Introduction:

For any significant software project it is important that the developers adhere to some sort of structure to ensure not only the success of their project, but also to make the software engineering process smoother. After all, any novice programmer can take some client's requirements and attempt to make some presentable program out of them, but the simplistic process of coding and fixing is inefficient and just won't produce the quality results of a well designed piece of software with an well organized team behind it.
Menus:

With the basic requirements of our Portal game previously defined, now begins the ever important design phase of the waterfall model. In this phase important decisions are made and careful considerations are taken about how best to implement the features we need. A good idea is probably to start with the simple and move on to the complex later. Now, the first thing presented to the user is a main menu with several options for them to choose from; in order to easily and naturally process the user’s choice, each option can be simply be a button. As the user moves the mouse over a button it would also be nice to highlight that button before the user clicks. Once a button has been pressed in the main menu, the state of the game may change in a manner appropriate to which button was pressed. Also, each menu should have a back button to return to the previous menu if there was one.
Pressing the exit button will end the game, which should be accomplished easily by a return statement once this condition is checked. If the Play button was pressed, the game will enter the most complicated Play State which will be discussed later. The level select button will take a player to another menu where specific levels may be chosen before entering the Play State; this Level Select State will be controlled by buttons similarly to how the Main Menu state is though each different level button will start the Play State at a particular level. The final button from the main menu is the high scores button and will take the user to a screen that displays the high scores until the user hits a button to return to the main menu. This state is not much more complicated
than the menu states, but will require different functionality; after some players have beaten the

game and achieved some scores those scores must be sent to an external source that will compare
and store them, and so this state must access that external source and display the topmost scores in
the list through a table of some sort.

Considering the Play State:

Now that the basic navigation between different states and how they will function has been
covered, the focus now shifts to the Play State, where the most design issues exist. Many unique
events can occur during the Play State, so one good way to identify them is by reviewing the
requirements and highlighting the nouns; this is particularly useful for object-oriented analysis and
design as the nouns often become classes and objects once coding starts.

From the Requirements section:

“Similarly to Portal we will implement portals which can be used to transport matter in an instant. This allows for creative puzzle solving as well as challenges which involve physics in order to succeed. We will allow for conservation of movement in velocity to allow objects and the player to exit the portal at the same speed as he came in. The objective of the game will be to finish the level using skills in platforming and puzzle solving. The player will be able to use portals at will to transport him/herself in order to successfully beat the challenges present in each level.”

The most important nouns that come up in this section are portal, physics, velocity, objects, and player. These nouns are important (they will all likely be either an object or a function) and will be considered from multiple perspectives during this design process.

Creating objects:

For simplicity’s sake, physical objects in the game should mostly be quads so it will be easier to place them precisely in the game and perform collision detection. Three essential quad objects in an level must be wall/floor tiles, the stage exit, and the player character. One of the first things that is done in the process of loading a level is to place all the objects in a certain position; to keep track of every thing's initial position and to make level design simpler, the initial position
of objects can be stored in a grid-like array.

By having the levels created and stored in arrays, it also makes it easier to support user created levels later on if there's time.

**Basic movement:**

Gameplay is centered around the player character which is controlled by the user. There are two basic ways in which the user can move the player: horizontal movement and jumping. It should be fairly straightforward to detect left and right key input and appropriately change the x position of the player quad. Also, when the player attempts to move horizontally it is preferable that the sprite for the player will change to suggest a walking motion. An easy way to accomplish
this would probably be to have an array of sprite images that each represent a different phase of
the walking animation cycle through whenever the player is holding a left or right key on the
keyboard.

**Collision detection:**

The very next thing that should be done once the player has the ability to move is to
implement collision detection to ensure that the player can't go beyond the scope of the level.
Since stationary objects will not move from their initial position from the start of the level, any
moving object can detect collision with a stationary object by checking its current position in the
grid against the cells around it to see if it is about to enter an occupied cell; so basically any
moving object would have free range over the connected empty cells in the grid. Another useful
feature of collision detection would be to know where collision has occurred; the function that
detects collision could ideally return a value that will represent which side of an object has
collided. Another consideration for detecting collision on the player is to possibly set up smaller
hit-boxes to represent collision on a specific parts of the player rather than having one quad
represent the player; since the player will not simply be a square, setting up these smaller hit-boxes
will make collision appear more meaningful as there are less unnecessary gaps between the smaller
boxes of the player's body parts than there would be with one large box.
Detecting collision between moving objects will be slightly different from detecting it between a stationary object and a moving object; this type of collision might be detected by assigning any empty cell a value when a moving object is within its bounds, then if a moving object moves into an empty cell that is marked as having another moving object in it a more precise calculation will be done to see if the two moving objects will collide.

**Gravity and jumping:**

Gravity is another issue relevant to player movement; whenever a player walks off an edge it should fall downwards rather than float. The most apparent way to implement some form of gravity is to impose a downward momentum whenever a player is not colliding with a stationary object on the bottom side (essentially whenever the player is not already touching the ground).
Momentum shouldn’t be too complicated; similar to how horizontal input changes the player’s x position this gravity would just alter the y position in a downward manner. Also, the downward momentum of gravity will increase over the time during which a player is not touching the ground; the longer a player has to fall, the faster they will go (to a certain limit). Jumping will work with gravity. When the user hits the jump key an upward momentum greater than that of gravity will be applied to the player. The player's y position will be the sum of this constant positive y value and the increasing value of gravity; the result will be that as a player jumps the initial upward change will be quick, but as gravity's influence grows that upward momentum will slow and eventually turn to downwards momentum that increases until the player comes in contact with a floor again.

**Portals:**

The next important design issue is determining how portals will be implemented. The player will be holding a portal gun and so it would look nice to have the arms rotate with respect to the mouse in such a way that the player is aiming the portal gun wherever the user moves the mouse. Once a mouse button is clicked, a projectile will be created from the portal gun and move in the direction of where the mouse was clicked; initially this projectile will move in a straight line towards its destination, but eventually we might like to change the path of the projectile to an arc.
which would require a different calculation for determining its path. The portal projectile will continue along its path until it collides with something and if that something it collides with is a valid portal surface, a portal will be placed on that surface. Each button of the mouse will fire a different colored portal and once two portals have been placed on two surfaces, the link between them will be made and they will open. When an object moves through an open portal (when an object collides with the center of a portal) it will appear coming out of the other portal. Each portal can be on a wall, floor, or ceiling so when an object moves through a portal that movement should be preserved between portals and if necessary, converted from horizontal to vertical or vice-versa. For example, if a player falls into a portal on the floor and the other portal is on a wall, the player's downward momentum will be converted to horizontal momentum and it will shoot out of the wall at the same speed it fell into the portal.

**Buttons and boxes:**

One more facet of the game that will use the methods already discussed is the implementation of buttons and boxes (companion cubes). The player will be able to pick up boxes if they are detected to be within range of the box and once a box is picked up it will travel with the player until dropped; this will be done first by detecting collision to determine if there is any box nearby to be picked up and second by simply matching the box's x and y positions with those
of the player. When a box is dropped gravity will act on it in a similar way to which it acts on
the player. The player can also trigger buttons by either standing on them or placing a box on
them. When a button detects collision with another object on top of it it will cause a change
somewhere in the field of play (perhaps opening the stage exit) and that change will remain active
as long as the button detects collision.

Score and ending the level:

During the course of a level a timer will run keeping track of how much time has passed,
and perhaps there will be a counter keeping track of the number of deaths a player has as well.
The timer will constantly increment until the player collides with an open stage exit at which point
the score is updated with the time and number of deaths.

Sounds and images:

During the course of the game, and perhaps even during the menus, there could be some
music that plays and sound effects that occur at relevant times. All sounds will be implemented
with the fmod library. Hopefully the fmod library is pretty straightforward so the sound files can
simply be stored in a directory of some sort and accessed when necessary (when certain actions
such as firing the portal gun are performed). Simple squares moving around the screen aren't very
appealing so to display more exciting textures bitmap images will be bound to certain objects when
necessary. Just like sound files these .bmp images will be stored in the directory until they need to be bound to an object.

**Pausing:**

The final aspect of the Play State will be transitioning to the Pause State. The Pause State can only be accessed during the Play State via a pause key (likely ’p’) and allows access to some of the other states through buttons; it will essentially be another menu, the pause menu. A unique thing of the pause menu though is that it does not irrevocably interrupt the Play State; when the game is paused the timer needs to stop, all input relevant to the Play State will be blocked, and ideally all currently moving objects will halt. Stopping the timer and block input can be easily done by creating a pause flag and blocking everything once the pause flag is set. Stopping currently moving objects might be more difficult however. A way to deal with this might be to store all movable objects on the field of play into an array along with values representing their current velocity. Once the pause flag is detected, all movement in any direction for these objects will be set to zero; then when the pause state ends all the movement can be restored by reassigning the values associated with each object from what was stored in the array at the time of pause.

**Conclusion:**

As is apparent, even for a simple platforming game with only basic features there is quite a
lot that can be considered during the design phase of Software Engineering. Even for deceptively simple projects it is useful and important to include some sort of design phase that considers all the functionality a program will eventually need and how to go about implementing it. With all of the important design issues sorted out, as the focus now shifts to the coding phase there is much less ambiguity as to how each component will be implemented. Meanwhile, the novice programmer is only just realizing that it might be a difficult issue to go back and restructure his/her initialization of objects in order to make collision detection work properly.