A note on architecture - by Hans Wichman

A lot of programming comes down to dividing complex problems into smaller problems. That's why programming is often linked with algorithms (even though it doesn't have to be), because solving algorithms also involves breaking big problems down into smaller steps.

In programming there are different tools for doing so. Programs can be split into smaller chunks called classes, and classes can be split into smaller chunks called methods. That sounds simple, but as you might know from experience: choosing which classes and methods to split your code into and choosing which responsibilities to assign to these classes and methods is often one of the hardest things in programming and is a complete field of study on its own (also known as Software Architecture / Software Engineering).

Over the course of history as the occupation of professional software developer evolved, certain questions pertaining to Software Architecture turned up again and again. Questions such as:

- how can we represent data using different views while keeping the code manageable?
- how can we assign and switch different behaviors for objects at runtime?
- how can we create nested data structures such as scene graphs or dialogue structures?
- etcetera

The description of such a problem and its possible solution is called a 'Design Pattern', a term first coined by the architect Christopher Alexander. Then came along the Gang of Four, a group of four people (*Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)* that published one of the most influential works on the topic ever, called *Design Patterns: Elements of Reusable Object-Oriented Software (1995),* containing 23 core object-oriented design patterns frequently used around the world today.

Any sort of non trivial software development requires architecture. For example looking at the first assignment "Implement a dungeon generator" some of the architectural questions that had to be answered were:

- Can we implement a dungeon generator that can be reused in other engines such as Unity?
- How do we visualize the result of the dungeon generation process?
- How can we satisfy both constraints?

(These same sort of questions can be asked for any of the other assignments).

Further research would inevitably lead to something called the MVC pattern (model, view, controller) in which the idea is to separate the definition of your dungeon (in terms of data and logic) from its representation on screen. In practice that means that instead of having one class called 'Dungeon' we would get several: DungeonModel, RoomModel, DoorModel, DungeonView, DungeonGenerator.

Although academically correct, forcing these patterns on you without teaching a course on Software Architecture first (year 2!) would just provide you with an additional hurdle. Writing one program called

'MyProgram' is the also not the solution however, therefore in this course we try to walk the middle ground.

In case of the dungeon generator we rolled the Model, View and Generator into one class called 'Dungeon' which extends Canvas and is responsible for generating the dungeon in terms of Rooms & Doors and through the canvas is capable of providing a (debug) view of this data. The Rooms & Doors are pure data objects, they don't extend Canvas or GameObject, they are pure data descriptors of which rooms and doors we have, how big they are and where they are. As such they can be passed around to any other class wanting to implement a means of alternative visualization (a TiledDungeonView for example).

If you are familiar with patterns or if you like a(n even bigger) challenge, you are free to convert the provided code to an MVC setup, but be warned, this is not an easy task, and one probably better left alone until after the course.

A better approach is to wait until some future time when you would like to reuse some of your classes in another engine such as Unity, because trying to that will tell you *exactly* which part of your classes should have been split into Model and View classes in order to promote reuse.