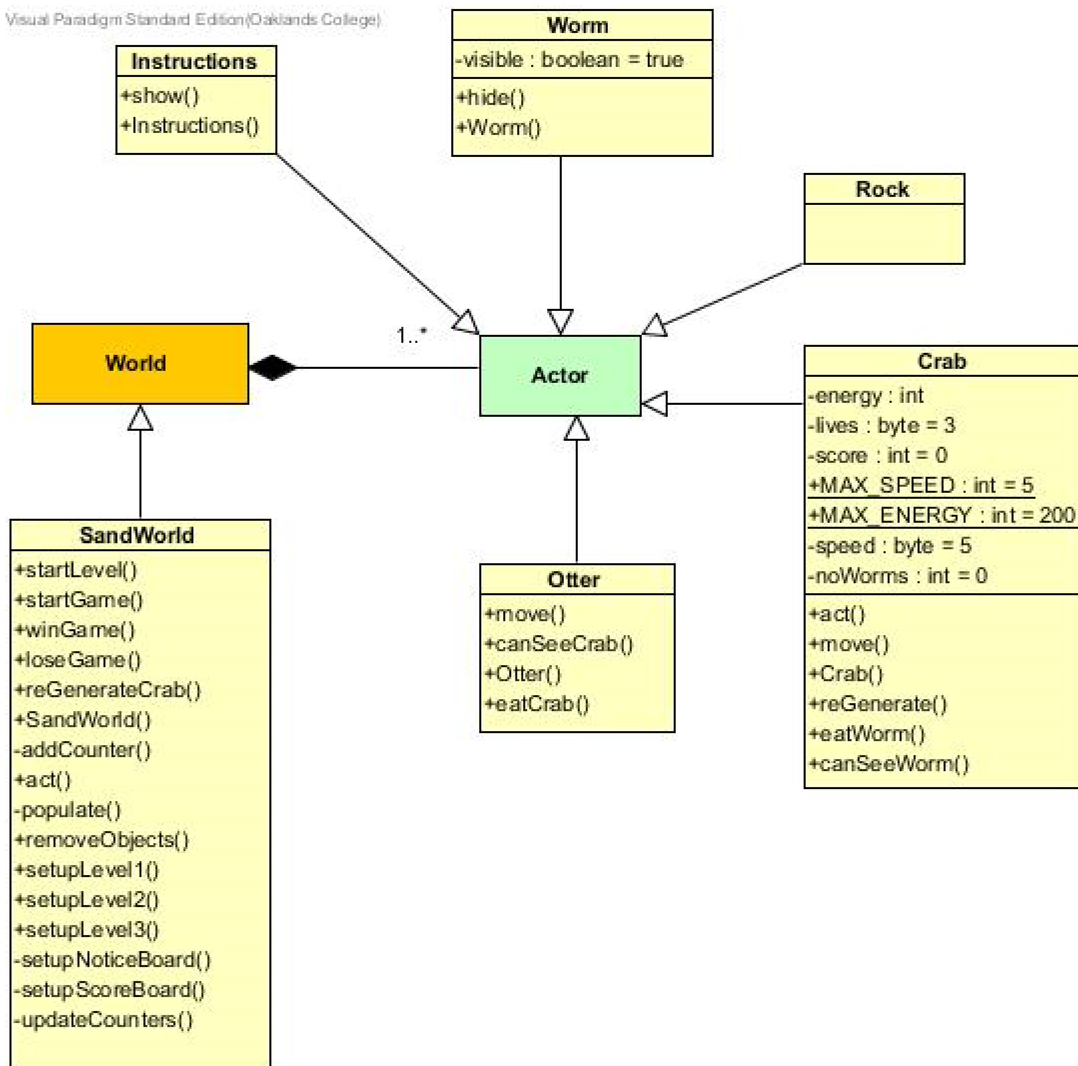
D2: explain how the structure and design of a game can assist in maintenance and capacity for extension.

Game Design

The original design for the CrabEscape game was

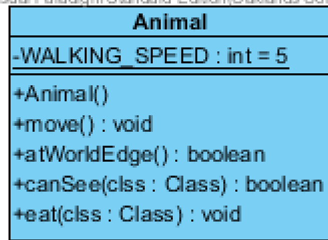


There were however a number of issues with this design. In particular the SandWorld class contains too many setup methods, and no attributes. There is no direct connection between the SandWord and the Instructions class, the Crab or the Otter.

Inheritance

Inheritance is one key way of implementing the **DRY** design principle (Don't Repeat Yourself). The **Otter** moves around looking for **Crabs** and if it gets close enough eats the **Crab**. The **Crab** moves around looking for Worms, and if it gets close enough eats the worm.

To avoid any unnecessary duplication an **Animal** class should have been introduced based on the **Actor** class, and containing a move() method, an eat() method and a canSee() method. Eat and CanSee need a parameter of class so that the method can be used they can be used on any object.



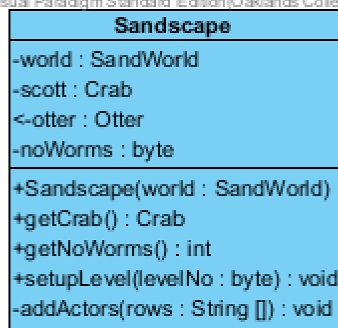
The Animal is an Actor, and the Otter and the Crab are Animals, and therefore Actors, inheriting all the properties and methods of Actors and Animals. All animals need to be able to detect the edge of the world, and respond accordingly.

Single responsibility Principle

Each class should have a single responsibility, the **SandWorld** class is responsible for controlling the game, and setting up the game. This class is too big and does too many operations. That is why a Map class was added (**Sandscape**), and this class is responsible for setting up the various game levels and providing access to the Otter and the Crab.

It is then easier to then to maintain and extend the game as the map class can be updated if new levels are needed, and the SandWorld class can be updated if the way the game works changes.

It is also easier to maintain and debug two smaller classes, rather than one large class.



Readability

One of the most important factors that affect the ease of maintenance and extension is making the code and the design as understandable as possible. The most important factor is the choice of names for classes, attributes and methods. For example, **Sandscape** is not necessarily the most obvious name for this class. Quite a number of games are based on the concept of a **Map**, so calling this class **Map**, would have been better for future developers to immediately understand what its primary responsibility is.

Similarly Instructions was a poor name for a class that's purpose is to display game instructions to the user. NoticeBoard, or MessageBox are much clearer. In fact a NoticeBoard can be used for more than just displayInstructions(), it can also be used to displayWinGame() i.e. messages to the user when they win the game.

The method populate is another example of a poor choice of name, as another developer will not know what objects are populating what object without looking at the code or comments.

Consistency of naming is also crucial with singular nouns for class names, names starting with verbs for method names, and plural names used for lists or arrays. All names in java other than class names should start with a lower case letter, and each subsequent work starting with a capital letter. They all help coders understand the code at a glance without having to examine the code in detail.

Block comments should also be used to explain each class, and each method as they can be used to automatically generate API documentation in html. They need however to explain the details that are not obvious from the name of the class or method. Here is an example.

```
 /**
 * This method will show the user a description of how
 * the game works, which keys to use and how the scoring
 * system works.
 */
public void showInstructions()
{
    showDescription();
    showKeys();
    showScoring();
}

/**
 * This method declares and array of strings containing
 * the text for the game description and that text is
 * drawn onto the Noticeboard.
 */
private void showDescription()
{
    yPos = TOP_MARGIN;
    xPos = LEFT_MARGIN;

    String [] lines =
    {
        "In this game you move the crab in order to eat worms.",
        "You complete the level when all the worms are eaten, ",
        "you must avoid being eaten by the otter, the otter ",
        "cannot see the crab if the crab is behind a rock."
    };

    showHeading("Game Description");
    showLines(lines);
}
```

The layout of the code using indentation and blank lines is important, as a typical game can be millions of lines of code, so it is vital that developers can scan quickly through the code to find any particular part.

Another aspect of this is using named constants or variables rather than numbers in the code. It make the code more readable, and also easier to extend and maintain. In this example it is easier to move the position of the text in the NoticeBoard, by just changing the values of the constants.

```
public class NoticeBoard extends Actor
{
    public static final int LEFT_MARGIN = 60;

    public static final int TOP_MARGIN = 80;

    public static final int LINE_HEIGHT = 20;

    private int yPos;

    private int xPos;
```

Minimize Upfront Design

This principle is also known as YAGNI (You ain't going to need it!). The showDescription method uses and array of strings to setup messages that the NoticeBoard will display. Ideally the actual messages should be in a text file, which is loaded by the method. This would make it much easier to extent or maintain the code the actual messages can be more easily changed without having to change or re-compile the codebase. So for example different versions of the messages in different languages would be possible.

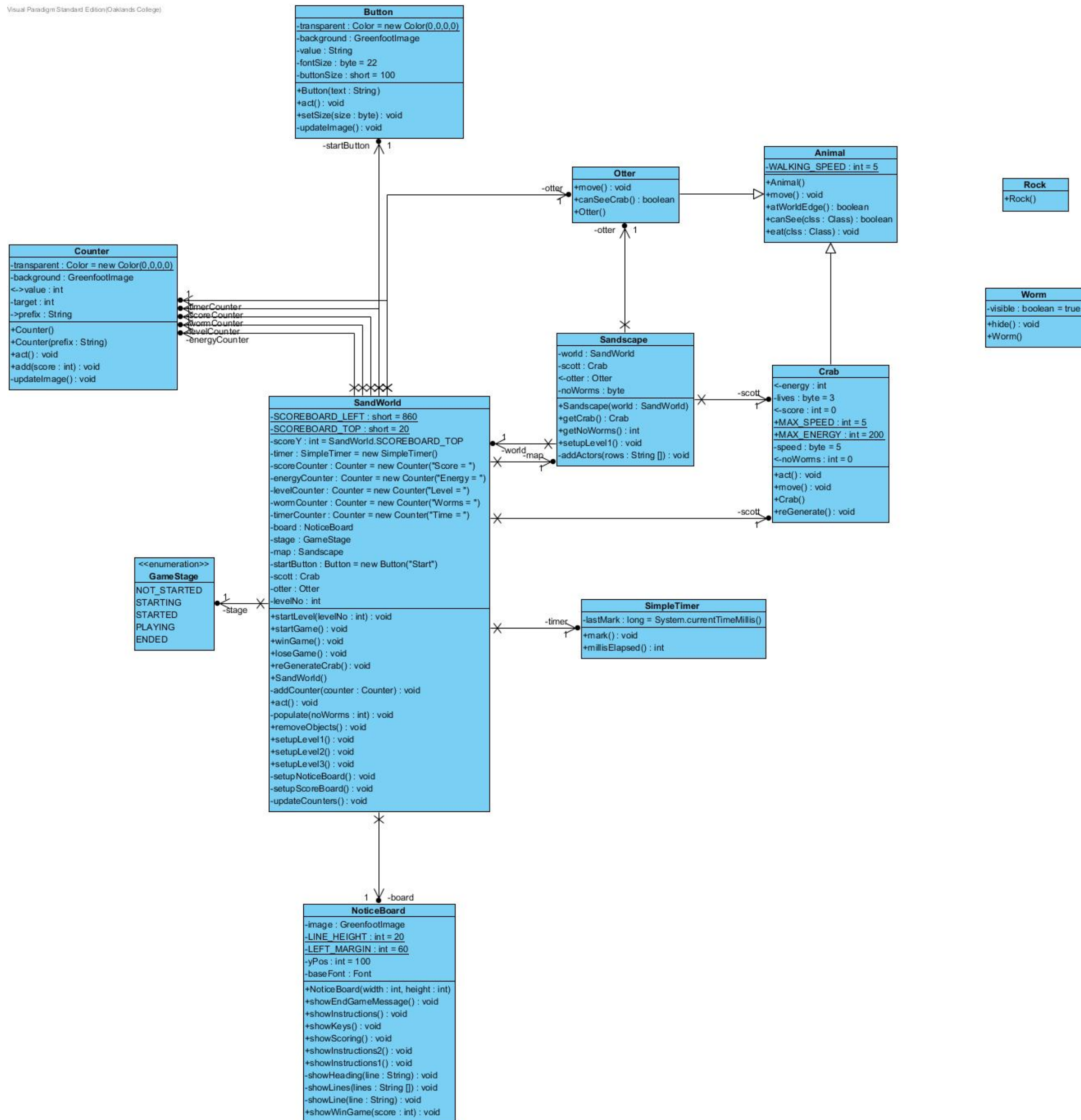
However this game was being developed for assessment purposes, and I knew I wasn't going to need it!!!

The need to Re-factor

Although the design started out nice and simple, as the code was largely added without re-factoring it as we went along the end result looks overly complex. It would be easier to maintain, and easier to extend if it was examined again, and simplified where possible. There are well established "smells" that can be found and removed, making the code much easier to maintain.

Final Design

The following design was reverse engineered from the final code version of the game. It shows how complex the final version is. Using established design patterns is one way that this could perhaps be simplified.



Good sources of reference

1. "Code Complete 2" by Steve McConnell http://www.stevemcconnell.com/cc.htm
2. "Top 15+ Best Practices for Writing Super Readable Code" http://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118
3. "Code Refactoring" http://refactoring.com/catalog/index.html
4. "Design Patterns in Games Programming" http://www.gamasutra.com/blogs/MichaelHaney/20110920/90250/Design_Patterns_in_Game_Programming.php