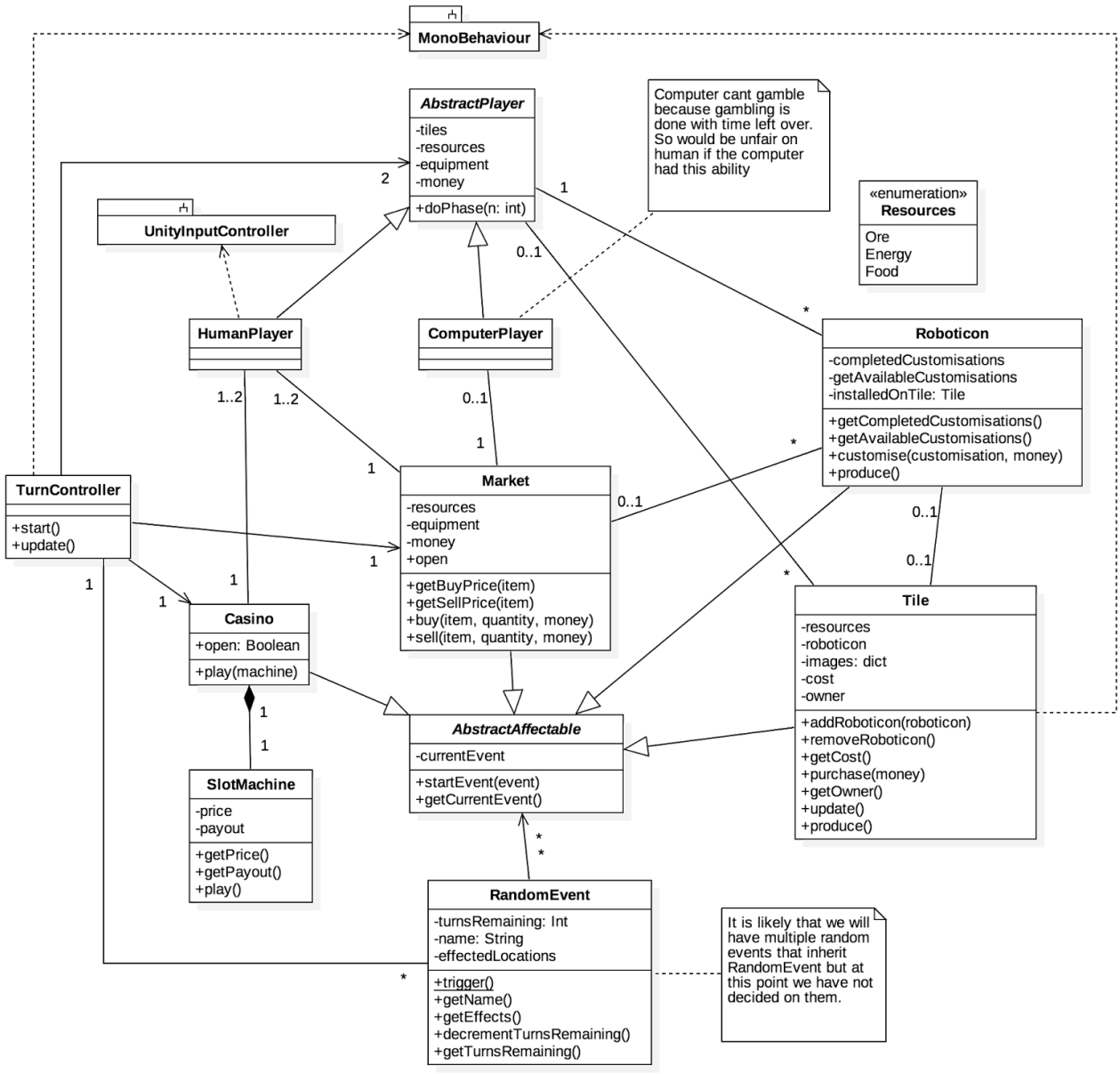
Architecture

Open larger image - Figure 1

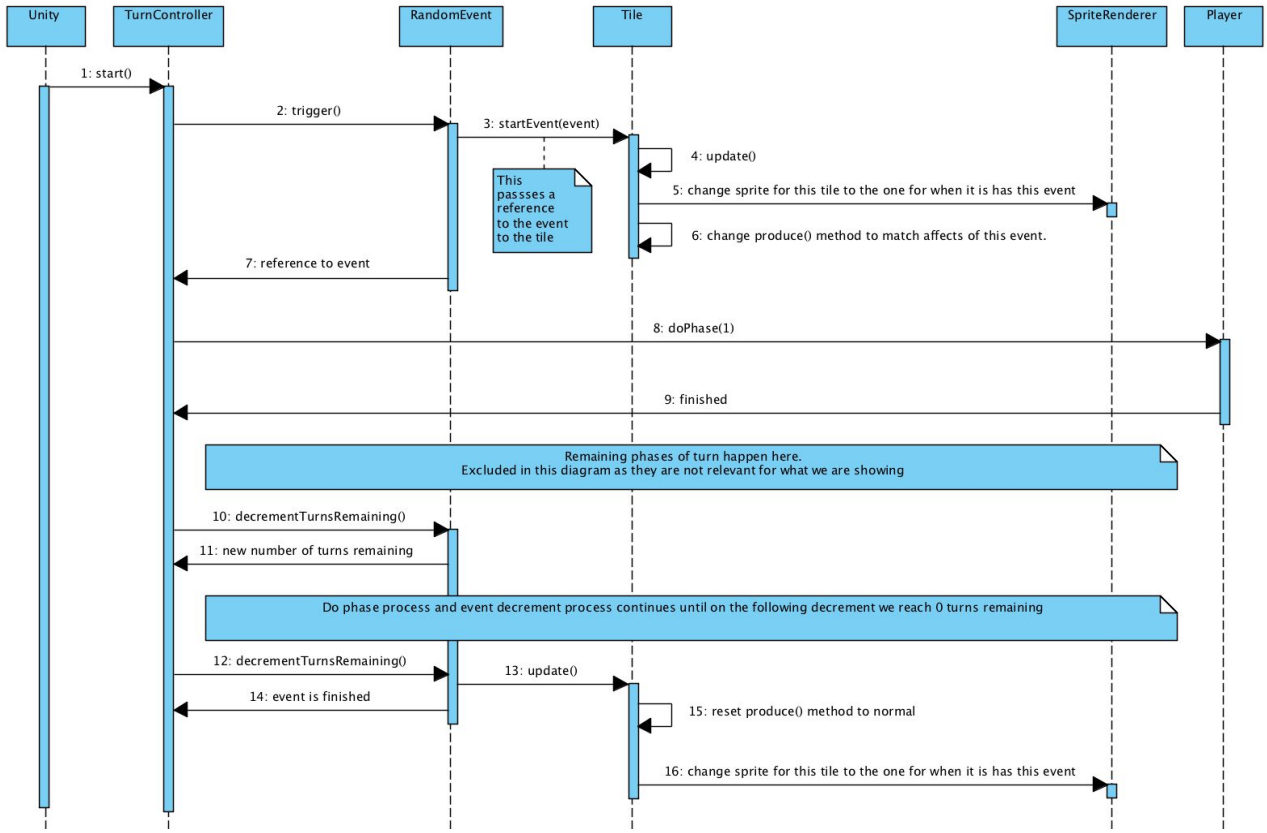


Introduction to architecture

Figure 1 is a UML Class diagram of our proposed architecture. This is a conceptual model with many details that still need to be developed, omitted. In the RandomEvent class, trigger() is underlined to show that it is a static method (the object does not need to be instantiated to call the method).
There are also several directed associations, for example from TurnController to AbstractPlayer. This is to show that TurnController is interacting with abstract player, but the communication is one way. AbstractPlayer does not need to be concerned with the existence of TurnController. Dependencies are shown with a dotted arrow.

We decided to create the diagram with StarUML[1] because it is available for Windows, Mac and Linux allowing anyone in our team to use it on their system. StarUML[1] also has a built in error checker which checks your diagram for simple errors such as duplicate and redundant associations. StarUML[1] allows faster development of diagrams than Photoshop or other image editing software for example. Being efficient with our time is important in software engineering to stay on schedule.

Open larger image - Figure 2



We created the sequence diagram shown in Figure 2 primarily to test our conceptual model of our architecture and how it fits into the Unity environment. A sequence diagram allows you to test against other diagrams by checking required behaviour is actually possible and is sensible. This was very useful for us and resulted in us making some changes to our UML Class Diagram. The sequence diagram also helped us check we understand how our model fits into the Unity platform. The diagram shows how a random event being triggered on a tile changes the sprite for that tile.

Creating this diagram has also exposed where things can happen in parallel. This will help us with the game's performance when it comes to implementation. Wherever there is no need for a response message we can call things asynchronously or in another thread.

Certain periods have been excluded from the diagram because they are not relevant to what we were trying to achieve with the diagram. Only the first phase of one turn is shown to demonstrate how this works but as the remaining phases of each turn would take up too much room on the diagram and add little value we have excluded these.

To create the UML Sequence diagram we used VisualParadigm[2] because we found StarUML frustrating and a number of fellow students recommended VisualParadigm[2] especially for sequence diagrams.

Justification of architecture

We have used notes to include additional information which explain why certain things appear as they do, for example why we only have one random event in the diagram despite the requirement for many different types of random events.

UnityInputController is a subsystem which the Human player depends on. We have included this to show that we have considered how we will interact with the user, but that we are not going to implement this ourselves.

We could have used fewer classes to achieve the same result, but designing the architecture with a larger level of abstraction makes it much easier to test and debug the game when it comes to implementation. Designing it this way also makes the architecture more flexible so that it can cope with changing requirements, or constraints we have not yet thought of, due to our use of object oriented programming and encapsulation.

Classes contain data only about themselves and keep most of it private, only allowing external classes to access its members when it is necessary, via the use of getters/setters. Take the Tile class for example, cost of a tile is persistent for the entire game, it makes no sense for outside classes to be able to modify it, but they should be able to access it. We use a getter for this purpose, in this case GetCost, which returns the cost but abstracts the actual variable that contains the cost away so that outside classes cannot edit it.

Many classes depend on inheritance, sometimes from abstract base classes, such as the player classes which only contain the logic needed to make turns, whilst most of the core player functionality is derived from the abstract player base class. This also means that we can change how the AI or human players take their turns, with no effect on the other class. This is particularly useful for implementing and testing the classes.

Because of the way Unity works, we had to adapt our architecture to suit the engine. For an object to exist in a Unity scene, it has to inherit from the base MonoBehavior class, which unity provides. This class contains Unity specific information, such as world position. This means that classes such as TurnController and Tile will have to derive from this class, as they are present in the game scene. TurnController has to be present in the game scene as Unity requires the script to be attached to an object for the code to run. It is common in unity game development to have an invisible level controller somewhere in the scene. TurnController will be able to access all our other classes and scripts like you would in any other program once it's in the scene.

We use the TurnController class to control all of the progress of a turn and make sure all parts of the game are in the correct state for that turn. All of the initialisation and instance setup is done via the Start function, then the Update function is called, which is a continuous loop that persists until the play session has ended.

The multiplicities in the diagram work for both 2 human players against each other and 1 human player against a computer. They also provide useful information for example, that a tile cannot belong to more than one player at a time.

We have designed the architecture so that all classes are highly cohesive, and lightly coupled. An example of this is where we have directed associations, this means that there is no unnecessary association in one direction when its not needed. Keeping our classes cohesive, for example separating the Casino from the SlotMachine will make future changes easier. If the casino class is separate from the SlotMachine we can make changes or additions to each one without affecting the other. This design will also be easy to write unit and Integration test suites for.

[1] StarUML. “‘StarUML 2. A sophisticated software modeler.” staruml.io. [Online]. Available: http://staruml.io/ [Accessed Nov. 7 2016]
[2] VisualParadigm. “Visual Paradigm Software Design Tool.” visual-paradigm.com [Online]. Available: https://www.visual-paradigm.com/ [Accessed: Nov. 7 2016]