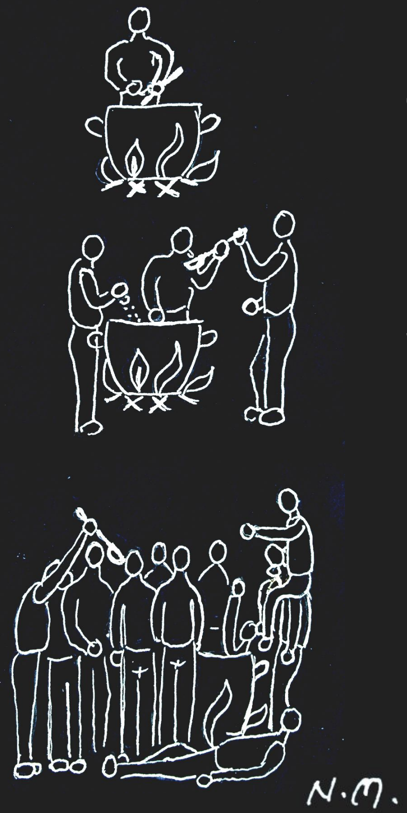
From Pong to 15-person project

Mikael Kalms
CTO @ Fall Damage
kalms@falldamagestudio.com

Effective strategies for handling project growth (with a focus on code)

Your daily struggle

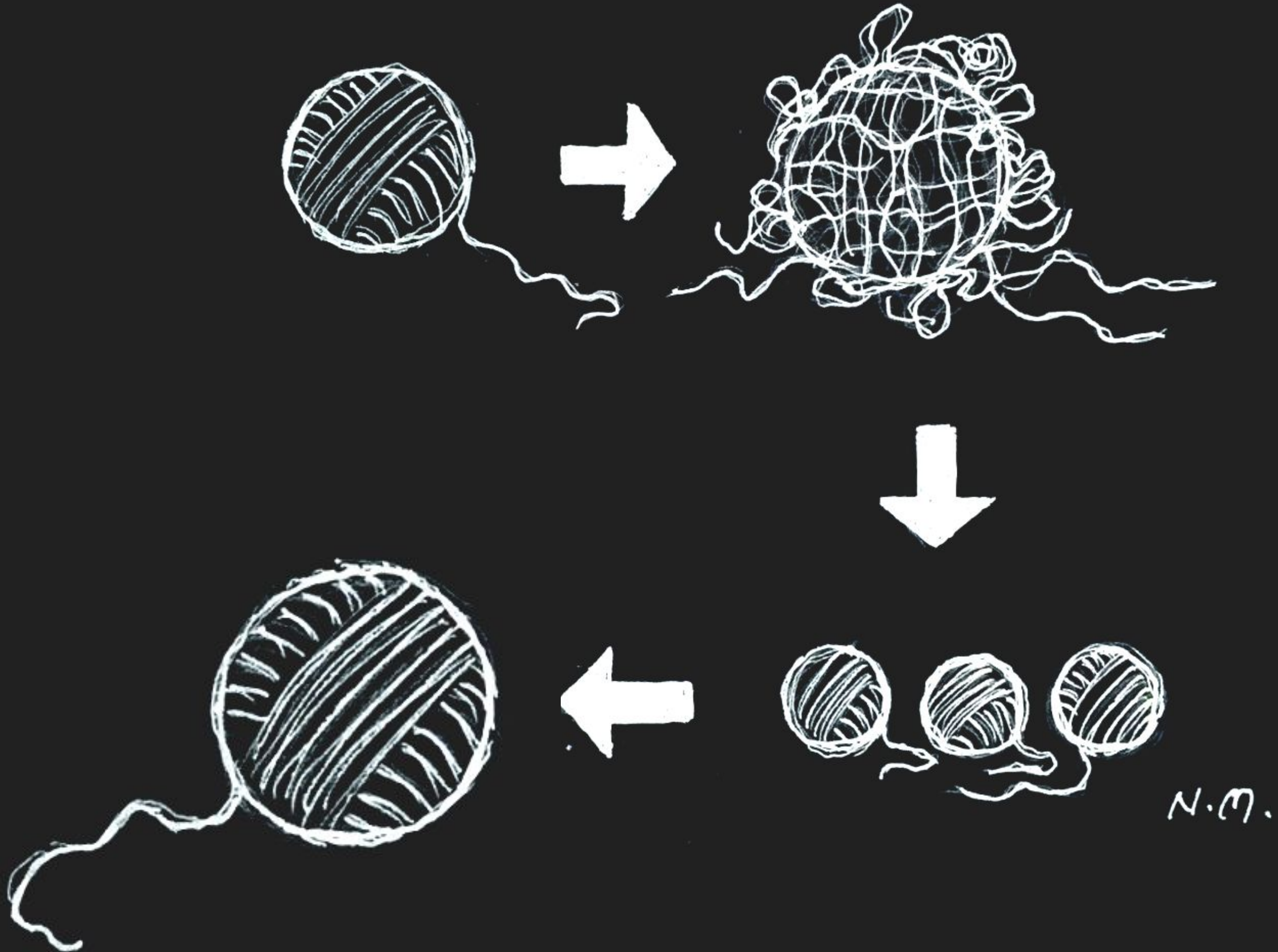


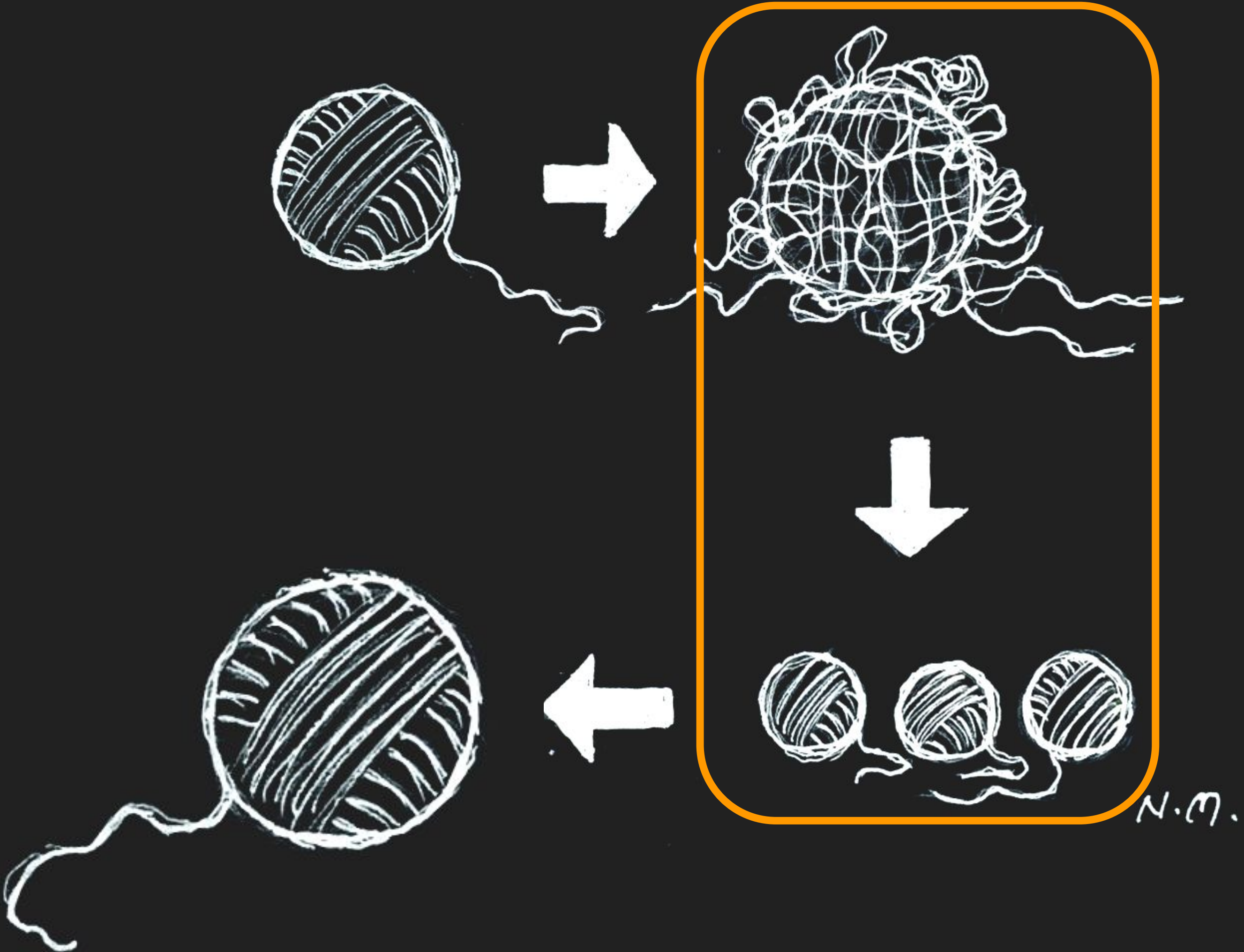
Organizing lines of code

Software architecture

Process automation

Organizing lines of code


N.M.


N.M.

Demo: Pong game

Tour of Unity project

Source: https://www.github.com/Kalmalyzer/Pong

Principles at work

One “thing” = one Prefab

Custom logic for one “thing” = one MonoBehaviour

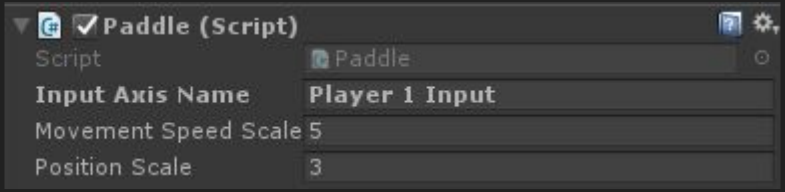
An application = a scene containing interlinked prefabs

This is a reasonable structure, for this project

When the project grows, the structure will also need to change.

Instances / Prefabs / ScriptableObjects

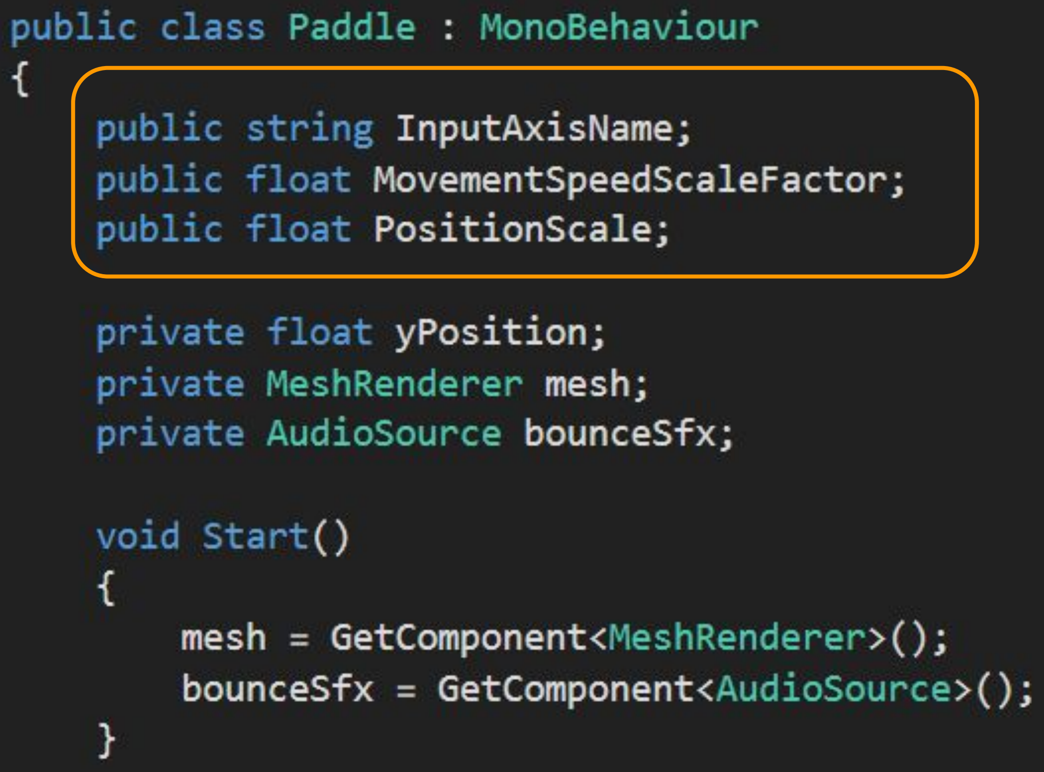
Change parameters in the prefab, or in the instance?



Paddle (Script)

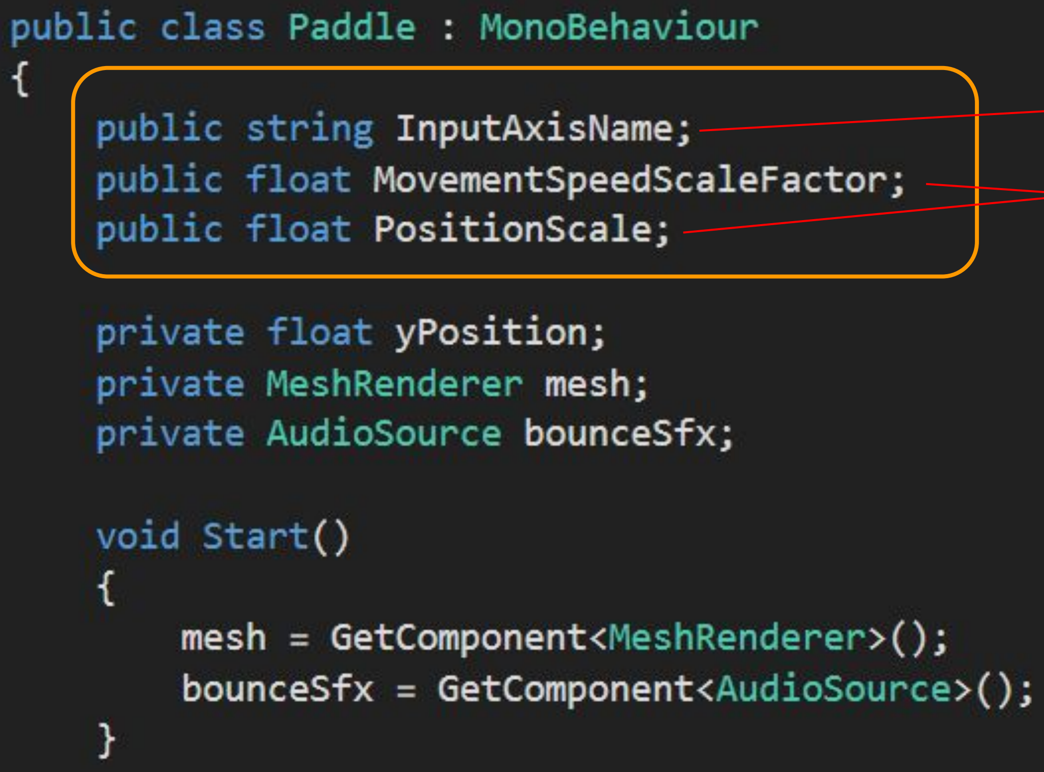
Script	Paddle
Input Axis Name	Player 1 Input
Movement Speed Scale	5
Position Scale	3

Change parameters in the prefab, or in the instance?



```
public class Paddle : MonoBehaviour
{
    public string InputAxisName;
    public float MovementSpeedScaleFactor;
    public float PositionScale;

    private float yPosition;
    private MeshRenderer mesh;
    private AudioSource bounceSfx;

    void Start()
    {
        mesh = GetComponent<MeshRenderer>();
        bounceSfx = GetComponent<AudioSource>();
    }
```

Different purposes



```
public class Paddle : MonoBehaviour
{
    public string InputAxisName;
    public float MovementSpeedScaleFactor;
    public float PositionScale;

    private float yPosition;
    private MeshRenderer mesh;
    private AudioSource bounceSfx;

    void Start()
    {
        mesh = GetComponent<MeshRenderer>();
        bounceSfx = GetComponent<AudioSource>();
    }
```

Customize for each player

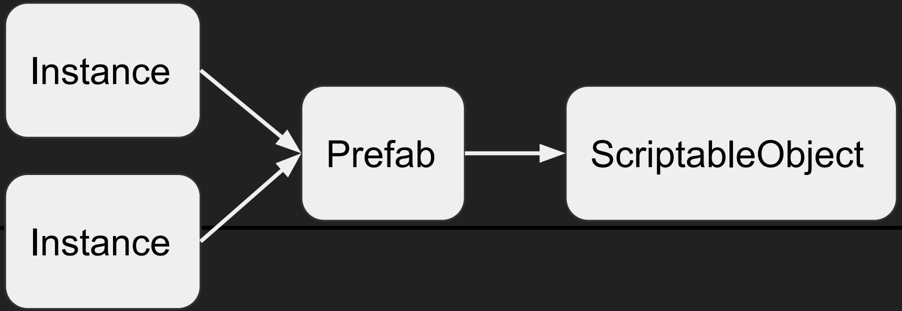
Shared for both players

Strategy: Instance -> Prefab -> ScriptableObject

Do you need something only once? Create a prefab, then instantiate.

Do you need something multiple times, possibly with some instance-specific modifications? Create a Prefab, then instantiate, then override some settings.

Do you want to ensure the same setting across multiple instances? Create a ScriptableObject, then source data from there instead.

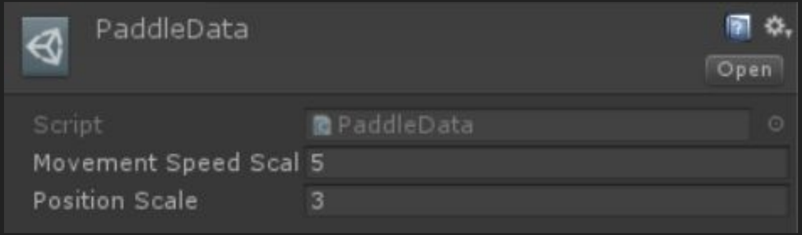


End result

```
public class PaddleData : ScriptableObject
{
    public float MovementSpeedScaleFactor;
    public float PositionScale;
}
```

```
public class Paddle : MonoBehaviour
{
    [Header("Static Data")]
    public PaddleData PaddleData;

    [Header("Customizable settings")]
    public string InputAxisName;

    private float yPosition;
    private MeshRenderer mesh;
    private AudioSource bounceSfx;

    void Start()
    {
        mesh = GetComponent<MeshRenderer>();
        bounceSfx = GetComponent<AudioSource>();
    }
```

End result

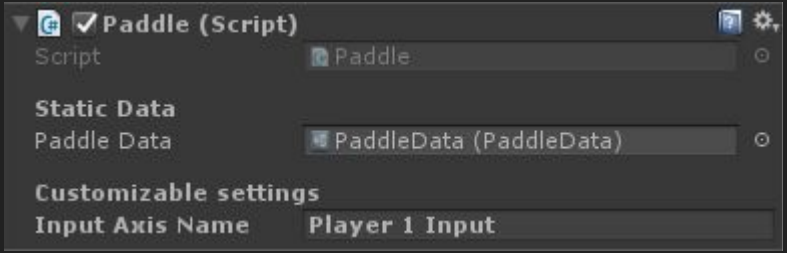


PaddleData

Script: PaddleData

Movement Speed Scal: 5

Position Scale: 3



Paddle (Script)

Script: Paddle

Static Data

Paddle Data: PaddleData (PaddleData)

Customizable settings

Input Axis Name: Player 1 Input

Splitting up large MonoBehaviours

Single Responsibility Principle

Each class should handle one single thing.

If done well, you can give a short description of the class which makes it clear

- What does this class do?
- What does this class *not* do?

This principle can be used regardless of the size of a program.

What does this do, anyway?

```csharp
public class Paddle : MonoBehaviour
{
    [Header("Static Data")]
    public PaddleData PaddleData;

    [Header("Customizable settings")]
    public string InputAxisName;

    private float yPosition;
    private MeshRenderer mesh;
    private AudioSource bounceSfx;

    void Start()
    {
        mesh = GetComponent<MeshRenderer>();
        bounceSfx = GetComponent<AudioSource>();
    }

    void Update()
    {
        float delta = Input.GetAxis(InputAxisName) * PaddleData.MovementSpeedScaleFactor * Time.deltaTime;
        yPosition = Mathf.Clamp(yPosition + delta, -1, 1);

        transform.position = new Vector3(transform.position.x, yPosition * PaddleData.PositionScale, transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        bounceSfx.Play();
    }
}
```

What does this do, anyway?

```
public class Paddle : MonoBehaviour
{
    [Header("Static Data")]
    public PaddleData PaddleData;

    [Header("Customizable settings")]
    public string InputAxisName;

    private float yPosition;
    private MeshRenderer mesh;
    private AudioSource bounceSfx;

    void Start()
    {
        mesh = GetComponent<MeshRenderer>();
        bounceSfx = GetComponent<AudioSource>();
    }

    void Update()
    {
        float delta = Input.GetAxis(InputAxisName) * PaddleData.MovementSpeedScaleFactor * Time.deltaTime;
        yPosition = Mathf.Clamp(yPosition + delta, -1, 1);

        transform.position = new Vector3(transform.position.x, yPosition * PaddleData.PositionScale, transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        bounceSfx.Play();
    }
}
```

Whole-game parameters

Per-player parameter

Automatic component-to-component linkage

Input handling

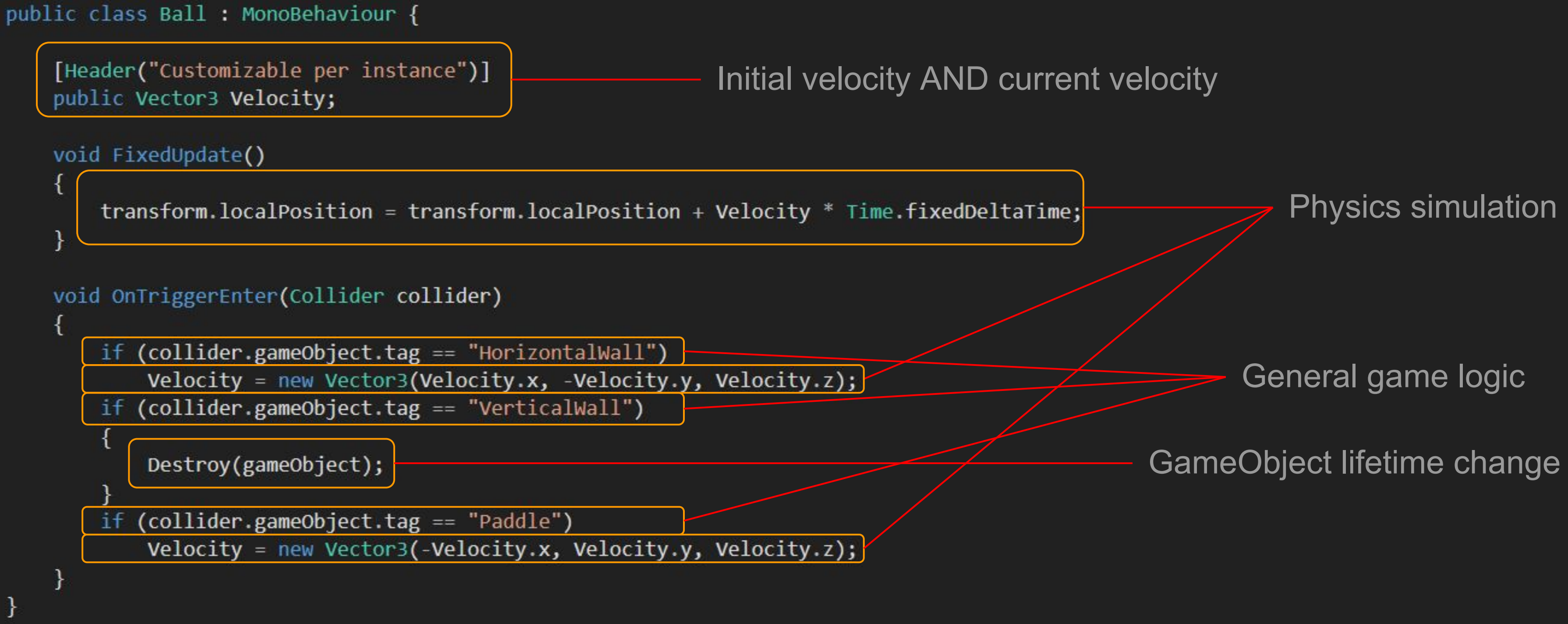
“physical” simulation

Audio triggering

What does this do, anyway?

```csharp
public class Ball : MonoBehaviour {

    [Header("Customizable per instance")]
    public Vector3 Velocity;

    void FixedUpdate()
    {
        transform.localPosition = transform.localPosition + Velocity * Time.fixedDeltaTime;
    }

    void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "HorizontalWall")
            Velocity = new Vector3(Velocity.x, -Velocity.y, Velocity.z);
        if (collider.gameObject.tag == "VerticalWall")
        {
            Destroy(gameObject);
        }
        if (collider.gameObject.tag == "Paddle")
            Velocity = new Vector3(-Velocity.x, Velocity.y, Velocity.z);
    }
}
```

What does this do, anyway?



```
public class Ball : MonoBehaviour {

    [Header("Customizable per instance")]
    public Vector3 Velocity;

    void FixedUpdate()
    {
        transform.localPosition = transform.localPosition + Velocity * Time.fixedDeltaTime;
    }

    void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.tag == "HorizontalWall")
            Velocity = new Vector3(Velocity.x, -Velocity.y, Velocity.z);
        if (collider.gameObject.tag == "VerticalWall")
        {
            Destroy(gameObject);
        }
        if (collider.gameObject.tag == "Paddle")
            Velocity = new Vector3(-Velocity.x, Velocity.y, Velocity.z);
    }
}
```

Initial velocity AND current velocity
Physics simulation
General game logic
GameObject lifetime change

Different types of responsibilities in our classes

General game logic

Input handling

Physics simulation

Presentation

Exposing parameters to editor/inspector

Unity engine event handlers

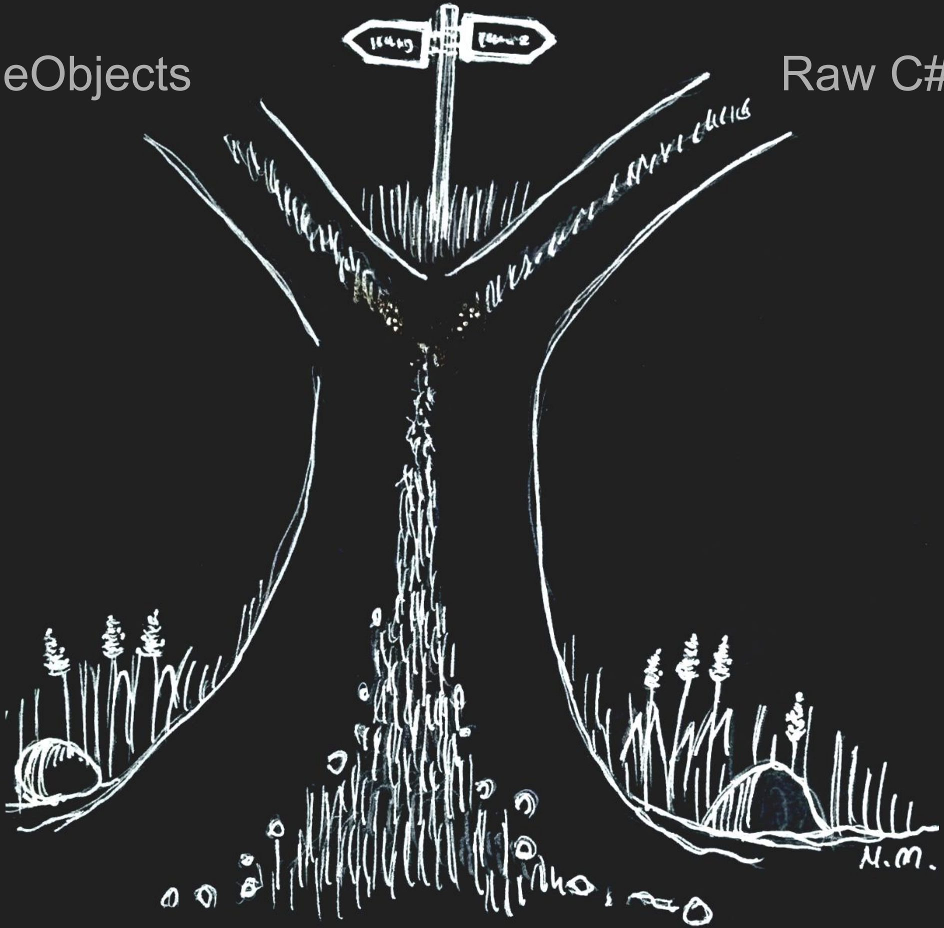
GameObject lifetime management

Different types of responsibilities in our classes

General game logic Input handling Physics simulation Presentation	Could reside within MonoBehaviours, ScriptableObjects, or raw C# classes
Exposing parameters to editor/inspector	Could reside within MonoBehaviours or ScriptableObjects
Unity engine event handlers GameObject lifetime management	Should reside within MonoBehaviours

ScriptableObjects

Raw C# classes



Moving logic MonoBehaviours -> ScriptableObjects

Great presentations:

Unite Europe 2016 - Overthrowing the MonoBehaviour tyranny in a glorious ScriptableObject revolution

Unite Austin 2017 - Game Architecture with Scriptable Objects

Allows using Unity Inspector to view, edit and link together your code & data

Moving logic MonoBehaviours -> regular C# classes

Regular C# classes have better language facilities than Unity’s own objects for breaking down code into small, composable chunks

Regular C# code can also be shared with native .NET code bases outside of Unity

However, most of the “wire up the game” work has to be done in C# code instead of within the Unity Editor

Principles at work

Split logic up into different classes according to type of responsibility

Move logic that doesn’t have to be in MonoBehaviours into regular C# classes

Reorganized version

```
public class Ball : MonoBehaviour {

    [Header("Customizable per instance")]
    [FormerlySerializedAs("Velocity")]
    public Vector3 InitialVelocity;

    private BallLogic ballLogic;
    private BallSimulation ballSimulation;

    void Start()
    {
        ballSimulation = new BallSimulation(InitialVelocity);
        ballLogic = new BallLogic(ballSimulation);
    }

    void FixedUpdate()
    {
        transform.localPosition = ballSimulation.UpdatePosition(transform.localPosition, Time.fixedDeltaTime);
    }

    void OnTriggerEnter(Collider collider)
    {
        bool isAlive = ballLogic.Hit(collider.gameObject.tag);
        if (!isAlive)
            Destroy(gameObject);
    }
}
```

Reorganized version

```
public class Ball : MonoBehaviour {

    [Header("Customizable per instance")]
    [FormerlySerializedAs("Velocity")]
    public Vector3 InitialVelocity;

    private BallLogic ballLogic;
    private BallSimulation ballSimulation;

    void Start()
    {
        ballSimulation = new BallSimulation(InitialVelocity);
        ballLogic = new BallLogic(ballSimulation);
    }

    void FixedUpdate()
    {
        transform.localPosition = ballSimulation.UpdatePosition(transform.localPosition, Time.fixedDeltaTime);
    }

    void OnTriggerEnter(Collider collider)
    {
        bool isAlive = ballLogic.Hit(collider.gameObject.tag);
        if (!isAlive)
            Destroy(gameObject);
    }
}
```

Initial velocity

Object lifetime change

Principles at work

Use interfaces to share minimal parts of the MonoBehaviour with other classes

Use delegates to support method calls “outward”

Reorganized version

```
public interface ILocalPositionAdapter
{
    Vector3 LocalPosition { get; set; }
}
```

```
public class Ball : MonoBehaviour, ILocalPositionAdapter {

    [Header("Customizable per instance")]
    [FormerlySerializedAs("Velocity")]
    public Vector3 InitialVelocity;

    public BallLogic BallLogic { get; private set; }
    private BallSimulation ballSimulation;

    public Vector3 LocalPosition
    {
        get { return transform.localPosition; }
        set { transform.localPosition = value; }
    }

    void Awake()
    {
        ballSimulation = new BallSimulation(InitialVelocity);
        BallLogic = new BallLogic(this, ballSimulation);
        BallLogic.OnDestroyed += () => Destroy(gameObject);
    }

    void FixedUpdate()
    {
        BallLogic.FixedUpdate(Time.fixedDeltaTime);
    }

    void OnTriggerEnter(Collider collider)
    {
        BallLogic.Hit(collider.gameObject.tag);
    }
}
```

Reorganized version

Interface for exposing info from Monobehaviour

```
public interface ILocalPositionAdapter
{
    Vector3 LocalPosition { get; set; }
}
```

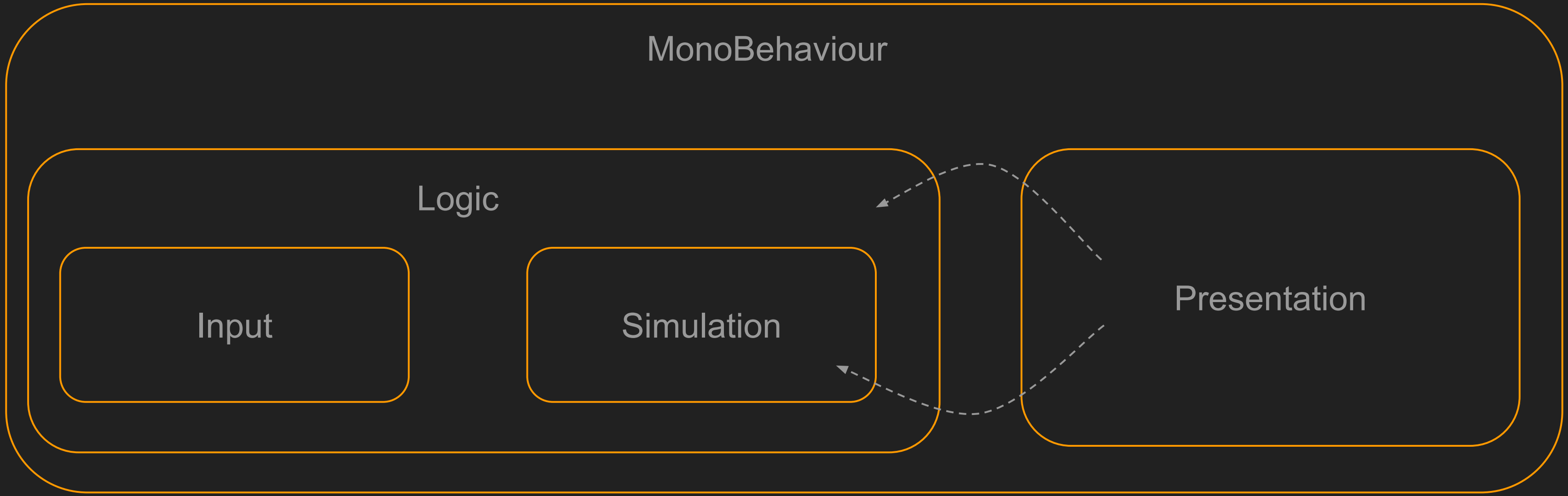
Delegate for “outward” Method calls

```
public class Ball : MonoBehaviour, ILocalPositionAdapter {

    [Header("Customizable per instance")]
    [FormerlySerializedAs("Velocity")]
    public Vector3 InitialVelocity;

    public BallLogic BallLogic { get; private set; }
    private BallSimulation ballSimulation;

    public Vector3 LocalPosition
    {
        get { return transform.localPosition; }
        set { transform.localPosition = value; }
    }

    void Awake()
    {
        ballSimulation = new BallSimulation(InitialVelocity);
        BallLogic = new BallLogic(this, ballSimulation);
        BallLogic.OnDestroyed += () => Destroy(gameObject);
    }

    void FixedUpdate()
    {
        BallLogic.FixedUpdate(Time.fixedDeltaTime);
    }

    void OnTriggerEnter(Collider collider)
    {
        BallLogic.Hit(collider.gameObject.tag);
    }
}
```

Software architecture

Know what’s in your toolbox

Can we create an almost-hierarchy of our classes?



Separate Logic and Presentation

Can you make it so that the Logic portion never reads from Presentation?

(Logic may sometimes need to wait for Presentation to catch up)

Create data-only classes

Sometimes a Thing is just a collection of data.

Create helper classes

Classes that work on data they don’t own

Create static methods

More functional programming -> less bugs!

Opinion piece by John Carmack:
https://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php

Decouple your objects

Buffers

Queues

Message buses

Messaging

1. Receiver(s) subscribe to events of type X with the message broker.
2. Sender(s) post events of type X to the message broker.

Excellent for Logic->Presentation decoupling

Source: http://www.willrmiller.com/?p=87

Organizing scene loading in a project

Stop using LoadSceneMode.Single

Use LoadSceneMode.Additive instead.

Use explicit unloads when you want to unload a scene - sooner or later you **will** need to keep a few objects alive during a scene transition.

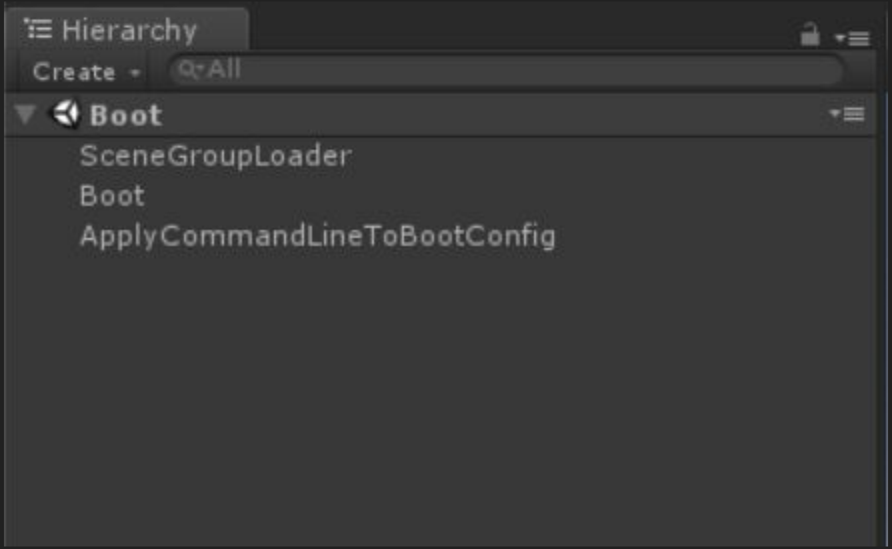
Stop using DontDestroyOnLoad

This makes you lose control over object lifetime.

You can mark something as DontDestroyOnLoad, but you cannot un-mark it!
(Well, you can manually delete it.)

Instead, put your long-lived objects into a special, long-lived scene.

Have a minimal boot scene



Boot
SceneGroupLoader
Boot
ApplyCommandLineToBootConfig

Support a clean, controlled shutdown

Make your application capable of releasing practically all resources before the application quits.

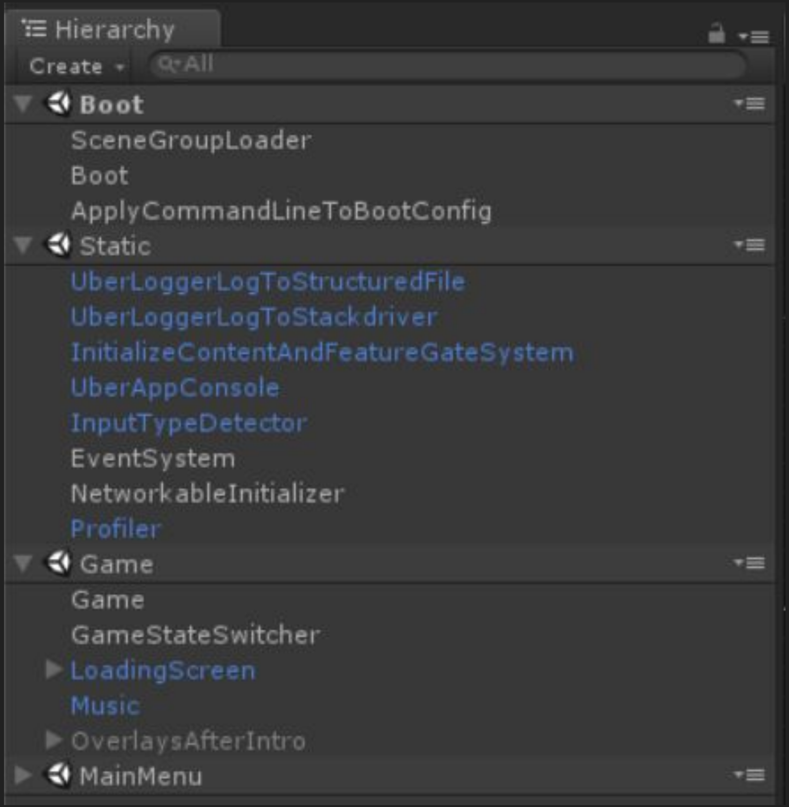
No global variables still assigned, if possible.

No DontDestroyOnLoad-marked GameObjects, if possible.

This is the perfect spot to look for resource leaks.

This will also leave your Unity Editor in a good state when you exit Play mode.

Our game’s scene layout


Hierarchy
Create
All

Boot
SceneGroupLoader
Boot
ApplyCommandLineToBootConfig
Static
UberLoggerLogToStructuredFile
UberLoggerLogToStackdriver
InitializeContentAndFeatureGateSystem
UberAppConsole
InputTypeDetector
EventSystem
NetworkableInitializer
Profiler
Game
Game
GameStateSwitcher
LoadingScreen
Music
OverlaysAfterIntro
MainMenu

Reduce scene file merge pains

- Use a real version control system (Git, Hg, Perforce, Plastic, ...)
- Store all assets as text
- Move objects out of scene files by making prefabs out of them
- $\boxed{\text{Split scene files into multiple smaller scenes}}$

May need extra tooling

Process automation

FRRRR
K.M.

Test your code

Build unit tests for your simulations and your logic, where it makes sense

Test your content

1. Build tests which your content creators can run to validate their prefabs
2. Make Unity’s “Test Runner” also run that validation logic

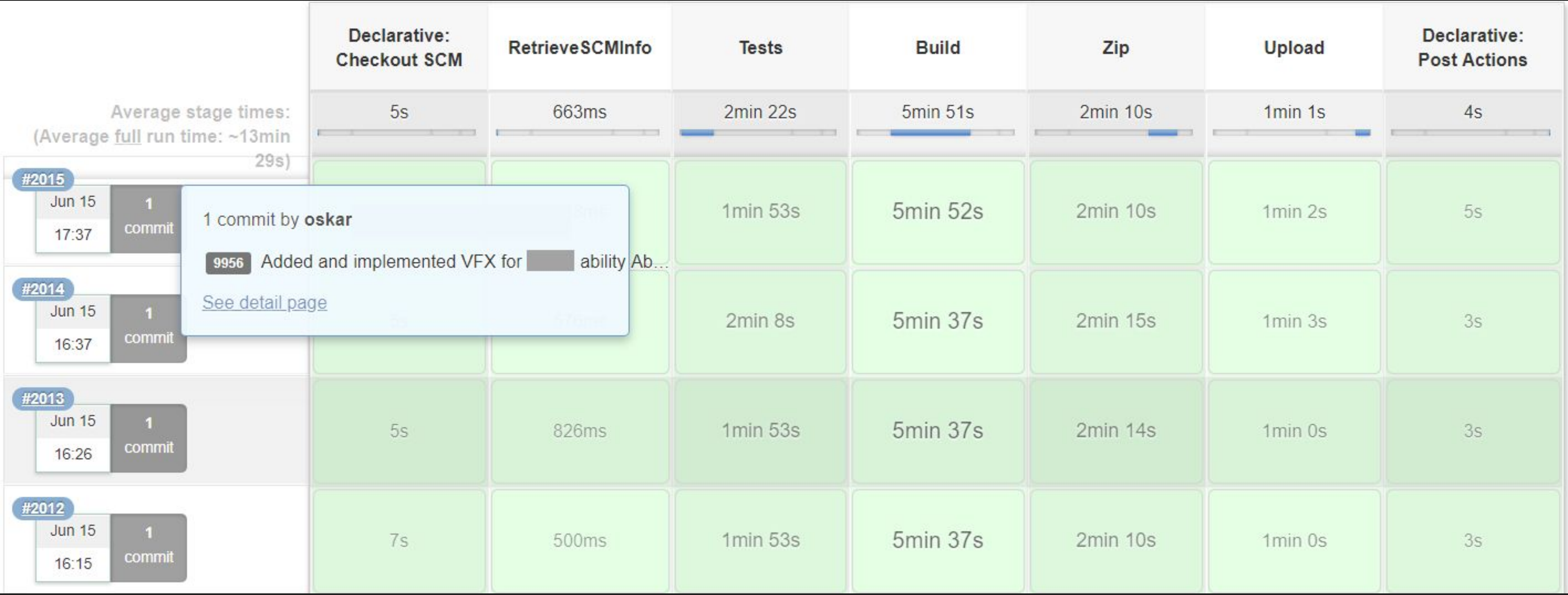
Problems found before starting the game = time saved

Create automated playthroughs

- Make an AI that can play your game
- Make it play the game every night
- Log any errors

Every error that your AI finds, is an error **you** don’t have to find

Automate building / testing



	Declarative: Checkout SCM	RetrieveSCMInfo	Tests	Build	Zip	Upload	Declarative: Post Actions
Average stage times: (Average full run time: ~13min 29s)	5s	663ms	2min 22s	5min 51s	2min 10s	1min 1s	4s
#2015 Jun 15 17:37 1 commit	[illegible]	[illegible]	1min 53s	5min 52s	2min 10s	1min 2s	5s
#2014 Jun 15 16:37 1 commit	[illegible]	[illegible]	2min 8s	5min 37s	2min 15s	1min 3s	3s
#2013 Jun 15 16:26 1 commit	5s	826ms	1min 53s	5min 37s	2min 14s	1min 0s	3s
#2012 Jun 15 16:15 1 commit	7s	500ms	1min 53s	5min 37s	2min 10s	1min 0s	3s

1 commit by **oskar**

9956 Added and implemented VFX for [redacted] ability Ab…

See detail page

Thank you.

Mikael Kalms

kalms@falldamagestudio.com

https://www.falldamagestudio.com

https://github.com/falldamagestudio/