# Iteration Three

It is a silly game where nobody wins.

## The goal

To implement a scoring system and add lives to the game.

### Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Time Estimate** | **Actual Time Taken** |
| Analysis and Planning | 90 | 120 |
| Coding | 90 | 120 |
| Testing | 30 | 60 |

### Analysis and Planning

Two essential components of any platform game would be a scoring system, and a limited number of lives. I would like players to be able to earn more points the faster they collect the hardhats in the game. Points and lives will be maintained across levels with lives decreasing and points earned on a given level reset if a player contacts lava or the timer runs out.

I will give players a total of three lives, after which the game will end, and the hardhats will be worth double the number of seconds left in the level.

After my last iteration I learned that attaching attributes using \_\_proto\_\_ has been deprecated, so I will be creating a new class called Game which will be attached to the State prototype using State.prototype. The Game class will contain the timer, a tally of total points, a points buffer and the number of lives left. Data within the Game class will be modified using class methods and get methods will be used to return clock and points strings.

Game.countDown() will decrease the timer by the number of seconds passed to it as an argument. It will return false if there is no time left on the clock, otherwise returning true.

Game.collectItem() will perform game actions based on the type of item ‘collected’.

Game.winLevel() will add the pointsBuffer onto the totalPoints then reset pointsBuffer to zero.

Game.loseLife() will reset the pointsBuffer to zero and subtract a life. If there are no more lives to subtract it will return false, otherwise returning true.

Game.end() is a reserved function that will currently print a message to the console based on the final status. I have plans to use it to handle more complex end-game actions in a future iteration.

### Class Diagram Before:



### Class Diagram After:



### Design

## Activity Diagram



## Planning a Complex Algorithm

Define the problem

*Handle game events based on collision with a collectible item (e.g. points for hats)*

Inputs to the routine

*Type of item collected*

Outputs from the routine

*None*

Pre-conditions

Time is known, game status is playing

Post-conditions

*Game event is triggered (e.g. points added to points buffer)*

Name the Routine

Game.*collectItem()*

Think about error handling

*I will use a switch method with the default switch used in future to handle error condition (maybe a sound effect?)*

### Pseudocode

Game.collectItem (typeOfItem) {

switch (typeOfItem) {

case ‘coin’:

points = secondsLeft x 2  
add points to buffer  
break

default:

do nothing  
break

}

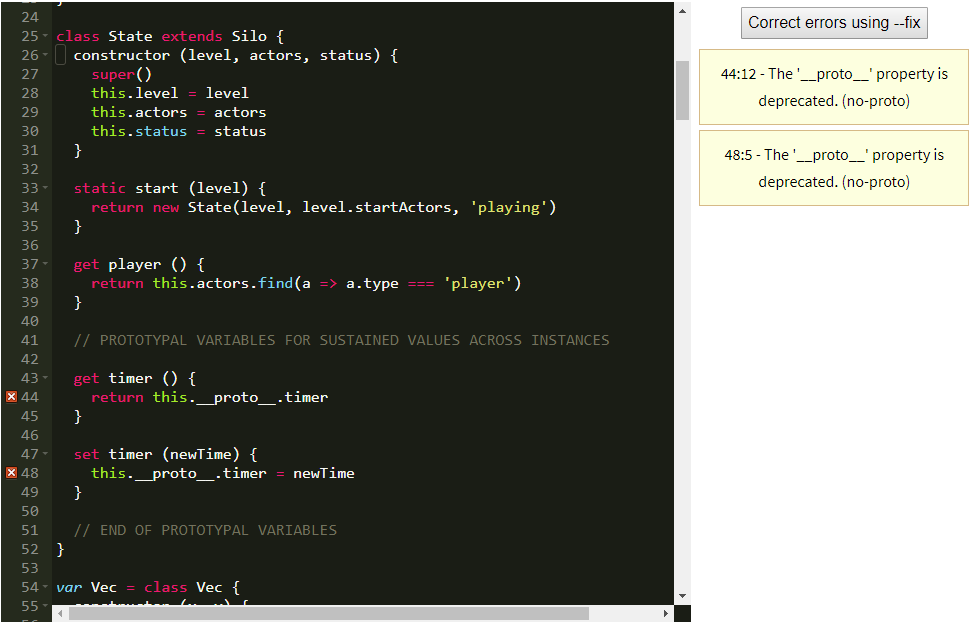
}

### Evaluation

I had originally planned to attach all the attributes needed for this iteration to the State prototype, and even got so far as a working copy using get and set methods to access \_\_proto\_\_ data directly. After running it through standardJS I learned that using \_\_proto\_\_ was deprecated and I needed to find a better way to do it.

After some more time looking into alternatives I settled on making a new class (Game) to contain the required data and attach that to State.prototype at the start of the game. I learned to do a bit more research before implementing new features during development.

### standardJS Before



### standardJS After

