**Dung Eon : A Dungeon Crawler**

**Concept of Operations**

**COP 4331, Spring, 2016**

Team Name: Bobby Sandwich

Team Members:

* Robert Bland
* Jacob Crandall
* Jacob Jiskoot

Modification history:

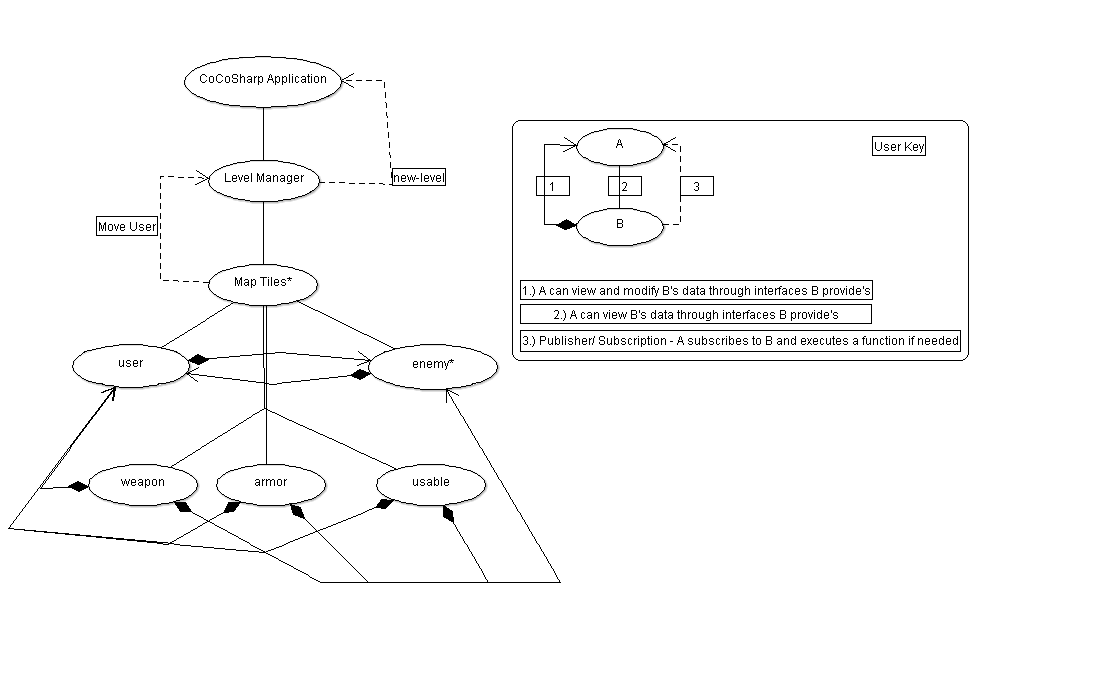
|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Who | Comment |
| v1.0 | 3/1/2016 | Jacob Crandall | Diagram & Notes initial |
| ... |  |  |  |

**Contents of this Document**

High-Level Architecture

Design Issues

**High-level Architecture**



We hoped to let the objects in our system for the most part exist independently. For this reason we are using a semi-tiered (based on object creation / ownership) peer to peer / client-server architecture with a couple instances of publisher / subscription when we need to jump outside of tiers.This lets us keep our goal of being easily upgradeable, modifiable and easily create new versions.

This describes the general flow of the program. The CocosSharp game engine provides us with an interface for creating a general application, this is our starting point. Upon starting (via main menu) a level manager will be created. The application will be in charge of this level manager. Then upon creating a level manager it will create multiple map tiles. Each map tile will have a chance to spawn an enemy, a user or items. Because map tiles can create these objects it has a direct relationship to it down a tier.

The couple of exceptions to this rule are in our two usages of publisher / subscription and our modification requests. The modification requests (represented by a black diamond and arrow) are still peer to peer style relationships but of a slightly different category. These will call functions that request to change data. For instance a use item may request to change a user’s attack stat. An enemy may request to change a user’s health stat. This differs because the other relationships imply more of a reference to the object and viewing of their data rather than modifying it.

The other exception is the publisher / subscription usages. Specifically level manager subscribes to map tiles and checks for touches on a map tile. Upon seeing a touch it initiates functionality for moving. The new-level line has the level manager check the map-tiles data for if the conditions to end a level are complete. If so then the application creates and loads a new level manager (essentially taking us to the next level).

**Design Issues**

**Reusability** - Reusability is important because it allows future developers (or us) to make easy additions without having to create some of their own interfaces (unless they wish to do so). For instance if we would like to create a new type of weapon we do not have to implement new interface abilities for that specific weapon, only to inherit the current weapon interface. This architecture is general enough to allow for future development and additions to be made. It provides a core that is easily built upon to allow for multiple different versions of the game to potentially be made.

**Maintainability** - This architecture gives a good breakdown of different parts of the system. While there are many relations the different parts are still somewhat isolated. The documented relations also allow us to easily see how changes to one part of a system may affect another system. Or if we add functionality to one part of a system, how we need to change the request / provide interfaces. This does not provide amazing documentation in terms of HOW but it gives a good idea of WHAT to maintain should we make changes.

**Testability** - Testability is very important to us and in hand with that, prototypes are as well. For this reason the tiers are very important in our architectural decisions. We can create prototypes that we build off of and are constantly testing. For instance we can create the initial starting level manager and map tiles interfaces. Then we can implement users. The users to map tiles relationship will be testable even without enemies or items.

**Performance** - We had little concern for performance in the decision of our architecture. The planned system is not excessively complicated. Our biggest concern is response time. Because we are making something for entertainment it should flow well and not be frustrating to the user. However, as I said this is not expected to be an issue and as such was not made a priority in deciding the architecture. However, as we are creating prototypes it may prove to be an issue. If this is case we will unfortunately have to look into how we can optimize our system.

**Portability** - Portability is of importance to us but that is not hugely evident in our architectural decisions. This is mostly because the main Cocosharp will be able to implement future portability with relative ease should we choose to go that route. The lower tiers will have to change no code to implement portability, only this upper upper tier.

**Safety** - We had little concern for safety in the decision of our architecture. Crashes of the system will cause no real issues and there are few other issues to consider. Should we implement all interfaces described above correctly there should be no undefined behavior and as such no way to cause a crash in the system. We have no currently planned save-system so any type of recovery from a failure is not necessary. Unfortunately a player may lose progress but the plan is to avoid failure not handle it. This will be done by only calling defined behavior via interfaces provided by objects.

Prototypes are going to be an integral part of our development and implementation lifecycle. As such they will be used continuously for many purposes including both testing and evaluating alternatives should our initial ideas not work. For this reason creating a core-prototype of the map-tiles will be incredibly important. You can see this in how central the map-tiles are in our diagram. From this prototype we can build up other map-building with the level manager, or work on how map-tiles interface with items or characters. This initial prototype may be what we use but it at least gives us a base for creating other objects. It gives us an environment to test and create other objects. As we create iterations of our project the prototypes for the compartmentalized sections of code are going to help us keep an eye out for issues (specifically performance issues) that this code may create. We can then alter the implementation if necessary.

One of the biggest technical difficulties we expect to encounter is getting all parts of the system to interact properly. Experience is key and designing systems and we have little experience with design aspects or our tools. For this reason we may not have considered certain necessary ideas that we need in the interface to allow components to effectively communicate the information they need to. Luckily the fix for this is pretty simple, simply add to an object's interface so it can give or get information it needs. However, this introduces a new risk which is maintaining our organization and therefore compromising our ability to have as compartmentalized code as possible. This introduces issues with a lot of our goals in the project, such as maintainability, reusability and the ability to be easily understood. The best fix for this is to simply be aware of it and make sure that we are all referring back to these architecture documents and respecting relationships. If there are serious communication between objects then we may have to reconsider certain parts of the architecture, which will be done as a group.

One of the biggest reasons for selecting this architecture is the idea that we can compartmentalize pieces of our code. The main reason for picking this architecture is that it lets us maintain our incremental development model. Once we have developed simple prototypes of each object we can easily upgrade and add to these separate parts. This architecture provides an easy to see relationships visualization. So should a section of this be upgraded we can see what it directly interacts with and how it should be tested. This means that we may not have the most efficient system unfortunately. Requests and data may take a slightly inefficient route in order to maintain an idea of organization in the code.