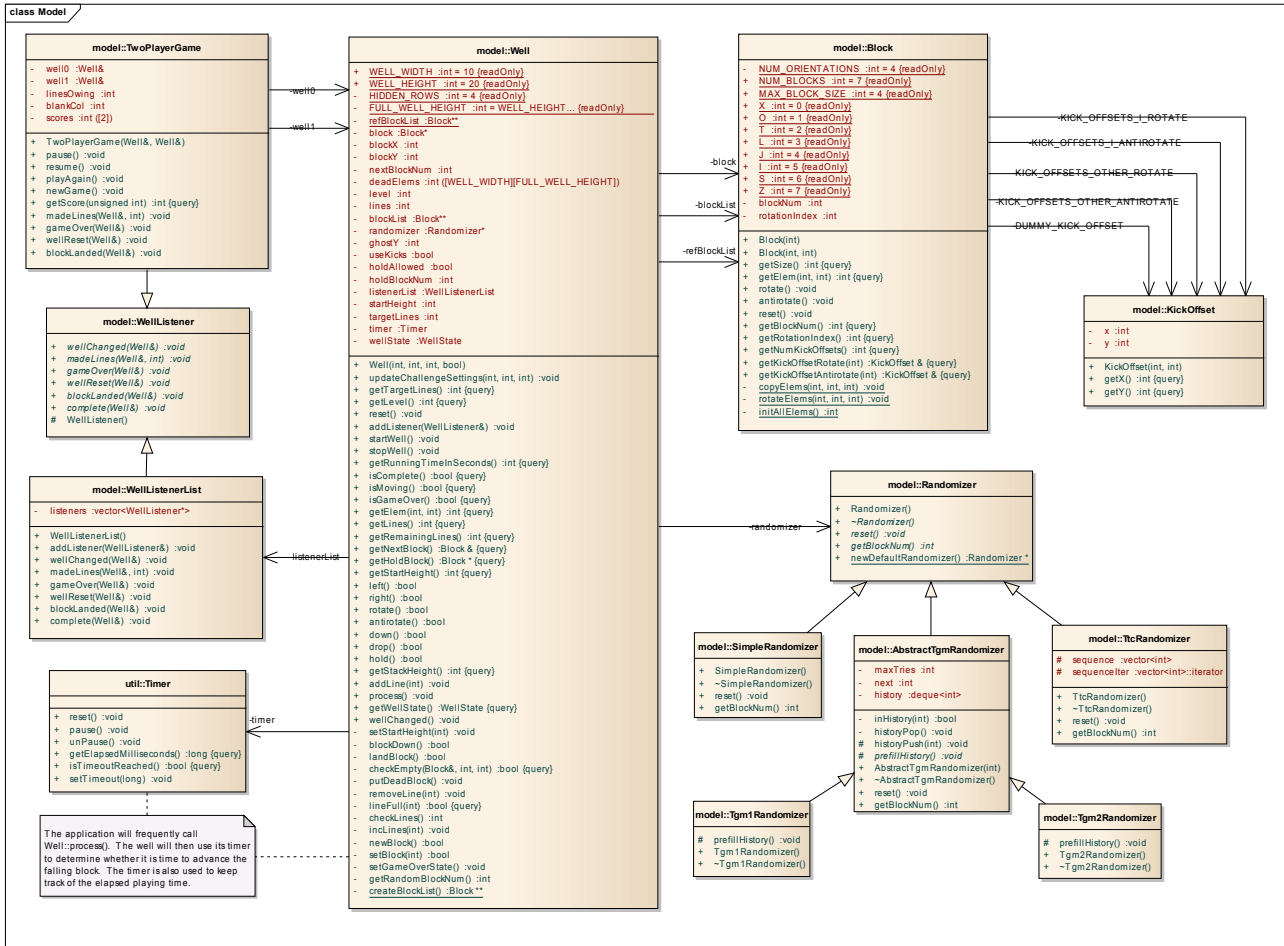


Illustration 1: Toddris model classes

1.1 Block

The Block class represents one of the seven possible game pieces and includes methods to rotate the block. A block is represented as a square composed of 2x2, 3x3 or 4x4 elements. An element is either (a) 0 representing empty space or (b) a non-zero number representing part of the game piece. The 'size' of a block is found using the 'getSize' method and the value of a particular element is found using 'getElem'.

In practice all four possible orientations of each of the seven different blocks are pre-calculated and the state of the Block instance is determined by 'blockNum' denoting which of the seven block types this block represents and 'rotationIndex' denoting its orientation. A 'reset' method is provided to return the block to its original orientation to assist in reusing Block instances.

1.2 Well

'Well' is the most important class in the toddris game. This class represents the well into which pieces are dropped, the blocks that have already landed, the block currently in play, the 'next' block and the 'ghost' block. The 'ghost' block shows the position in which the block would land if

dropped from current position.

Methods are included to rotate and move the in-play block, display or hide the 'ghost' block, add lines to the bottom of the Well for use in two player games and to store or use the 'hold' block (note that the hold block facility is disabled by not being connected to any input as the author does not like this facility).

The arrangement of elements in the well is implemented by using a two dimensional array to denoting the 'landed' elements and the (x, y) coordinates of the in-play block. The 'getElem' method overlays the in-play block onto the well to give a snapshot view of the state of the current state of the well.

Description of element	Calculated as	Possible values
Empty space	0	0
Block in play	Block Number	1 – 7
Landed block	0 – (Block Number)	(-1) – (-7)
'Dead element' (added during two player game)	Block::NUM_BLOCKS + 1	8
Ghost element	Block Number + Block::NUM_BLOCKS + 1	9 – 15

1.3 Randomizer

Randomizer is a pure virtual class is used by the Well class to select blocks at random. Several different algorithms are implemented.

1.4 KickOffset

The KickOffset class is used in the implementation of 'wall kicks'. If an attempt to rotate a block fails (i.e. the rotated block does not fit at its current coordinates) then a series of small displacements of the block are attempted. For more details see <http://harddrop.com/wiki/SRS> .

1.5 TwoPlayerGame

The TwoPlayerGame class is used to model a two player battle. Lines made by each player are tracked and result in lines being added to the bottom of the opposing player's well. The number of games won by each player is also tracked.

1.6 WellListener / WellListenerList

WellListener is a pure virtual class which may be extended by classes wishing to be notified of changes to the state of a well through various callback methods. For example the TwoPlayerGame class extends WellListener so that it can be notified of lines made by each player adding lines to the other player's well in response. The Well::addListener method is used to register listeners.

WellListenerList the class used by Well to keep track of registered listeners.

2 State Machine

A state machine pattern is used to handle the different logical states the game may be – in menu system, playing one player game, playing two player game, paused, game over, etc. Depending on the current state input will be handled differently and may cause transition to another state. The states of the application are shown below.

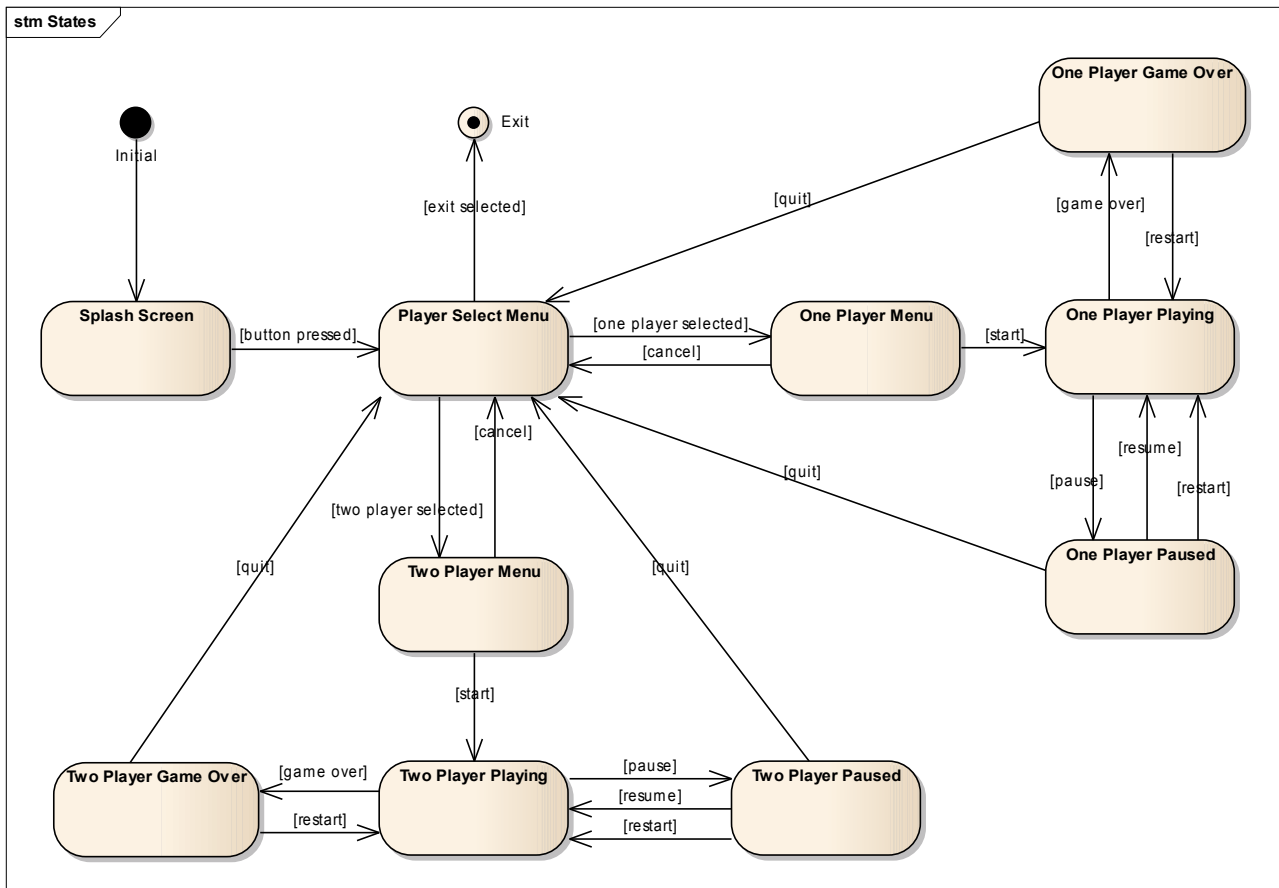


Illustration 2: Toddtris application states

2.1 Implementation of State Machine

The StateMachine is implemented by classes in the “app” namespace.

2.2 AppState

The AppState class is the superclass for all of the games' states. Empty virtual methods exist which are invoked by the state machine framework in response to button presses from players. Any state interested in a particular button press need simply override the relevant method to receive callbacks. A virtual 'prepare' method may also be overridden by a state to receive a callback that this state has been entered so that it might carry out any required set up; for example when a paused state is entered it will stop the timer used by the well to automatically drop blocks.

As a simple approximation to multi-threading a 'process' method may be overridden to perform any background processing and will be called frequently by the state machine framework. Any implementation of 'process' should quickly return control to the framework by simply returning from this method. In practice this method is overridden in states where the game is in progress to allow blocks to fall by checking a timer and moving the blocks down when required.

A pure virtual 'draw' method must be implemented by all states. This method is responsible for rendering the current state on screen and will be invoked by the state machine framework.

When an event occurs giving raise to a state transition the state will notify the state machine framework via a callback to the AppStateMachine class (see next section). The framework will then be responsible for handling the transition. For example, when the player starts a new one player game, the OnePlayerMenuState class will trigger a state transition by calling

AppStateMachine::startOnePlayerGame.

2.3 AppStateMachine

AppStateMachine is responsible for reading the input state (i.e. button presses) and calling the relevant methods on the current state in response to those button presses. AppStateMachine will also call the 'process' method on each state after reading the input as described in the previous section and then request that the state render itself. This process is repeated in an loop until the 'exitApp' method is called. A number of methods are provided which the current state may use to invoke state transitions.

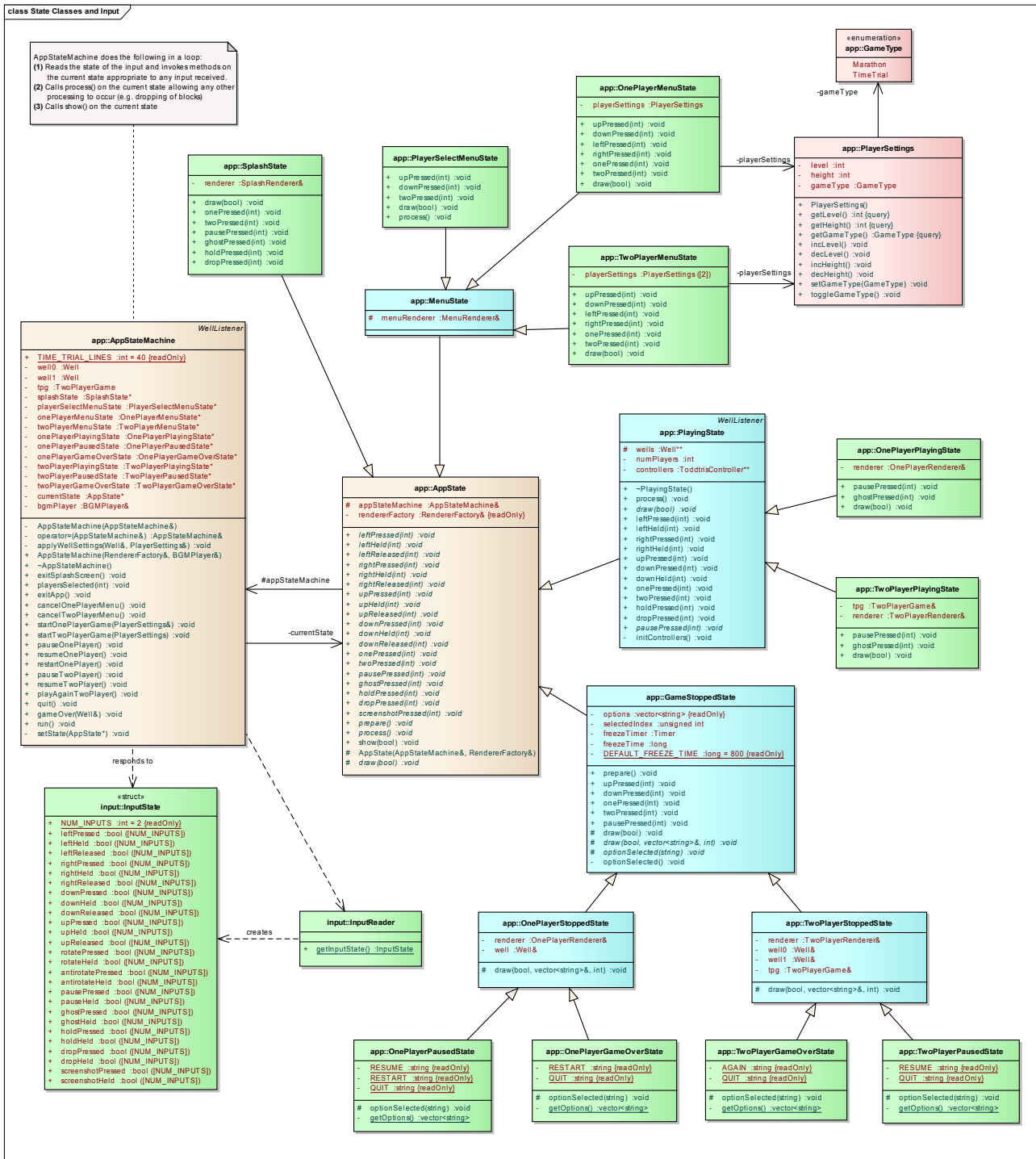


Illustration 3: Toddtris state machine implementation

2.4 InputReader / InputState

These classes are used by AppStateMachine to read button presses. Different implementations exist for different platforms.

2.5 main.cpp / init.cpp

main.cpp provides the 'main' method for the game. It simply does some initialisation and then hands control to the state machine. The initialisation is handled by the 'init' method in init.cpp of which different versions exist depending on the (a) the platform and (b) the renderer to be used (text-based or graphical).

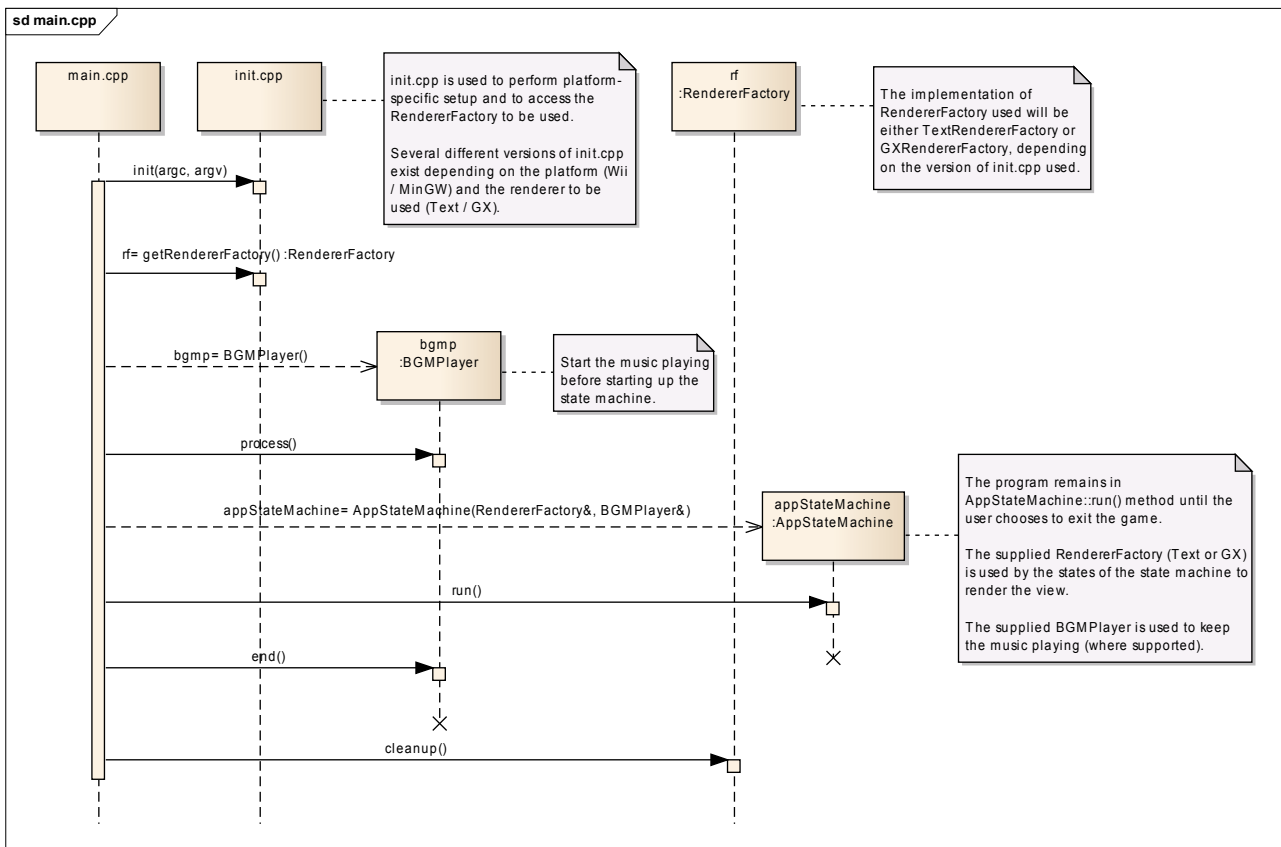


Illustration 4: Outline of Toddtris state machine process

3 Well Controller

The WellController class is used when the game is in a playing state to manipulate the players' wells.

A WellController is created for each well and the PlayingState class delegates button presses to this. WellController will in turn delegate left, right and down button presses to 'repeater' classes. The repeater classes are used to repeat a movement when the player holds a button down.

The WellController and repeaters further delegate to an ActionHandler subclass which in turn manipulates the well. The point of the ActionHandler is that there is StillActionHandler for use when the game is paused and a MovingActionHandler for use when the game is in progress. In practice this is not really required; In the current code base WellController is only ever used when the game is in a 'playing' state.

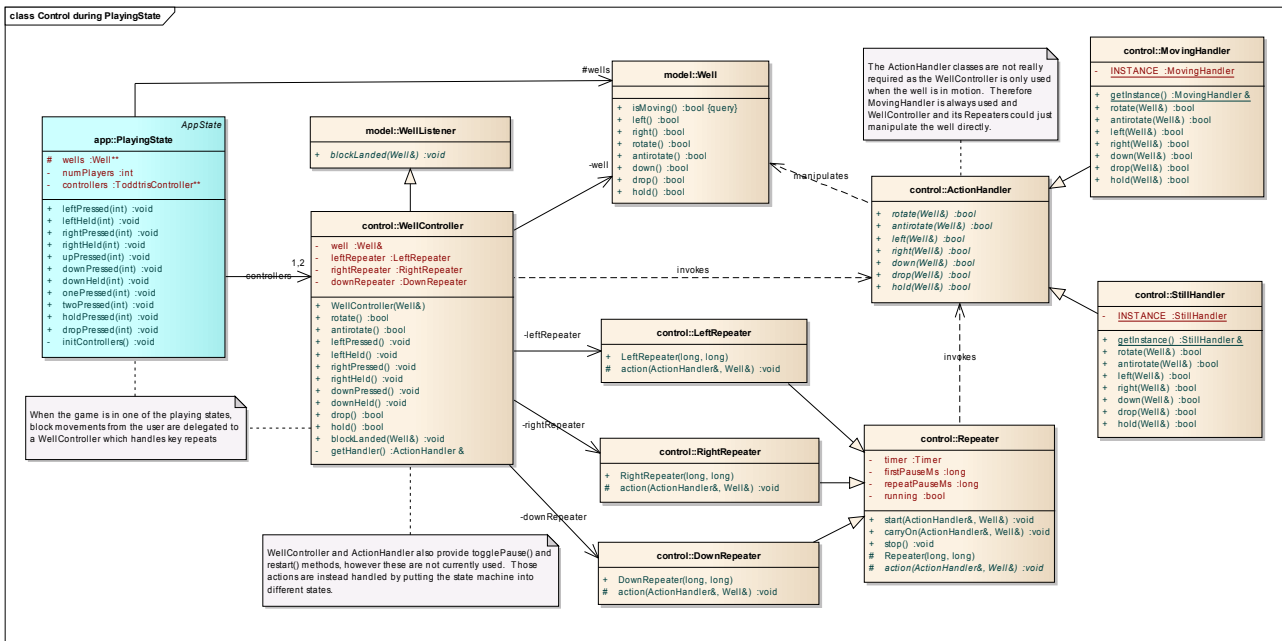


Illustration 5: Toddtris WellController class

4 Rendering

Rendering of the game on screen is handled by various 'renderer' classes which are called by the state classes (Section 2) to render the current state on screen. The state classes deal with abstract renderer classes. Both text-based and graphical implementations of the renderers are available which may be swapped in by simply changing the `RendererFactory` implementation which is passed to the state machine on startup.

4.1 SplashRenderer

Superclass for simple renderers which are required simply to render the splash screen which is shown when the game starts.

4.2 MenuRenderer

MenuRenderer is used by the various menu states to render menu screens with highlighting of selected options.

4.3 OnePlayerRenderer

Used to draw the screen for a one player game that may be in the in-play, paused or game over state.

4.4 TwoPlayerRenderer

Used to draw the screen for a two player game that may be in the in-play, paused or game over state.

4.5 WellRenderer

WellRenderer is used by `OnePlayerRenderer` and `TwoPlayerRenderer` above to render wells and associated detail on screen including the next block, time, and lines made.

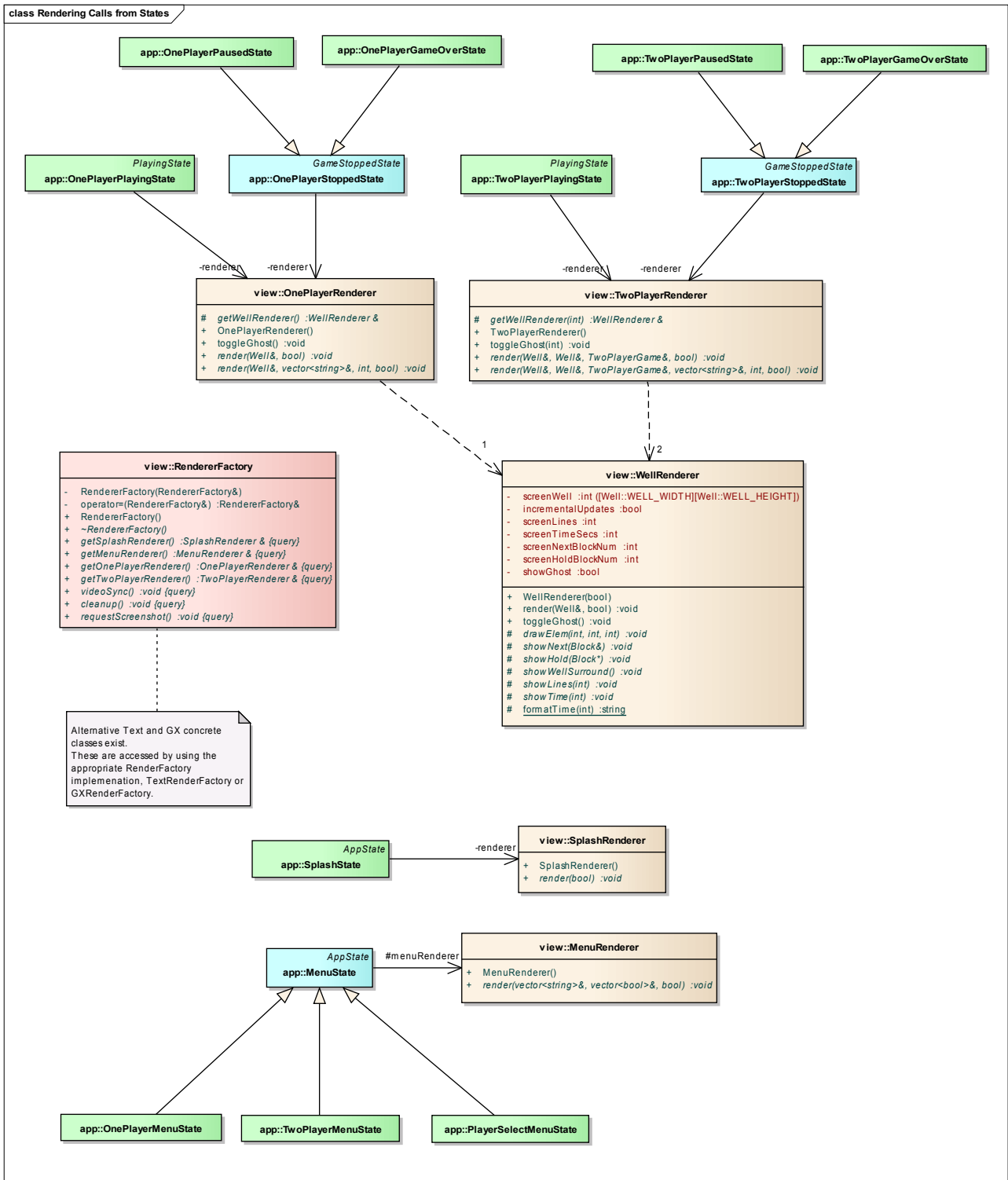


Illustration 6: Use of renderers by state classes

4.6 Renderer Implementations

Two implementations of the renderer classes are available – text-based renderers or graphical renderers. The graphical renderers are available only for the Wii version of the game. Which renderers are used is determined simply by the implementation of RenderFactory which is passed to the state machine and used by the state classes to access concrete implementations of the abstract renderer classes defined above.

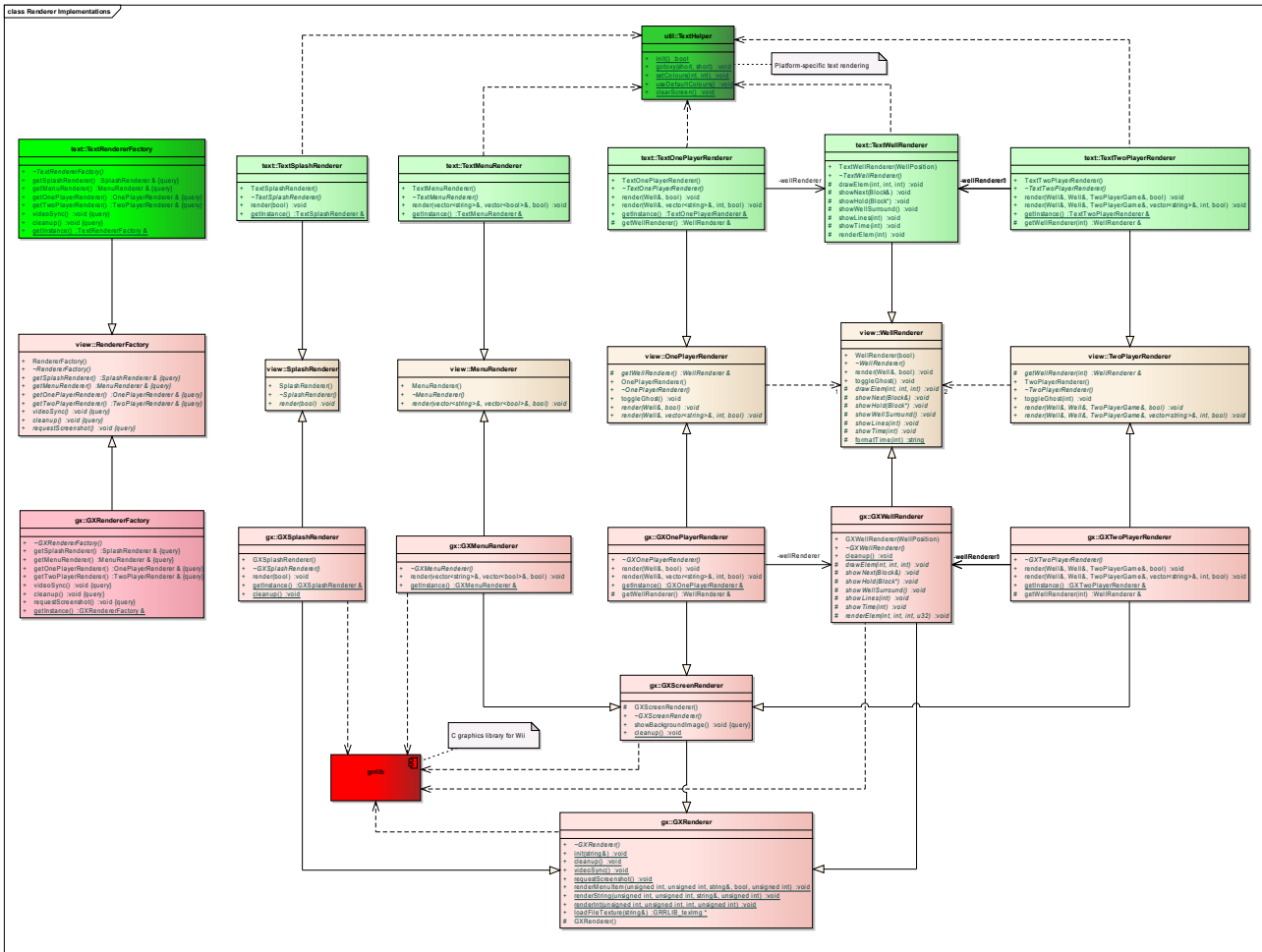


Illustration 7: Text based vs graphical implementation of renderer classes

5 Platform Specific Code

The method of reading input, rendering text on screen and determining the current system time varies between platforms. These functions are performed via helper classes which are implemented differently according to the platform. The affected classes are shown below. Note that the `BGMPlayer` implementation for `MinGW` is simply a dummy implementation and will not actually play the background music.

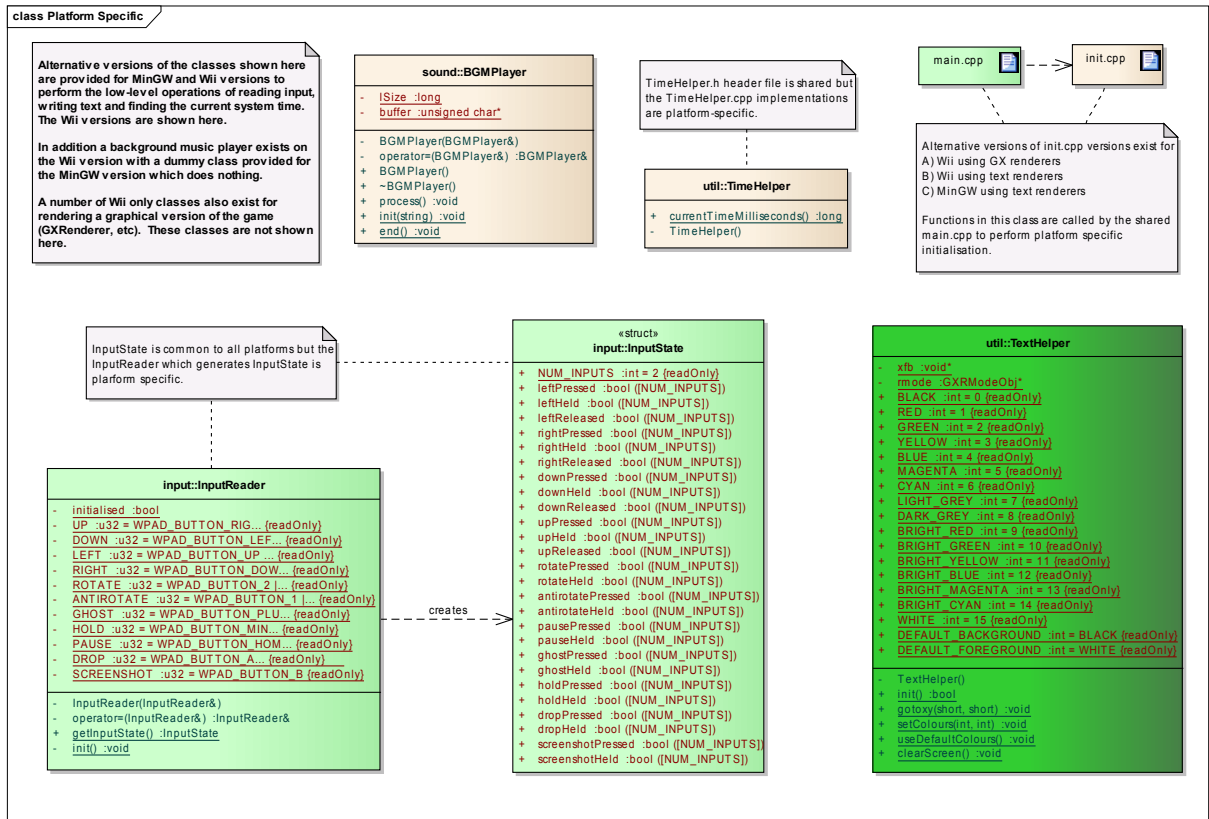


Illustration 8: Platform specific classes

6 Namespaces

Illustration 9 below shows the namespaces into which Toddtris' classes are organised and the relationships between them. Only those classes which are conceptually public (i.e. used by classes in other namespaces) are shown.

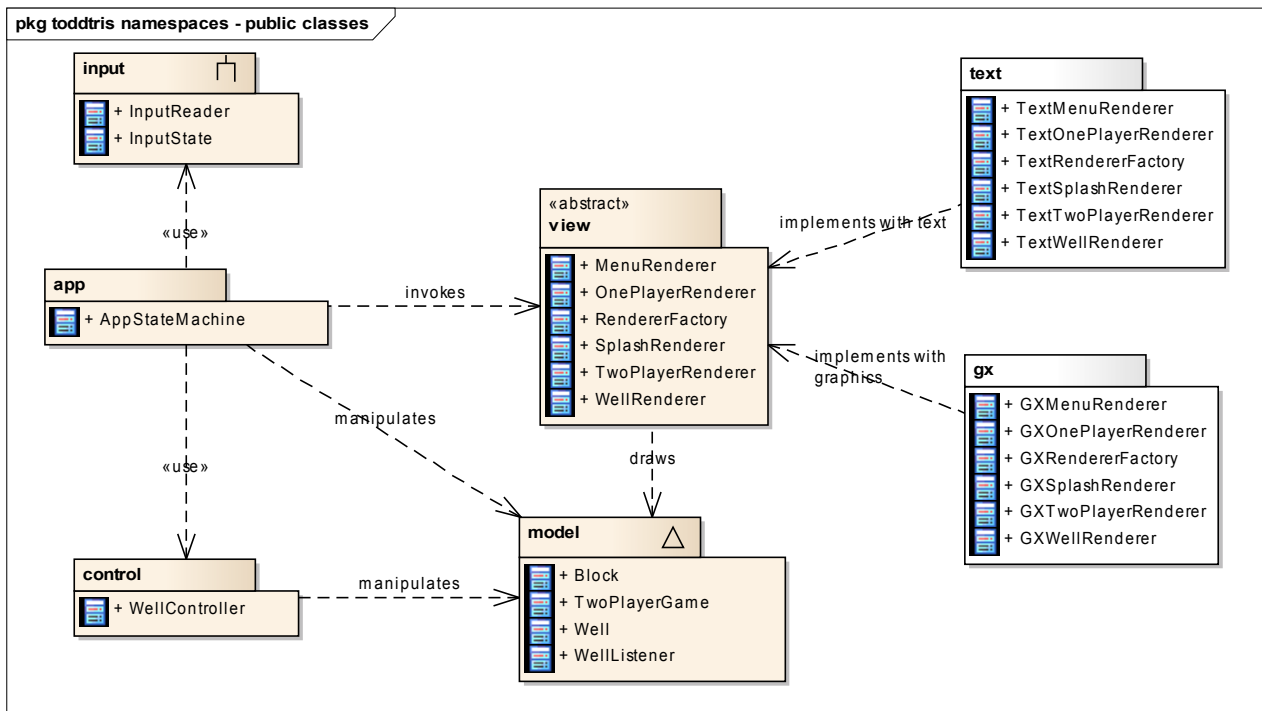


Illustration 9: Toddtris namespaces and 'public' classes

Illustration 10 Shows the the Toddtris namespaces with all classes. The 'util' namespace is also included here; this namespace contains utility classes to find the current system time, implement a timer facility and handle paths to files.

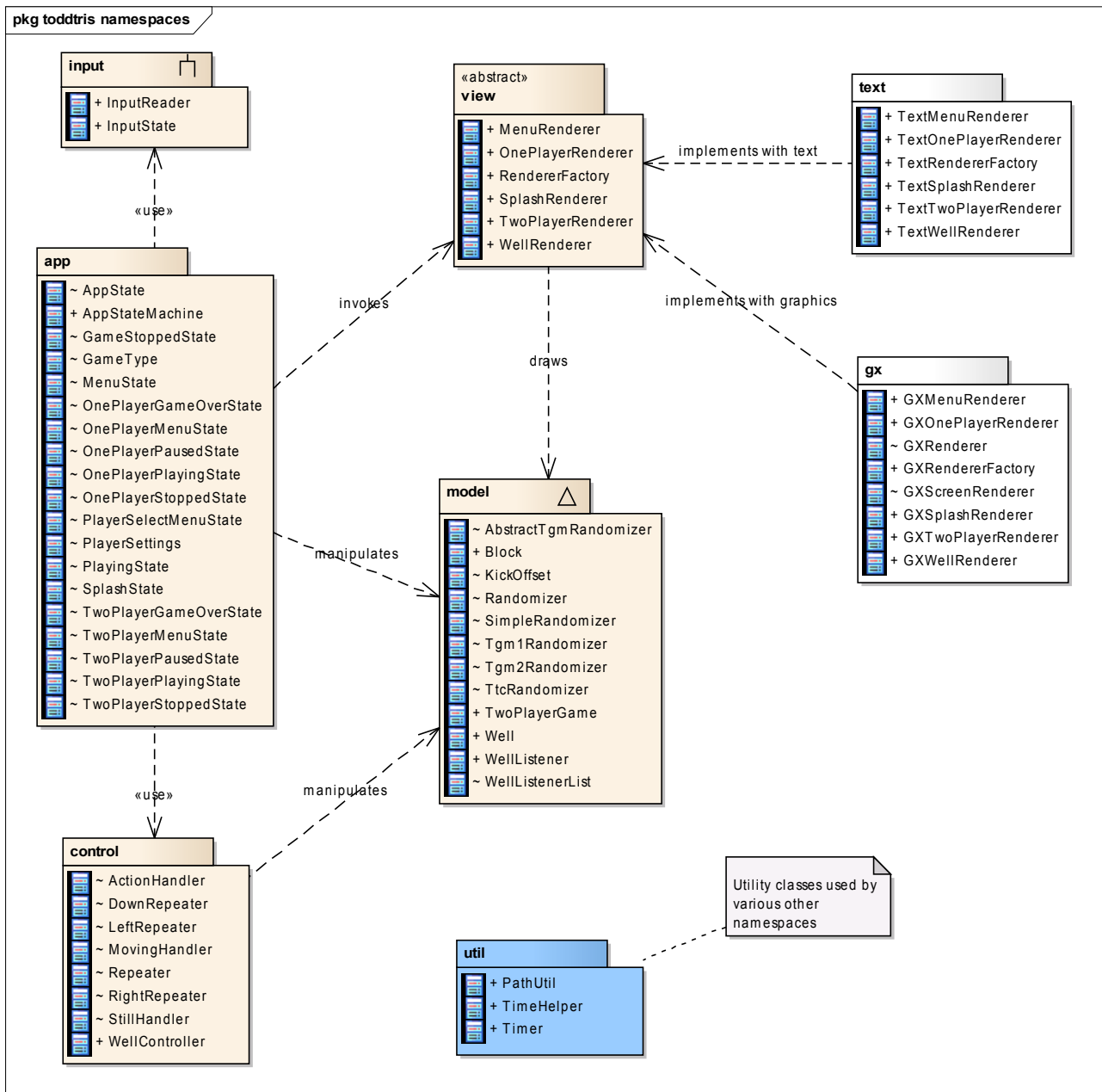


Illustration 10: Toddtris namespaces showing all classes