TL 4 Presentation

MILCCN Studios - Casey
Nautilus - Shawn
Tappa Tappa Keyboard - Taran
Make It Awesome - Amara
Penguine - Emily
Team 5 Guys - Austin
Teal Team 6 - Daniel

Learning Outcomes

- Oral Exam Information
- GRASP
- Coding Standards
- The Roast

Oral Exam

- The week of **November 11th** - less than a month away !
- Moscow campus students (unity) - meeting with BC on **Monday** and **Friday**
- Moscow campus students (godot) - meeting with BC on **Tuesday**
- Coeur D'alene students - most likely meeting with BC on **Thursday**
    - At some point in lecture BC will ask students to pick their preferred date (people who attend lecture get first picks)
- There is an **Oral Exam Marking Key** on her website that you will partly fill out (sections are specified in the oral exam drop box assignment)
- The oral exam drop box assignment is due **November 10th**
- The oral exam itself is a 20-30 min meeting with BC
- JEB 324

Oral Exam Requirements

(these are documented better in the oral exam marking key)

Contribution Requirements

- Your contribution to the game - description of feature
- Gantt chart self estimation - did your original estimation match your actual time
- play your game and point out where you code is called and ran
- Show your scripts and what methods of yours was called

Technical Requirements

- Test plan walk through
- Well documented prefab explanation
- Show a class in our code and where there could be either static or dynamic binding
  - You will also need to add a virtual function in your code
- Example of reuse in code - where you violate copylaw
- Code patterns (1 big or 2 small ones)

Oral Exam Requirements
(info on BC website)

Oral Exam

See the attached Oral Exam Marking Key for the areas that you will be marked on. This is worth 20% of your final grade. It is a completely individual mark. This will be a one-on-one 30 minute interview with the professor. (Or a 20 minute with the TA 20 minute with the professor for larger classes.) Walk me through your feature and your code.

- Your code must function the way the client has requested.
- Your code must be approximately equal (or greater) in work requirement to your team mates.
- You must have at least two patterns and be able to justify their use.
- Your documentation must match your code.
- You must comply with your team's coding standards.

Walk me through your time estimates and actual time spent.

Where to go:

JEB 324

What to bring:

(Arrive a bit early so you are logged in and ready to go when the time starts)

- Computer capable of running your code with access to your GIT repository.
  - Note: If it is not in your GIT repository it is not markable. (By definition, anything printed is out of date.
  - You cannot bring any study aids with you except what is in your code, but feel free to document your code as much as you want (for example if you think you might not remember what the key word 'virtual' does for static binding put a note about it in your code.)

No study aids but you can comment your code as much as you want!

GRASP

- General Responsibility Assignment Software Patterns
  - a set of "nine fundamental principles in **object design and responsibility assignment**"
- 9 principles
  - Controller
  - Creator
  - Indirection
  - Information expert
  - Low coupling
  - High cohesion
  - Polymorphism
  - Protected variations
  - Pure fabrication
- These patterns solve some software problems common to many software development projects

GRASP

- General Responsibility Assignment Software Patterns
  - a set of "nine fundamental principles in object design and responsibility assignment"
- Design of objects?
  - We are initially trained to think of objects in terms of data structures and algorithms but with Responsibility driven design (RDD) this shifts in treating objects as having roles and responsibilities
- Responsibility driven design (RDD) - asks questions like
  - What are an object's responsibilities?
  - What are an objects roles?
  - What are an object's collaborations?
  - Objects become - service providers, information holders, coordinators, controllers, etc

GRASP

- What is an object responsibility?
- A responsibility is defined as “a contract or obligation of a classifier.”
- Responsibilities are related to the obligations of an object in terms of its behavior
  - Two types of responsibility
  - Knowing responsibilities of an object include:
    - knowing about private encapsulated data knowing about related objects
    - knowing about things it can derive or calculate
  - Doing responsibilities of an object include:
    - doing something itself, such as creating an object or
    - doing a calculation
    - initiating action in other objects
    - controlling and coordinating activities in other objects
- A responsibility is not the same thing as a method

GRASP

- Methods are implemented to fulfill responsibilities
- Responsibilities are implemented using methods that either act alone or collaborate with other methods and objects
- Assumption:
  - responsibilities are implemented by means of methods
  - the larger the granularity of responsibility, the larger the number of methods
  - this may entail an object interacting with itself or with other objects
- Therefore:
  - fulfilling responsibilities requires collaborations amongst different methods and objects

GRASP

- Lots of information BUT the whole point:
  - One of the reasons object oriented approaches are so valuable is that they allow the design of the software to closely match the real world
  - This makes it easier to understand what the code is doing, given an understanding of the problem domain
  - Experienced object-oriented developers (and other software developers) build up a repertoire of both general principles and idiomatic solutions that guide them in the creation of software
- GRASP is a *methodical approach to object oriented design* created by Craig Larman
- GRASP is *patterns of assigning responsibilities*
- GRASP *recognizes that responsibility assignment is something we already do*
  - All the diagrams we made were about assigning responsibilities

Low coupling pattern

- Coupling: the degree to which each program module relies on the other module
- Another definition: degree of interaction between two modules
- Another definition: Coupling is a measure of how strongly one element is connected to, has knowledge of, or
- relies on other elements
- The problem: How can our design support
    - Low dependency?
    - Low change impact?
    - Increased reuse?
- The solution:
    - Assign a responsibility so that coupling remains low
    - Use this principle to evaluate alternative assignment of responsibilities.
- 5 levels
    - 5. Data (Best)
    - 4. Stamp
    - 3. Control
    - 2. Common
    - 1. Content (Worst)

Low coupling pattern

- Starting at the worst: **Content Coupling**
- Definition: one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling

```
public class Enemy
{
  private int health;
   public void TakeDamage(int damage)
   {
       health -= damage;
   }
}
public class Player
{
   public void AttackEnemy(Enemy enemy)
   {
       enemy.health -= 10;
   }
}
// Directly modifying Enemy's private health
```

```
public class Enemy
{
   public void TakeDamage(int damage)
   {
      health -= damage;
   }
}
public class Player
{
   public void AttackEnemy(Enemy enemy)
   {
       enemy.TakeDamage(10);
   }
}
//use public methods to access the health
variable
```

Low coupling pattern

- Next: **Common Coupling**
- Definition: The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change.

```
//global variable
public static int PlayerHealth = 100;

public void AttackPlayer() //enemy class
    {
     PlayerHealth -= 20;
// Directly modifying the global variable
    }

public void ApplyHealthBoost() //powerup class
    {
     PlayerHealth += 50;
// Also modifying the global variable
    }
```

```
public class Player //create player class
    {
        public int Health { get; set; }
    }
public void AttackPlayer(Player Player) //enemy
class
    {
     player.health -= 20;
    }

public void ApplyHealthBoost(Player Player)
//powerup class
    {
     player.health -= 50;
    }
```

Low coupling pattern

- Next: **Control Coupling**
- Definition: one module passes an element of control to the other
- Example
  - control-switch passed as an argument
- Why is this bad?
  - modules are not independent
  - module b must know the internal structure of module affects reusability

Low coupling pattern

- Next: **Stamp Coupling**
- Definition: data structure is passed as parameter, but called module operates on only some of individual components

```
public class ItemManager
{
public Dictionary <string, int> Inventory {get; set;}
public void AddItem(string itemName, int quantity)
    {
        Inventory.Add(itemName, quantity);
    }
}
public class Player
{
    public ItemManager Inventory { get; set; }
    public void UseItem(string itemName)
    {
        // Player only needs to access one item from the Inventory
        if (Inventory.Inventory.ContainsKey(itemName))
        {
            // ... use logic
        }
    }
}
```

```
public interface IInventory
{
   int GetItemCount(string itemName);
   bool HasItem(string itemName);
}
public class Player
{
   private readonly IInventory _inventory;
   public Player(IInventory inventory)
   {
       _inventory = inventory;
   }
   public void UseItem(string itemName)
   {
       if (_inventory.HasItem(itemName))
       {
           // ... use logic
       }
   }
}
// Implementation of IInventory could be a class like
"PlayerInventory" which only exposes the necessary data
```

Low coupling pattern

- Next: **Data Coupling (BEST) -** This lowers the dependencies between modules
- Definition: every argument is either a simple argument or a data structure in which all elements are used by the called module

```
public class Player
{
   public void AttackEnemy(Enemy enemy)
   {
       enemy.Health -= 10;
      // Player directly modifies Enemy's health
   }
}
public class Enemy
{
   public int Health { get; set; }
}
```

```
public class Player
{
   public void AttackEnemy(Enemy enemy)
   {
       enemy.TakeDamage(10); // Player only
needs to know about the "TakeDamage" method
   }
}
public class Enemy
{
   public int Health { get; set; }
   public void TakeDamage(int damage)
   {
      Health -= damage;
   }
}
```

High Cohesion

- Classes should be designed to do one thing, and one thing only
- Think of it as a functional unit - every part of a screwdriver should be optimized for driving screws
- Any kind of a multitool is a compromise

Types of Cohesion

- Coincidental - parts are grouped completely arbitrarily (grab bag)
- Logical - parts all do the same thing, more or less (three wrenches, five sockets and an oil filter wrench)
- Temporal - parts are all called at the same time (funnel, oil filter wrench, drip pan)
- Procedural - parts act in order (same as above, but add a X wrench and a jack because you rotate your tires when you change your oil like you’re supposed to, right?)
- Communicational - parts all need the same data (drill bits, screwdriver bits)
- Sequential - one part picks up where another leaves off (saw, grinder, sander)
- Functional - parts work together to do one thing (socket kit with ratchet)
- Perfect - you cannot simplify this any further (one, singular, 10mm wrench)

Coincidental Cohesion

```
Unity Script | 0 references
public class FredDialogue : Scooby
{
    public string favcolor = "Orange";
    public int dialoguenum = 0;

    private PongScoreManager scoremanager;
}
```

Logical Cohesion

```
    // update the score display in the pong game
    2 references
    private void UpdateScore() ...

    // add one to the player score
    1 reference
    public void addPlayerScore() ...

    // add one to the ai score
    1 reference
    public void addAIScore() ...

    // dynamic binding to check win condition for pong game
    2 references
    public bool CheckGameOver() ...

    1 reference
    private void ResetRound() ...

    // dynamic binding to end the Pong game
    3 references
    public override void EndGame() ...

    0 references
    public void HandleContinueButtonClick() {
        Debug.Log("Loading Level1");
        SceneChanger.Continue();
    }
}
```

Temporal Cohesion

```
Unity Script | 0 references
public class FredActions : Scooby
{
    0 references
    public int PlayerResponds()...
    //response includes minigame choice
    0 references
    public int MiniGameOpens()...
}
```

Procedural Cohesion

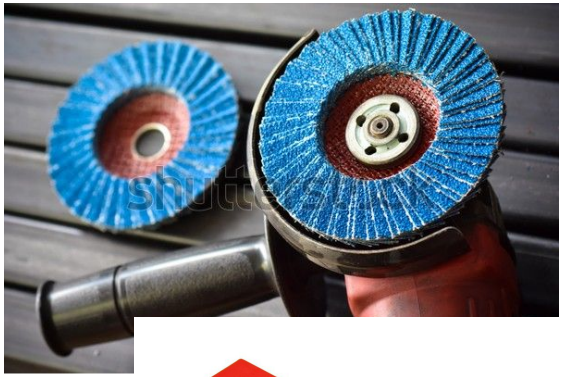
```
1 reference
private int getDialogueNum()...

//buttons to reset the buttons ('unclick' them)
public Button r1;
public Button r2;
// private int responseNum = 0; ...
0 references
public void hitResponse1()...

//button 2 calls onclick
0 references
public void hitResponse2()...


//a button being hit will trigger this. decides how to respond
2 references
private void toNextDialogue()...

//prints the new dialogue
/*  ...
```

Communicational Cohesion

```
25
26  >       private void Awake() ...
30          }
31          private void Update()
32          {
33              MyInput();
34          }
35  >       private void MyInput() ...
47          }
48  >       private void Shoot() ...
79          }
80          private void ResetShot()
81          {
82              readyToShoot = true;
83          }
84  >       private void Reload() ...
88          }
89  >       private void ReloadFinished() ...
93          }
94      }
```

Sequential Cohesion

```
8       [SerializeField]
9       private GameObject PauseMenuObject;
10      /*
11      [SerializeField]
12      private GameObject SkillMenuObject;
13      [SerializeField]
14      private GameObject ItemMenuObject;
15      */
16
17      [SerializeField]
18      private Button PlayButton;
19
20      // Start is called before the first frame update
21  >   void Start() ...
29      }
30
31      // Update is called once per frame
32  >   void Update() ...
39      }
40
41  >   public void PauseGame(){ ...
45      }
46
47  >   public void ResumeGame(){ ...
51      }
52  }
```



Daniel Franks | Teal Team 6

Functional Cohesion

```
12          [SerializeField]
13          private Camera playerCamera;
14
15
16      [SerializeField]
17          public float walkSpeed = 10f;
18          //Me change this to public
19
20          [SerializeField]
21          private float lookSpeed = 7f;
22
23          [SerializeField]
24          private float lookYLimit = 100f;
25
26          // Vector tracking movement direction
27          public Vector3 moveDirection = Vector3.zero;
28      //Me change this to public
29
30      float rotationX = 0;
31
32          // The required CharacterController object
33          CharacterController characterController;
34
35      // Start is called before the first frame update
36      void Start() ...
42      }
43
44      // Update is called once per frame
45      void Update() ...
55      }
```

imagine

in your mind

the perfect class

nirvana

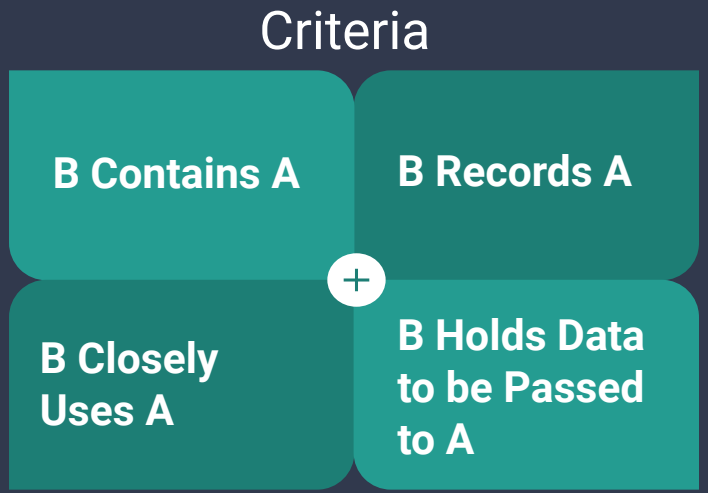
Perfect Cohesion

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Environment : MonoBehaviour
6   {
7       // Start is called before the first frame update
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
14      void Update()
15      {
16
17      }
18  }
```

Creator Pattern

- Similar to factory GoF pattern
- Instantiate objects that don’t instantiate themselves
- Decreases coupling

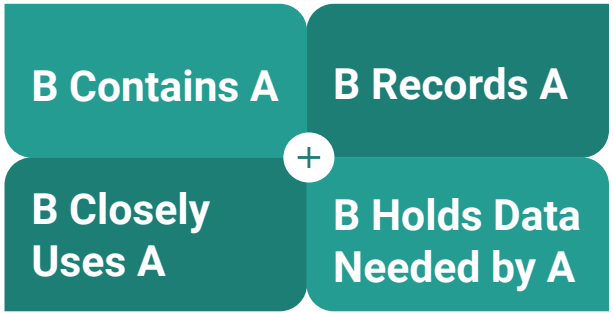
Selecting a Creator

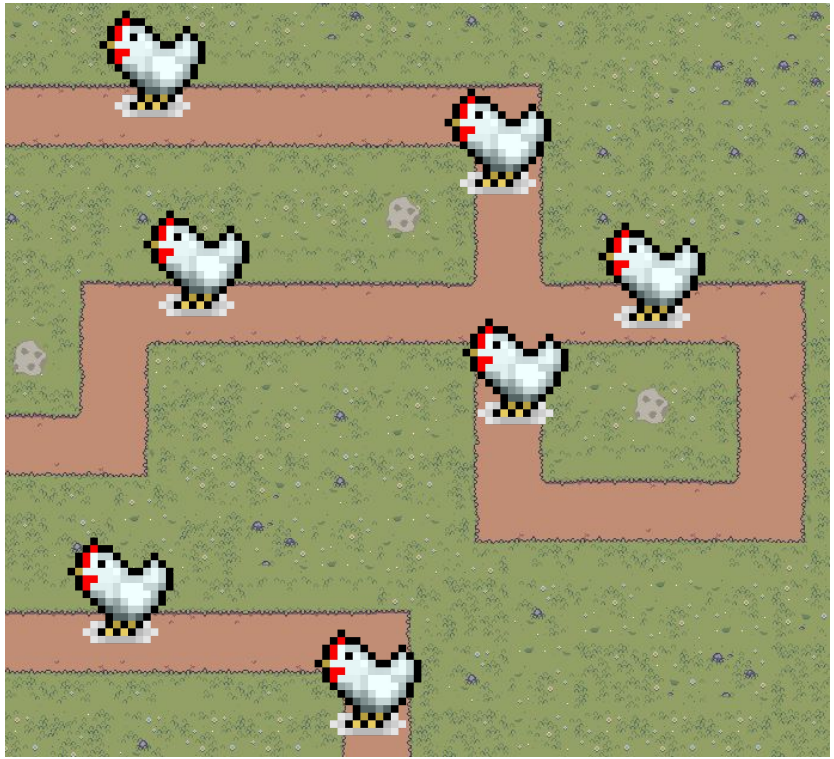


- Goal: Select object with already strong coupling
- Each criteria indicates higher coupling

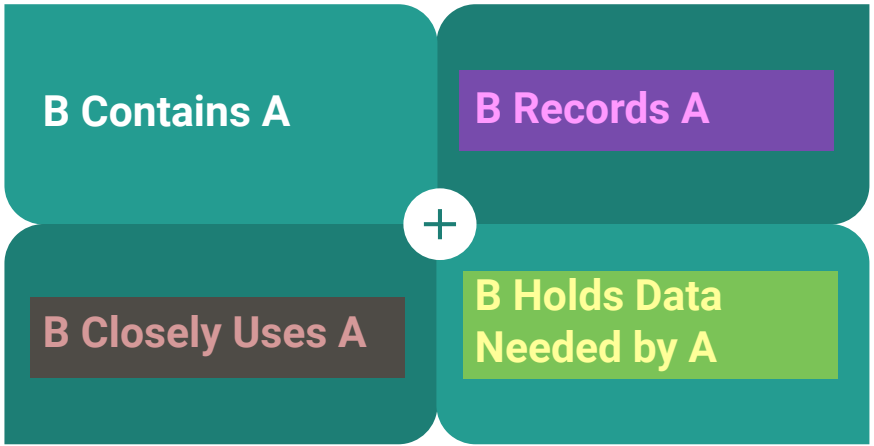
Example Problem

- Chicken objects need to be instantiated
- Who will do it?
  - Chicken factory
  - Level manager
  - Round manager





Example Problem Continued



src > Nathan > RoundManager.cs > ...

```
24 references
16  public partial class RoundManager : Node2D {
17
18
        13 references
19      public RoundStatusTracker roundStatusTracker;
        10 references
20      private ILevelData lastLevelData;
        12 references
21      private List<SpawnOrder> spawnQueue;
        8 references
22      private List<BaseChicken> liveEnemies;
        0 references
23      private System.Timers.Timer spawnTimer;
        5 references
24      private Difficulty difficulty;
        8 references
25      private double currentTime;
        7 references
26      private double nextSpawnTime;
```

Solution

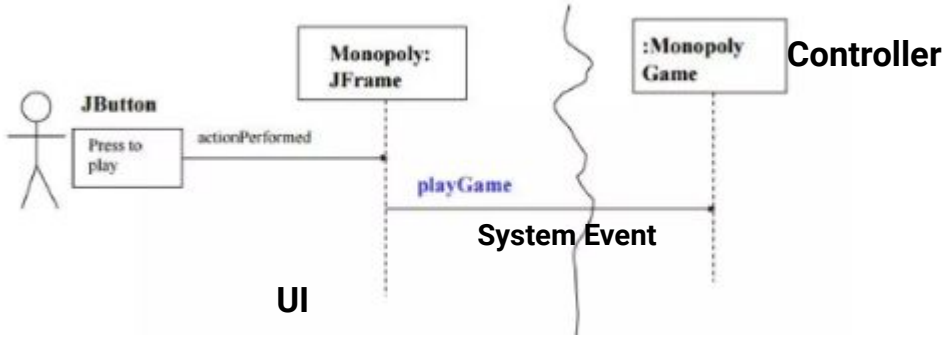
src > Nathan > RoundManager.cs > ...

```
16  public partial class RoundManager : Node2D {
39
        2 references
40      private void spawnEnemy(){
41          if (spawnQueue.Count == 0){
42              return;
43          }
44          GD.Print("Im spawning a chicken!");
45          SpawnOrder order = spawnQueue[0];
46          spawnQueue.RemoveAt(0);
47          order.Enemy.EnemyDied += HandleEnemyDiesSignal;
48          order.Enemy.EndOfPath += HandleEnemyFinishedSignal;
49
50          order.Enemy.Start(lastLevelData.LevelPath);
51          this.liveEnemies.Add(order.Enemy);
52          this.nextSpawnTime = this.currentTime + (order.spawnDelay / 1000.0);
53          GD.Print(order.Enemy);
54          // spawnTimer.Interval = order.spawnDelay / 1000.00;
55          // spawnTimer.Enabled = true;
56      }
```

Controller

What is it?

The Controller pattern assigns the responsibility of handling system events to a **non-UI** class that represents either the entire system or a specific use case. A controller object is a **non-UI** entity tasked with receiving or handling system events.



Controller – Problem and Solution

What first object **beyond the UI layer** receives and coordinates (“controls”) a system event?

Assign a responsibility to a class based on one of the following:

- Class is “root object” for overall system or major subsystem. (Façade Controller)
- A new class based on use case name. (Use Case Controller)

Example: Imagine a banking system with use cases like "Open Account," "Close Account," "Transfer Funds," and "Check Balance."

A Facade Controller (BankSystemController) could act as a centralized entry point that receives system events for all of these use cases and delegates to them appropriately.

A Use Case Controller might handle a specific use case like OpenAccountController or TransferFundsController, each focusing solely on the tasks related to that individual operation.

Controller – Some Issues

Most common issue is to have too few controllers making them “bloated controllers”, this leads to low cohesion. You must have a balance between having too many controllers and too few controllers. Examples:

- Single controller classes for every single event (Façade Controller)
- Controller performs many of the events itself rather than delegating work
- Controller maintains a lot of state information about the system (i.e. variables and attributes)

Aspect	Facade Controller	Use Case Controller
Scope	System-wide or subsystem-wide	Specific to one or a few use cases
Responsibility	Manages high-level coordination for multiple related use cases or subsystems	Focused on a single use case or closely related use cases
Complexity	More abstract, broader scope, risks bloating	More focused, easier to maintain
Granularity	Less granular, higher-level operations	More granular, specific operations
Usage	When centralizing control over many subsystems or use cases	When handling a distinct, isolated use case

Polymorphism

Problem: Handling objects based on the class they belong to and creating pluggable software components

Solution: When behaviors vary depending on class, the differences need to be assigned to specific classes where the behaviors differ

```
Unity Script | 1 reference
public class Scooby : MonoBehaviour
{
    //tracking what number dialogue response
    protected int SCdialogueNum = 0;

    //tracking affection points
    public int SCAP;

}
```

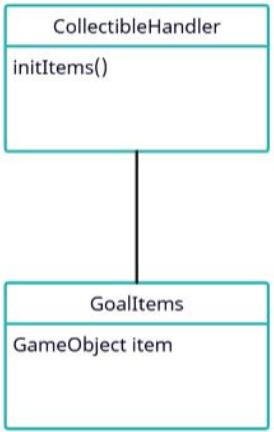
Information Expert

Problem: What is the principle for assigning responsibilities to objects?

Solution: Responsibilities should be assigned to the class with the most information

Information Expert - Example

GoalItems is the “expert” that handles tracking items and calculations with them




```
public class GoalItems : CollectibleHandler
{
    public GameObject stillsuit; //singular item temporarily
    public GameObject tent;
    public GameObject knife;
    public GameObject hooks;
```

Protected Variations

Problem: Design a system where variations in one element does not affect other objects

Solution: Create an interface to make variable objects appear invariable

Protected Variations - Example

Interface

```
    float verticalVelocity = rb.velocity.y;

    //Sends speed and grounded state to animation script function
    playerAnimationController.UpdateAnimation(moveInput, grounded, verticalVelocity, spacebarPressed);
}
private void Move()
{
    moveInput = Input.GetAxis("Horizontal");
```

Protects variable code

```
public class PlayerAnimationController : MonoBehaviour
{
    private Animator animator;
    private SpriteRenderer spriteRenderer;

    void Start()
    {
        animator = GetComponent<Animator>(); // Get the Animator component
        spriteRenderer = GetComponent<SpriteRenderer>(); // Get the SpriteRenderer component
    }

    // Called by PlayerMovement to handle animations
    public void UpdateAnimation(float speed, bool isGrounded, float verticalVelocity, bool spacebarPressed)
    {
        // Update Animator parameters based on the player's speed and grounded state
        animator.SetFloat("Speed", Mathf.Abs(speed));
        animator.SetBool("isGrounded", isGrounded);
        animator.SetFloat("VerticalVelocity", verticalVelocity); // For Falling
        //animator.SetBool("SpacePressed", spacebarPressed);


    }

    // This function handles flipping the sprite direction based on movement direction
    public void FlipSprite(float moveInput)
    {
        if (moveInput > 0)
        {
            spriteRenderer.flipX = true; // Face right
        }
        else if (moveInput < 0)
        {
            spriteRenderer.flipX = false; // Face left
        }
    }
```

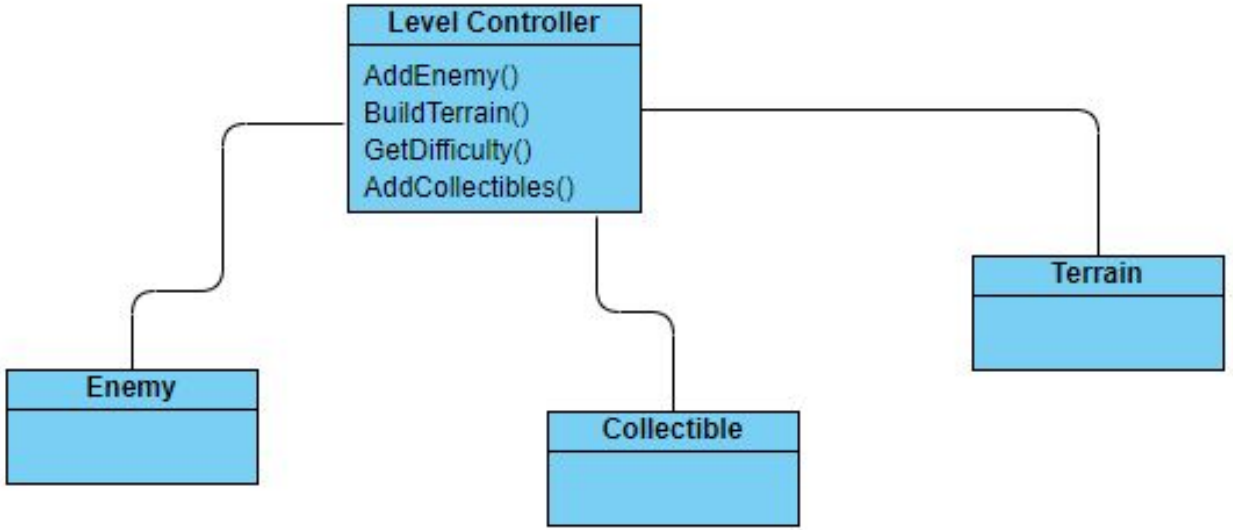
Pure Fabrications

Problem: Where is responsibility assigned when you run the risk of violating low coupling or high cohesion?

Solution: Assign a highly cohesive set of responsibilities to a new class created explicitly for that purpose.

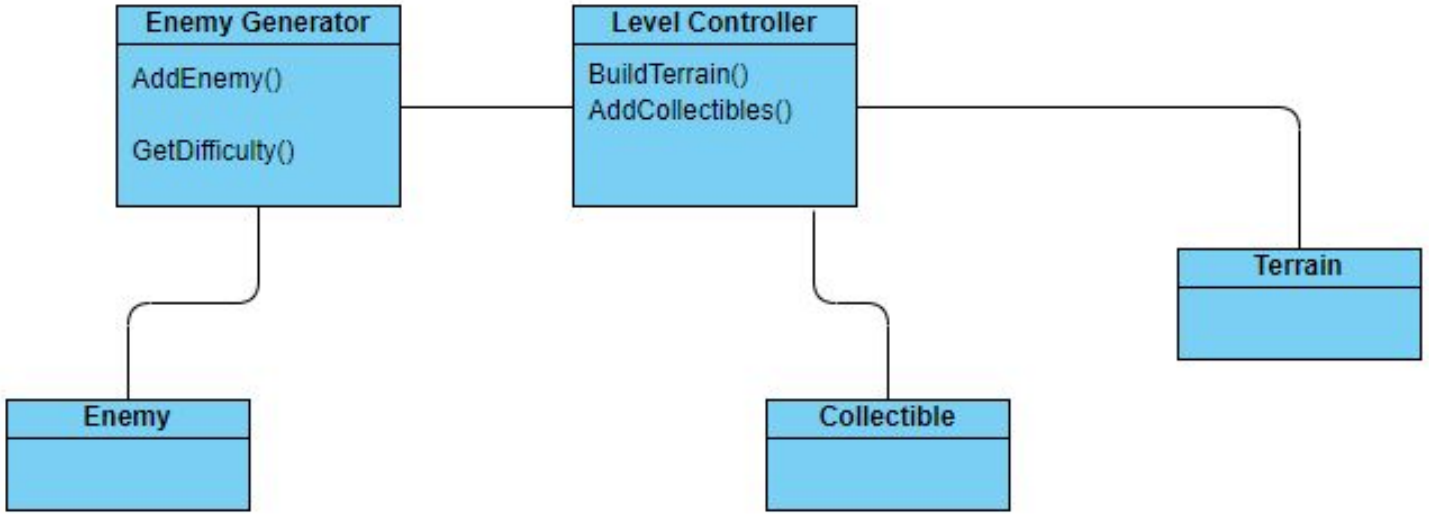
Pure Fabrications - Example

Low Cohesion



Pure Fabrications - Example (cont.)

Pure Fabrication: Higher Cohesion,
Lower Coupling



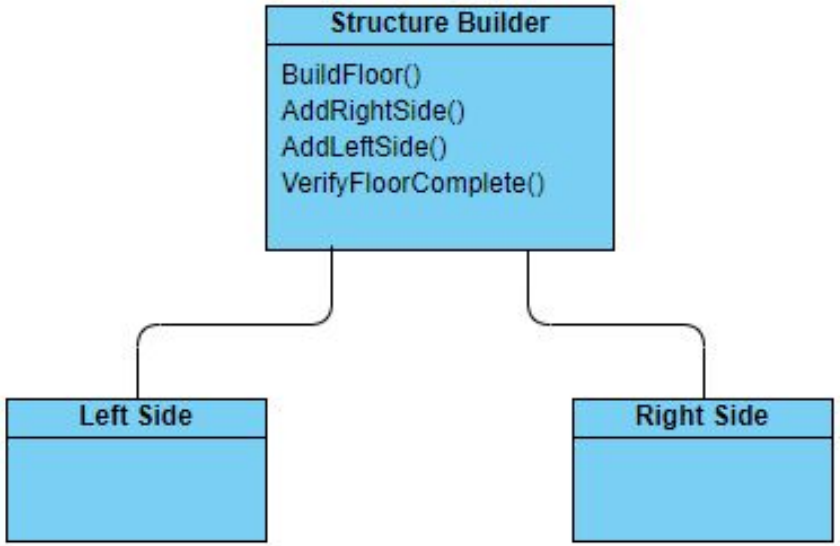
Indirection

Problem: Where to assign responsibility to avoid direct coupling between two or more classes?

Solution: Assign responsibility to an intermediary class to mediate between other classes.

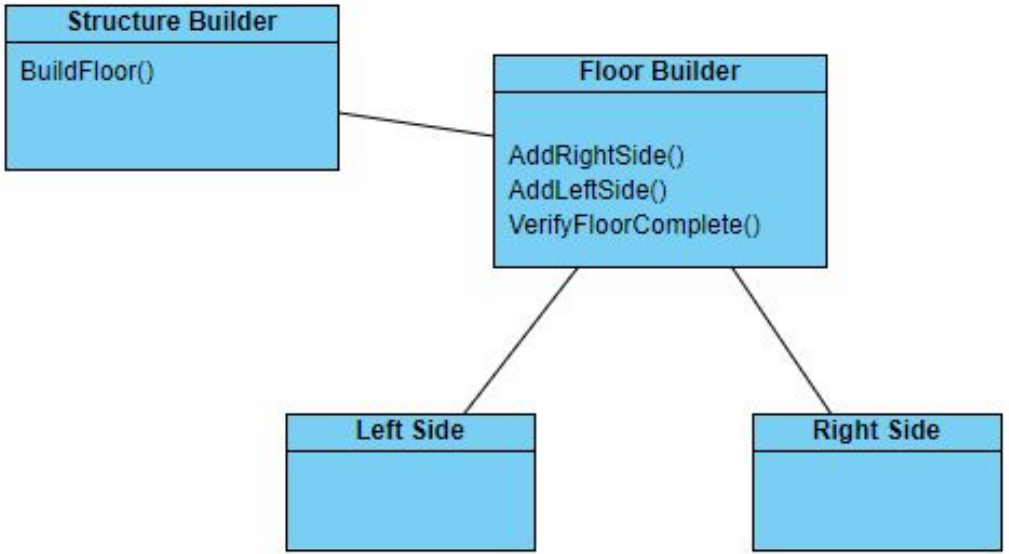
Indirection - Example

High Coupling



Indirection - Example (cont.)

Indirection: Lower Coupling



Coding Standards, Why?

They provide a guide for developers to format their code to improve consistency and readability.

General Naming Rules

Names should clearly convey the purpose of the item being named. Developers should avoid using abbreviations and acronyms that may not be immediately recognizable to others. As stated in the Google C++ Style guide, “…an abbreviation is probably OK if it's listed in Wikipedia.”

```
int fooBarBaz;     //Variable name is not descriptive.

void getFRQ();     //Function name uses an uncommon acronym.

class LvlLdr{}     //Class name uses unnecessary abbreviations.
```

```
int playerSpeed;  //Variable name is descriptive.

void getHTTPResponse(); //Function name uses a well known acronym.

class LevelLoader{}     //Class name is fully spelled out
```

Naming Classes

Class names should adhere to the general naming conventions described in the previous slide and should be formatted in PascalCase.

For interfaces, names should begin with the letter "I." The code below illustrates examples of appropriate class names.

```
/* Class Names */

class GameManager{}     //Class name uses PascalCase

class Player{}    //One-word class name is capitalized

interface ISampleInterface{}  //Interface name begins with 'I'
```

Naming Functions

```
/* Function Names and Parameters */

int findArea(int length, int width);  //Example using camelCase

void exit();       //One-word function name is lowercase

virtual int v_getHealth();     //Example of a virtual function

override int v_getHealth();    //Example of an override function
```

Function names and parameters should conform to the general naming rules specified previously, using camelCase for their formatting.

Additionally, names for virtual and override functions should start with the prefix "v_."

Naming Public, Private and Protected Variables

Public variables should be named using camelCase.

Private variables should begin with an underscore, followed by camelCase fo the remainder of the name. If a private variable is

Protected variables should begin with the p_ prefix and then use camelCase for the rest of the name.

If a variable is static, it should always start with the "s" prefix, prioritized over other prefixes.

```
/* Public Variable Names */

public int publicVariable;     //Public variable example

public GameObject player;      //One-word public variable example

public static int sStaticVariable;  //Public static variable example
/* Private Variable Names */

private int _privateVariable; //Private variable example

private float _speed;    //One-word private variable example

private static s staticVariable;     //Static private variable example
/* Protected Variable Names */

protected int p_protectedVariable;  //Protected variable example

protected int p_damage; //One-word protected variable example

protected static int sp_staticVariable;   //Static protected variable example
```

Constant Variables

Constant variable names should be written in all uppercase letters, with underscores separating each word.

```
/* Constant Variable Names */

const int CONSTANT_VARIABLE;  //Constant variable example
```

File Names and Exceptions

```
/* File Names */

file_name.txt

high_score.txt

config.json
```

While following the general naming rules, file names should be all lowercase with underscores between each word.

Naming exceptions can be made if a variable has a common meaning or purpose this name doesn't need to follow the naming standards. For example, the variables used for iterating in a for loop.

General Layout Rules

Indentations should always be 4 spaces (a tab).

Statements that involve curly braces, such as functions, classes, conditional statements and loops, the following rules apply:

1. The opening brace should be placed on the line below the statement.

2. The closing brace should be positioned on the line after the last line of the statement's content.

3. All content within a statement should be enclosed between the opening and closing braces, with no other content on the same line as either brace.

```
/* General Layout Example */

Statement
{
    /* Place all statement content here */
}
```

Class Layout

Classes should be designed to minimize coupling and maximize cohesion. When creating classes, ensure that the content within a class is related (high cohesion) and that only essential data is able to be passed between objects (low coupling).

Classes should follow the general layout rules, and should be laid out in this order:

1. Class data members, arranged from most to least visible (i.e., public, protected, private).
2. The Unity Start() method (if applicable).
3. The Unity Update() method (if applicable).
4. Other Unity methods, such as Awake() or OnCollisionEnter2D().
5. Any custom functions.

Additionally, each distinct element within a class should be separated by an empty line.

```
/* Class Layout Example */

public class ExampleClass : Examples
{
      public int publicNumber;
      public GameObject player;

      protected float p_protectedNumber;
      protected bool p_isPlayerDead;

      private int _privateNumber
      private float _playerHealth

     void Start()
     {
           /*Code to run on first frame*/
     }

     void Update()
     {
           /*Code to run every frame*/
     }

     void OnCollisionEnter2D(Collision2D collide)
     {
           /*Code to run on collision*/
     }

     public exampleClassMethod()
     {
           /*Example Class Method Code*/
```

Function Layout

```
/* Function Layout Example */

public bool isCoprime(int number1, int number2)
{
	int temporaryNumber;

	while(number2 != 0)
	{
		temporaryNumber = number1;
		number1 = number2;
		number2 = temporaryNumber % number2
	}

	if(number1 != 1)
	{
		return false;
	}
	else
	{
		return true;
	}
}
```

Functions will follow the general layout rules, parameters for the functions will be separated by a comma and space. Functions variable definitions should be placed at the beginning of the function followed by an empty line.

Functions should be designed to minimize coupling and maximize cohesion. When writing a function, keep it concise and focused on a specific task (high cohesion). If a function requires parameters, make sure that only the required data is passed in (low coupling).

If-Statement Layout

```
/* If-Statement -- Seperating variables and operators */

if(number1 <= number2)
{
   /*If-statement code*/
}
```

```
/* If-Statement -- Grouping operations */

if(number1 != (number2 * number3))
{
   /*If-statement code*/
}
```

```
/* If-Statement -- Grouping comparisons */

if((number1 < number2) && (number2 > 0))
{
   /*If-statement code*/
}
```

Readable and easy to understand.

1. Space between all variables and operators.

2. Operations needing to take place should be placed inside parentheses.

3. Multiple comparisons should be placed inside parentheses.

If-Statement Layout (cont.)

```
/* If-Statement -- More than 2 comparisons */

if((number1 > number2) ||
(number1 > number3) &&
(numebr1 != 0))
{
    /*If-statement code*/
}
```

4. More than 2 comparisons, each comparison needs to be placed on a separate line with the operator at the end of the line

```
/* Else if/Else Statements */

if(number1 > number2)
{
   /*If-statement code*/
}
else if(number1 >= 0)
{
   /*Else-if statement code*/
}
else
{
   /*Else statement code*/
}
```

5. If more than 3 else if statements are needed consider using a switch statement instead.

Switch Statement Layout

Follow the general layout rules like always.

When writing a switch statement make sure to include a default statement.

```
/* Switch Statement Example */

switch(switchVariable)
{
      case 1:
            /*Case 1 code*/
            break;
      case 2:
            /*Case 2 code*/
            break;
      case 3:
            /*Case 3 code*/
            break;
      case 4:
            /*Case 4 code*/
            break;
      case 5:
            /*Case 5 code*/
            break;
      default:
            /*Default code*/
            break;
}
```

While and Do-While Loop Layout

The same layout rules are required for while and do-while loops, the only additional key is that they must be able to escape their loop.

```
/* While Loop Example */

Random randomNumber = new Random();
while(loopFlag == true)
{
      randomNumber.Next();
      if((randomNumber % 2) == 0)
      {
            loopFlag == false;      //Can also use 'break' to exit loop
      }
}

/* Do-While Loop Example */
do
{
      randomNumber.Next();
      if((randomNumber % 2) == 0)
      {
            loopFlag == false;      //Can also use 'break' to exit loop
      }
} while(loopFlag == true);
```

For Loop Layout and Nesting Statements

```
/* For Loop Example */

for(int i = 0; i < 5; i++)
{
      /*For loop code*/
}
```

For loops should follow the general layout rules.When defining the initialization, condition, and post-loop housekeeping, use spaces between variables and operators, except when using the `++` or `--` operators.

Anything beyond 4 levels deep of nesting should be avoided if at all possible.

Code Comments - Multi-line Comments

Comments play a crucial role in communicating ideas to other developers. When writing comments, prioritize explaining ***why*** a section of code exists rather than simply stating ***what*** it does. Include comments whenever there are complex elements that require additional clarification, such as intricate if statements or REGEX expressions.

For comments written on a new line, use the `/* Comment */` syntax. If a comment spans multiple lines, each new line should begin with an asterisk. Put each idea within the comment on its own line for better readability.

```
/* Multi-line Comment Example */

/* This is an example of a Multi-line comment
 * Make sure to place new ideas on seperate lines
 * Begin new lines with an asterisk and a space, making sure they line up */
```

In-line Comments

If inline comments are used, they should be concise and to the point, using the `//` syntax. When multiple inline comments appear in a row, ensure that they are properly aligned for clarity.

```
/* Inline Comment Example */

private _foo;                  //Foo variable to give example
static private s_fooBarBaz;    //Variable to add more to the example
public testVariableOne;        //Notice that comments in this block lines up
```

Function Headings

```
/* Function Heading Example */

/* This function is here to identify whether two numbers are coprime
 * It applies the Euclidean Algorithm to find the GCD
 * When writing function headings, make sure to line up the asterisks
 * Make sure to note how this function interacts with other functions */
public bool isCoprime(int number1, int number2)
{
      /*Function code*/
}
```

Each function in your code should be preceded by a function header comment. This comment should provide a high-level overview of the function and explain how it interacts with other functions, variables, and classes.

File Heading

At the beginning of each script, there should be a file header that describes the purpose of the file. This comment must include the developer's name, role, project name, and class relationships like inheritance that are in the file.

```
/* File Heading Example */

/* Name: Shawn Young
 * Role: Team Lead 4 -- Project Manager
 * Project: Shipwreck Protocol
 *
 * This file contains the definition for the ExampleClass
 * This class provides an example for Nautilus's Coding Standards
 * It inherits from Examples and IExamplesInterface
 * This File Heading would also be a good place to mention design patterns */

public class ExampleClass : Examples, IExampleInterface
{
        /*Class Definition*/
}
```

Let the Code Roasting Begin!

Just kidding...

TL4’s didn’t do coding standards before we started working on projects. You all should be roasting us.

Gantt Roasts

Team 5 Guys

12-Sep					
Group RFP Prep	SA Prep	SA Prep			
	1.5	2			
✓	✓	✓			
		✓			
✓		✓			
✓		✓			
✓	✓	✓			
0	3	10	0	0	0

Design Maps
Programming
User Documentation
Testing

Setting up Godot	1
Godot Training	2
Godot Pong	3
Platformer|Outside Meetings	4
Requirements Collection	5

Tappa Tappa Keyboard

0 Collect Sound Assets	1													
1 Main Sound Controller	2				0									
2 Player Sound Controller	2						1							
3 Hazard Sound Controller	4						1							
4 HUD Sound Controller	2						1							
5 Testing	2										2,3,4			
6 Installation	2												5	
Total	15													

WE'VE MET BEFORE

TRUST ME

Requirements Collection
Sprite Design
Controller Design
Level Creat
Programmir
Testing
Installation
Total

Requirements C
Asset Collection
Tileset Construc
Rising Lava
Level Piece Des
Level Procedura
Enemy and Collectible Integration
Testing
Total

0 Collect Sound Assets
1 Main Sound Con
2 Player Sound Cc
3 Hazard Sound C
4 HUD Sound Con
5 Testing
6 Installation
Total

Research AI patterr
Design Basic Enem
Implement Pathfind
Create State Machi
Program Enemy Att
Develop Collision D
Integrate AI with lev
Test and Debug En
Optimize Performance of AI Code
Final Review and Adjusments
Total

Make Interface/HUD Theme and Create Assets
Make Menu Theme and Elements
Create Player HUD
Create Game HUD
Create Shop HUD
Create Scene Camera
Create Settings Menu
Create Character Creator
Create Main Menu/Pause Menu
Create Mobile Hud
Create Death Screen
Total

Patterns where?

Make It Awesome

	Date				
	Purpose	RFP	SA Presentation		
	Hours				
Amara					
Andreas					
Caden					
Elizabeth					
Joe					
Total					

predicted time (hrs)	actual time (hrs)	status	1	2	3
4		complete			
8		in progress			
6		scheduled			

Amara
Generate Items
Program functionality
Level Win Conditions
Incorporate Music
Animations
Testing
Totals:
Andreas
Complete P/G Chart
Complete Champion Doc
Design Level Layouts
Implement Worm Controls
Complete MVP
Expand Levels
Interactable Game Manager
Collect Map Assets
Create Testing Suite
Integration
Totals:
Caden
Enemy Design
Asset Creation
Basic Pathing
Player Interactions
Environment Collision
Testing
Installation
Totals:

Penguine

Implement Play SFX Functionality	2		planned
Character Touchup			planned
Integration	3		planned

Carson	
Scene Transitions	3
Overworld Test Panel	2
Room Navigation	3
Overworld Management System	6
Room/Overworld Design	3
Overworld Character	
Animations/GFX	5
End Screen Design	2
Integration	3
Testing	3
totals	30
Emily	
Dialogue Prompts	6
Character Designs Selected	1
Character Completed	6
Dialogue Prompts Implemented	7
Character Completed	4
Difficulty Level Character Assignment	4
Character Completed	4
Integration	4
Testing	3

TL 1 Deliverables	TL 2 Deliverables	TL 3 Deliverables
Date Due: 11/27	Date Due: 11/27	Date Due: 11/27

	Task	Predicted(hrs)	spent(hrs)
Amanda	Individual schedule		
	Champion		
	SA AI Slides		
	SA Global Class Diagram		
	SA Character Slides		
	Subtotal	0	0
Carla	Individual schedule		
	Champion		
	SA Character Slides		
	SA Global Class Diagram		
	Subtotal	0	0
Carson	Individual schedule		
	Champion		
	SA World		

Select Graphics and Sound for each LI	2	1	this week
Implement Dialogue Prompt Display and Reaction to User Input (Affection Point Update / Dialogue Sequence	5	2	this week
Set up minigame Date Initiation	2	1	this week
Implement Different Difficulty Levels for the Characters	2		planned
Implement Play SFX Functionality	2		planned
Character Touchup			planned
Integration	3		planned
Testing	3		planned
totals	25	8	
Carson			
Scene Transitions	3	3	complete
Overworld Test Panel	2	2	complete
Room Navigation	3		this week
Overworld Management System	6		planned
Room/Overworld Design	3		planned
Overworld Character Animations/GFX	5		planned
End Screen Design	2		planned
Integration	3		planned
Testing	3		planned
totals	30	5	
Emily			
Dialogue Prompts	6	4	this week
Character Designs Selected	1	1	complete
Character Completed	6	2	this week
Dialogue Prompts Implemented	7		planned
Character Completed	4		planned
Difficulty Level Character Assignment	4		planned
Character Completed	4		planned
Integration	4		planned
Testing	3		planned
Touchups	3		planned
totals	42	7	
Lance			
Requirements Collection	3	3	complete
Minigame Level Designs	4	6	this week
Pong Minigame Programming	2	4	complete
Math Minigame Programming	5	2	this week

Nautilus

> Plans are of little importance, but planning is essential.
>
> – Winston Churchill

Shawn		?	
Art Decisions	2		complete
Item Programming	5		this week
Collectables Programming	2		planned
Weapon Item Design	2		planned
Pick Music/Sounds effects	3		planned
Make Sound Manager	6		planned
Testing	5		planned
Documentation	2		planned
totals	27	0	

Gantt Chart made 3 days ago??? ✓
No patterns! ✓

Ben	Alice
complete	Complete
complete	
complete	Complete
complete	Complete
complete	Complete
this week	this week
planned	This Week
planned	planned
planned	planned
	planned
Davin	planned
complete	planned
complete	planned
complete	planned
complete	planned
complete	
planned	Kenny
this week	Complete
this week	Complete
planned	Complete
	Complete
	This Week
	This Week
	planned

Example: Management Summary | Gantt | Team Mate Delverables | Meetings | SA | Overhead

Shawns: Sheet1

Requirements Colle	5		this week
Screen Design	6		planned
Report Design	6		planned
Database Construction	2		planned
User Documentation	6		planned
Programming	5		planned
Testing	3		planned
Installation	1		planned
totals	34	0	

MILCCN

		23	4	
Milestone1	Created Room1 background layers	1	1	Done
Milestone1	inserted layers in unity and add barriers for collosions	2	2	Done
Milestone1	worked on gannt chart	1	1	Done
Milestone2	Room super class	4	4	Done
Milestone2	entry room design	2		Working
Milestone2	exit room design	2		Scheduled
Milestone2	boss room design	1		Scheduled
Milestone2	shop room design	1		Scheduled
Milestone2	additional room design	2		Scheduled
Milestone2	Grombo load in functions	4		Scheduled
Milestone2	Enenmy Load in functions	4		Scheduled
Milestone2	Item load in functions	4		Scheduled
Milestone3	Code Cleanup	2		Scheduled
Milestone3	Final Tests	2		Scheduled
Milestone3				

So many colors, so little work

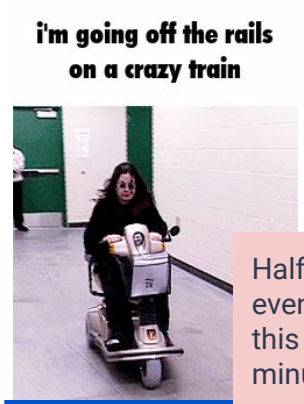
Me looking for the patterns and dynamic binding work items

+ ☰ 3 Gannt Meetings

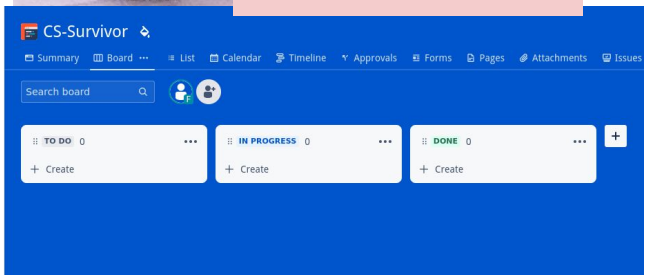
Created basic player movement script	1	2	Done
Created place holder art for player character	1	1	Done
Created basic health script	1	1	Done
Player Attack Design	5		Working
Player Sprite Animations	2		Working
Player Interact ability	2		Scheduled
Inventory	3		Scheduled
item Compatibilty testing	4		Scheduled
Code Cleanup	2		
Final Tests	2		

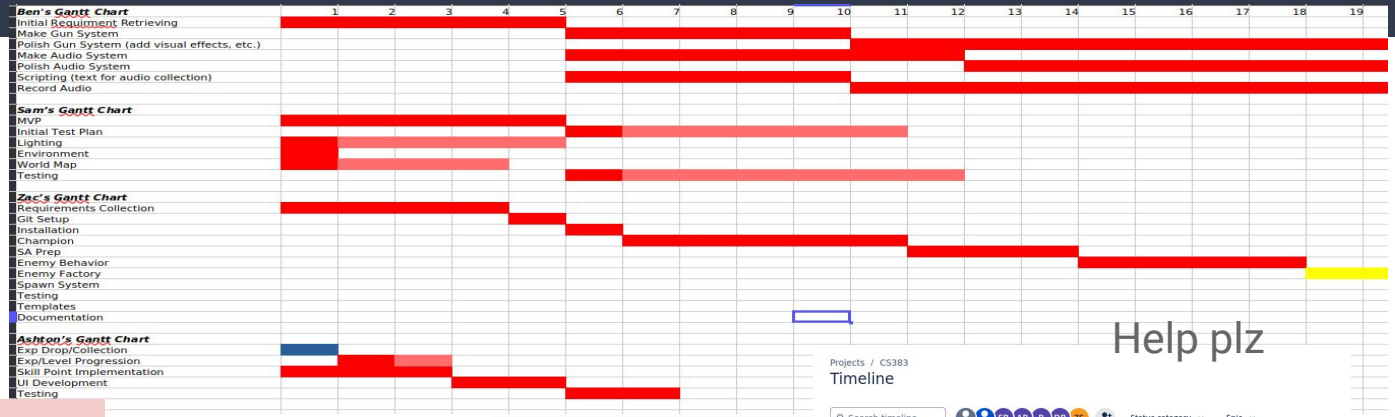
Teal Team 6

Extremely violent colors



Half of the time isn't even tracked we drew this up in like 30 minutes last night





Help plz

What the heck is a jira and why are there so many people in it