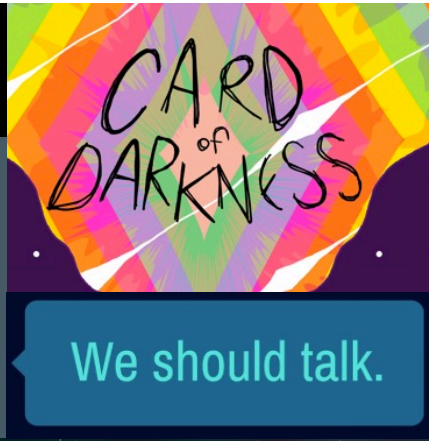
Advanced Gameplay Programming

Week 1

Who am I?


EXPLODING KITTENS
PASS THE BUCK
DOMINION
PROSPERITY
LEADERS


NEONIMO
THE NEW SHAPE OF STRATEGY
Good Sudoku
by zach gage
and jack schlesinger
Play
Good Puzzles
Beginner
Advanced
Expert
Pro
Impossible
Back
Good Sudoku
by zach gage
and jack schlesinger
Improve
Techniques
Hidden Pair (Split)
X-Wing
Naked Triple
Hidden Triple
Hidden Triple (Split)
Naked Quadruple
Y Wing
Prev
Next
Back
Good Sudoku
Daily Good Sudoku
Good Puzzle 7/22/2020 - Play
Classic
Arcade
Back


CARD of DARKNESS
We should talk.


SpellTower+
a game by zach gage
code by jack schlesinger


ONE HIT WONDER

Learning Goals of this Course

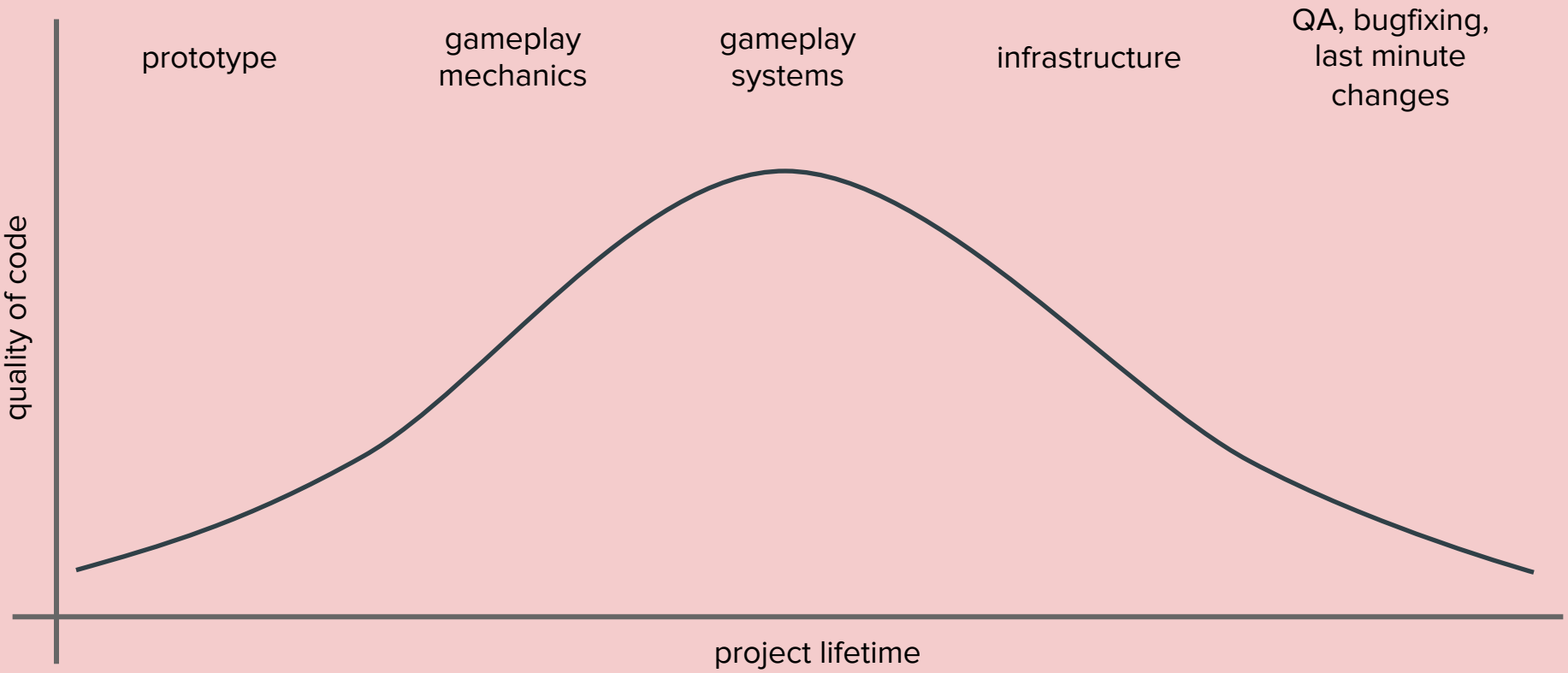
Robust and flexible systems that thrive in a playtesting-focused environment

Support the development of a game over it's entire lifecycle

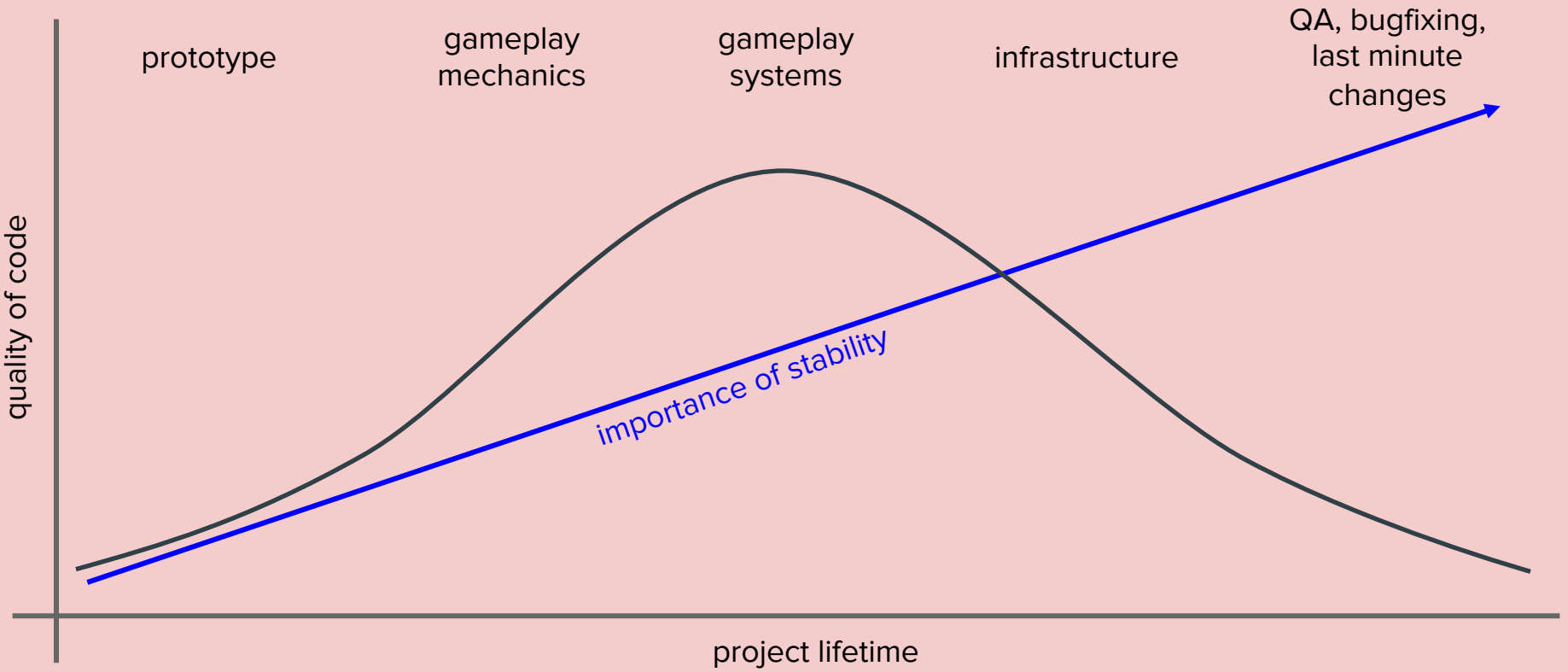
Architecture to reduce tech debt and use resources efficiently

Meet platform requirements and get a game ready for release

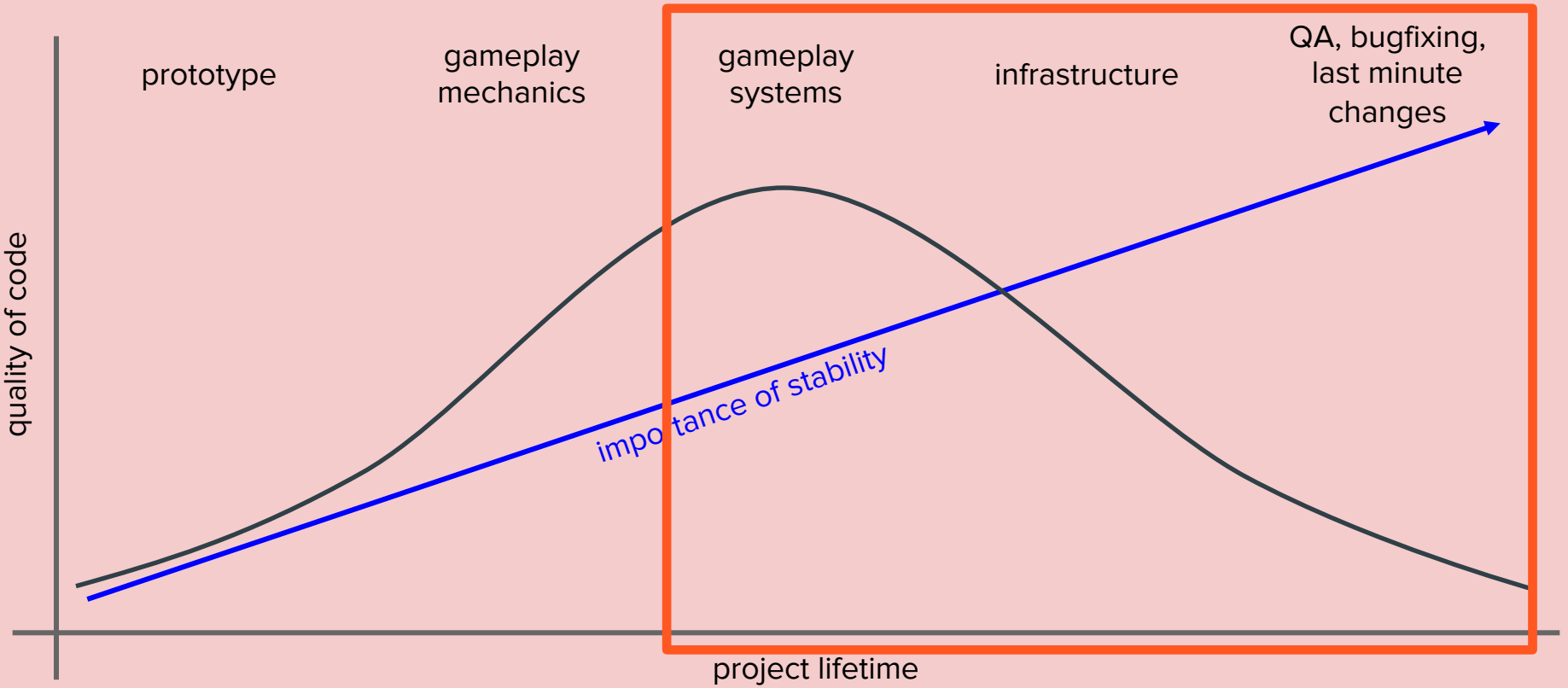
Games Over Time



Games Over Time



Games Over Time



Programming and Pattern Bootcamp (3-4 weeks)

Procedural Logic

Procedural Logic

Sub-Engine Programming

Making this together.

syllabus time

Break time

Gameplay Programming is deeply collaborative.

Playtesters ...

Playtesters ...

Designers ...

Playtesters …
Designers …
Programmers …

Playtesters ...

Designers ...

Programmers ...

...even yourself

Expressive

Consistent

Comprehensible

Clear

Clean

Without being easily understood, your code will be challenging to collaborate on.

Systemic

Single Responsibility

Clear Functionality

Simple Access

Low Exposure

Well encapsulated

Without being organized into systems, it's difficult for collaborators to own their domain.

Flexible

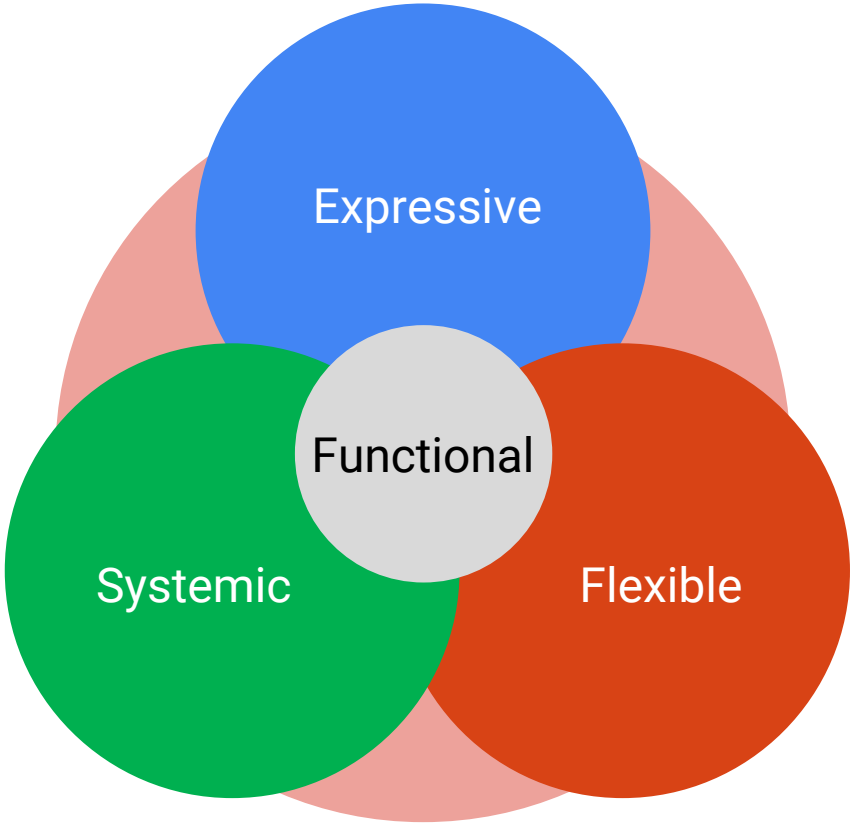
Iteratable

Reusable

Extensible

Dispensible

Without being flexible, other designers won't be able to steer the ship.


Expressive
Functional
Systemic
Flexible


Flexible
My Code
Systemic
Expressive

Coding Guidelines for Collaborative Code

DO Use Explicit Access Modifiers

Public / Private / Protected

Avoid implicit access modifiers

Use properties when you want to allow read-only access to a variable.

```
public int exampleInteger { get; private set; }

private string _exampleString;
public string exampleString {
    get { return _exampleString; }
}
```

DON'T Use Abbreviations

```
public float myNum;                    // don't abbreviate
public float myNumber;                 // vague
public float acceleration;             // good
public float currentAcceleration;      // better
```

DON'T Name on Implementation

```
// Don't name your functions based on implementation
public int ComputeVectorOfDamageMultiplied(Enemy attackingEnemy) { }
```

DO Name on Functionality

```
// Name your functions based on functionality
public int DamageAmount(Enemy attackingEnemy)
```

DON'T Use Piles of Primitives

```
private float _playerXPosition, _playerYPosition, _playerZPosition;
private float _playerXRotation, _playerYRotation, _playerZRotation;
private float _playerXSpeed, _playerYSpeed, _playerZSpeed;
private float _playerXAcceleration, _playerYAcceleration, _playerZAcceleration;
private bool _playerLockedInPlace;
private int _playerRefreshRate;
private float _playerXTopSpeed, _playerYTopSpeed, _playerZTopSpeed;
```

DO Use Composite Data Types

```
// ============== Instead, do this: ============== //

private struct AxisInformation {
    public float position, rotation, speed, topSpeed, acceleration;
}

private struct PhysicsInformation {
    public AxisInformation x, y, z;
    public bool lockedInPlace;
    public int refreshRate;

}

private PhysicsInformation player;

player.x.position += player.x.speed;
```

Why Use Composite Data Types?

The alternative is…

- **Difficult to read.**
- **Confusing to maintain or make changes.**
- **No ability to reuse structure.**

DO Use Return Values

```
// This is confusing, and not clear
if (currentSkill.timeOut <= 0 && currentSkill.chosen) {
    ...
}
```

```
// Instead:
if (IsAvailable(currentSkill)) {
    ...
}

...

public bool IsAvailable(Skill skillToCheck)
{
    return (skillToCheck.available <= 0 && skillToCheck.chosen);
}
```

Why Use Return Values?

The alternative is…

- **Unclear**
- **Hard to reuse**
- **Inflexible**

Expressive Code Explains It's Own Logic

```
player.x.position += player.x.speed;
```

DON'T use "magic numbers"

```
// Includes a "magic number"
if (Input.GetKeyDown(KeyCode.RightArrow))
{
    player.transform.position += 0.5f * Vector2.right;
}
```

DO Use Tuning Variables

```
// Instead, use a global tuning variable
private const float PlayerSpeed = 0.5f;

...

if (Input.GetKeyDown(KeyCode.RightArrow))
{
    player.transform.position += PlayerSpeed * Vector2.right;
}
```

DON'T Repeat Code

```
public void AttackClose()
{
    foreach (var tile in adjacentTiles.Distance(1))
    {
        Attack(tile);
    }
}

public void AttackTwoAway()
{
    foreach (var tile in adjacentTiles.Distance(1))
    {
        Attack(tile);
    }

    foreach (var tile in adjacentTiles.Distance(2))
    {
        Attack(tile);
    }
}
```

DO Use Parameterized Functions

```
public void AttackTiles(int distanceToAttack)
{
    for (var i = 0; i < distanceToAttack; i++)
    {
        foreach (var tile in adjacentTiles.Distance(i + 1))
        {
            Attack(tile);
        }
    }
}
```

DO Fall Out Of Functions Early (Pt. 1)

```
public void AddSpell(Player player, Spell spell) {

    var spellBook = player.getSpellBook();

    if (spellBook !== null) {
        if (spellBook.freeSlots > 0 && player.intelligence >= spell.level) {
            var playerSchool = player.getMagicSchool();
            if (playerSchool !== null && spell.school.name !== playerSchool.name) {
                spellBook.addSpell(spell);
            }
        }
    }
}
```

DO Fall Out Of Functions Early (Pt. 2)

```
public void AddSpell(Player player, Spell spell) {

    // check all your requirements in individual lines
    // and "early out" if any fail

    var spellBook = player.getSpellBook();

    if (spellBook === null) return Error("Player has no spellbook");

    if (spellBook.freeSlots === 0) return Error("Player spellbook has no free slots");

    if (player.intelligence < spell.level) return Error("Player intelligence too low");

    var playerSchool = player.getMagicSchool();

    if (playerSchool === null) return Error("Player has no magic school");

    if (spell.school.name !== playerSchool.name) return Error("Incompatible schools")

    spellBook.addSpell(spell);

}
```

DO Fall Out Of Functions Early (Pt. 3)

```
public void Attack() {
    if (holdingWeapon)
    {
        PlayAttackSound();
        SwingBlade();
        DamageEnemy();
        ...
    }
}
```

```
public void Attack() {
    if (!holdingWeapon) return;

    PlayAttackSound();
    SwingBlade();
    DamageEnemy();
    ...
}
```

Flexible code makes iterative design possible.

```
private const float PlayerSpeed = 0.5f;
```

DO Use Single Responsibility Functions

- “Simple Sentence Test”
- Refactor complex functions into smaller functions
- Keep functions short

DO Create a Prototyping Kit

- Common engine-specific extension methods
- Common engine-specific systems
  - Audio
  - Integration w/specific engine functionalities
  - Animation
- Common game design specific classes
  - Shufflebag
  - Easing
  - Most everything in Gameplay Programming Patterns

Systemic code travels with you from project to project.

```
private const float PlayerSpeed = 0.5f;
```

What should you focus on in this class?

Do’s

Write your code with your own coding standards and stick with them

Don’ts

What should you focus on in this class?

Do’s

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don’ts

What should you focus on in this class?

Do’s

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don’ts

Don’t try to make your code perfect

What should you focus on in this class?

Do’s

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don’ts

Don’t try to make your code perfect

Don’t try to do exactly the coding standards I set out.

What should you focus on in this class?

Do's

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don'ts

Don't try to make your code perfect

Don't try to do exactly the coding standards I set out.

Don't do more than 8 hours of work a week

What should you focus on in this class?

Do's

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don'ts

Don't try to make your code perfect

Don't try to do exactly the coding standards I set out.

Don't do more than 8 hours of work a week

Don't set out to create anything "one-size-fits-all" *

* But keep it when you do.

What should you focus on in this class?

Do's

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Don'ts

Don't try to make your code perfect

Don't try to do exactly the coding standards I set out.

Don't do more than 8 hours of work a week

Don't set out to create anything "one-size-fits-all" *

Don't make anything that looks good

* But keep it when you do.

What should you focus on in this class?

Do's

Write your code with your own coding standards and stick with them

Tell me what you want to learn

Focus on making your code:

- Expressive
- Systemic
- Flexible

Don'ts

Don't try to make your code perfect

Don't try to do exactly the coding standards I set out.

Don't do more than 8 hours of work a week

Don't set out to create anything "one-size-fits-all" *

Don't make anything that looks good

* But keep it when you do.

Keeping Track of Things

Services Locator

A phone book for your project.

Purpose: The services locator is a central location for core parts of your game

- Easy place to find them
- Easy place to replace and change them

Why use it?

- Singletons are bad.*
- GameObject.Find(“Name of Object”) is bad.**

---

*Singletons may not always be bad.
** Searching for a string literal is always bad

What does “**static**” mean?

Why is a static class / singleton bad?

- A public static class or singleton is like telling everyone you meet your home address in order to get letters
- The Services Locator is like a phone book
- Singletons give you two things -> accessibility, and uniqueness

Enter the Services Locator

- Separates code that needs a service from two things:
  - Who the service is (the concrete implementation type)
  - Where it is (how we get to the instance of it).
- Provides:
  - access to services/systems by managing a set of references to those systems.
  - control over initialization and lifecycle of the services.
  - a level of indirection between services and their users.

Example Services Locator

```
public static class Services
{
    public static AudioSystem Audio;
    public static GameController Controller;
    public static EnemyManager Enemies;
}
```

Using Services Locator

```
public class GameController
{
    private void Awake()
    {
        Services.Controller = this;
        Services.Audio = new AudioSystem();
        Services.Enemies = new EnemyManager();

    }
}
```

```
  private void Update() {
      Services.Enemies.MoveTowardsPlayer();
  }
```

When do we use it?

- When singletons are too rigid
  - Want access to an EnemyController
  - Want a different EnemyController in each scene.
- Testing
  - Might be too complicated or even impossible to have full blown versions of all your systems loaded and initialized just for testing
- “Hot-swapping”
  - Changing systems at runtime can be useful for rapid development
  - Makes your game configurable without having to restart the application

Lifecycle Management

“Manager” Pattern

Purpose: Organization and optimization.

Why use it?

- GameObject.Find(“String Literal”) is bad.
- Separation of concerns: A gun shouldn’t need to know how to make a bullet in order to fire.

What is it?

A type that models a group (e.g. bullets) or collective property (e.g. entities that belong to a team) and supports management of one or more of the following aspects:

- **Lifecycle**: Control how/when managed instances are created, updated and destroyed.
- **Shared State/Context**: Provide access to shared state or context to managed instances.
- **Access**: Provide and control access to the instances.
- **Queries**: Allow queries across the managed instances (i.e. queries against a set of instances and not against individuals).

Practical Examples

Bullet Manager: Your bullet hell shooter emits lots of bullets, and you need a way to wrangle them as groups.

Monster Database: Your dungeon crawler has tons of monsters, are used all over the place and are expensive to create (complex models, large textures, etc).

Faction System: Your gentrification game has a lot of factions (e.g. "hipster", "yuppie", "parent", etc.) and a lot of objects that they can be mapped to.

Benefits

- Lowers complexity
- Decrease coupling
- Helps with memory management
- Sequencing

Homework Review