Blind Guide Redesign Patterns

Students:
Victor Fortea Martinez, 500747258
Wasili Prattis, 500689732

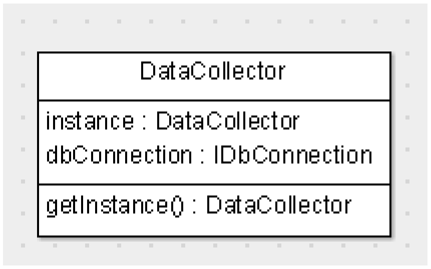
Chosen project:
Blind Guide - 2D slidescroller game made using the Unity3D engine and the C# programming language.
The game was created by Wasili Prattis and his Project Game Development (second year HvA project) team.
The player plays the role of a ghostly guide dog that wants to help his owner get to his home. There are multiple objects along the way, which can all be cleared using the player's special powers.
The blind guy can only take one hit during the entire game; taking a hit means game over.
All obstacles and enemies can die by one or multiple powers and some obstacles require a combination sequence of multiple attacks in order to be stopped

Contents

Singleton

We have in our project an access to database in order to save some data and load that data after. We only want to get one instance of this object. We only one instance of this class to be available, because the class writes data to local text files. We do not want to cause conflicts by having two instances writing to a file at the same time. To accomplish this, we use the singleton pattern, which allows the data collector to be globally available and helps us prevent conflicts when writing to local files.

Structure



Implementation

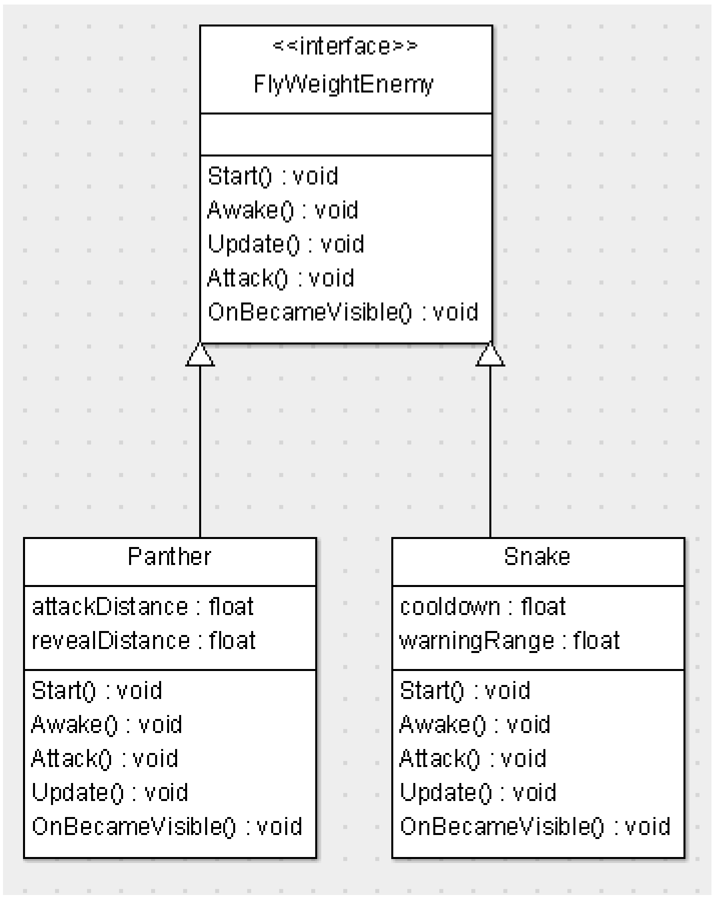
```
public class DataCollector {
    private static DataCollector instance;

    private IDbConnection dbConnection;

    private DataCollector()
    {
        dbConnection = (IDbConnection)new SqliteConnection(_constr);
        dbConnection.Open();
    }

    public static DataCollector GetInstance()
    {
        //make sure there's always an instance of the datacollector
        if (instance == null) instance = new DataCollector();
        return instance;
    }
}
```

GetInstance() can be called from any class to gain access to the public members of the data collector

Flyweight

If your project is big with a lot of instances, the flyweight pattern is very nice but if not is also a good practise to do it in order to not use memory that could be used differently. To do that I created an interface with 5 methods that are override depending the object called. The objects have also their own methods to get into the context.

Structure



Implementation

```
public abstract class FlyWeightEnemy : MonoBehaviour
{
    public abstract void Start();
    public abstract void Awake();
    public abstract void Update();
    public abstract void Attack();
    public abstract void OnBecameVisible();

}
```
All shared methods are declared in a base class

```
public class Snake : FlyWeightEnemy
{
    public SpriteRenderer spriteRenderer;
    //public PolygonCollider2D start, threat, attack;
    public Sprite initial, threatening, attacking, frozen, burnt;
    public float cooldown, maxCooldown, warningRange, attackRange;
    private bool dead = false, firstWarning = true;
    public AudioClip hiss;
    Transform blindGuyTransform;
    DataMetricObstacle dataMetric = new DataMetricObstacle();

    public override void Start()
    {
        spriteRenderer = gameObject.GetComponent<SpriteRenderer>();
        spriteRenderer.sprite = initial;
        dataMetric.obstacle = DataMetricObstacle.Obstacle.Snake;
    }

    public override void Awake()
    {
        blindGuyTransform = GameObject.FindWithTag("Blindguy").transform;
    }


    //name changed
    public override void Attack()
    {
        spriteRenderer.sprite = attacking;
        cooldown = maxCooldown;
        Invoke("Warning", maxCooldown);
    }

    public override void Update()
    {
        if (blindGuyTransform == null)
            blindGuyTransform = GameObject.FindWithTag("Blindguy").transform;

        if (!dead)
        {
            if (cooldown >=0)
            cooldown -= Time.deltaTime;

            if (Vector2.Distance(transform.position, blindGuyTransform.position) <= 
warningRange && Vector2.Distance(transform.position, 
GameObject.FindWithTag("Blindguy").transform.position) > attackRange)
            {
                Warning();
            }

            if (cooldown <= 0)
            {
                if (Vector2.Distance(transform.position, blindGuyTransform.position) <= 
attackRange)
                {
                    Attack();
                }
            }
        }
    }

    public override void OnBecameVisible()
    {
        dataMetric.spawnTime = Time.timeSinceLevelLoad.ToString();
    }
}
```

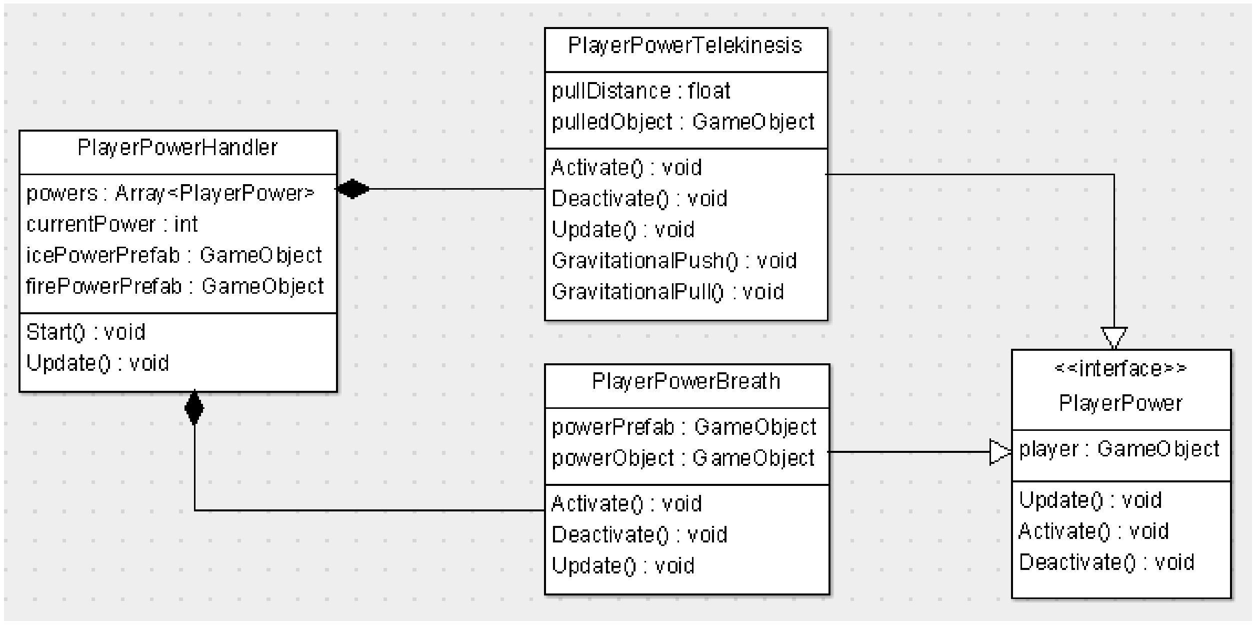
Snake class uses these methods to define its behaviour

```
public class Panther : FlyWeightEnemy
{
    public float attackDistance, revealDistance, speed, throwSpeed, frameTime,
       animationTime, fade, visibility, speedCap, offset;
    float frameTimer, rotate = -20;
    public Sprite[] running, sneaking, leaping;
    Sprite[] triggeredAnimation;
    public Sprite burnt, frozen;
    SpriteRenderer spriteRenderer;
    public int curSprite = 0, speedmod = 5;
    public bool goLeft;
    private bool dead = false;
    public bool isEndBoss = false;

    DataMetricObstacle dataMetric = new DataMetricObstacle();

    public override void Start()
    {
        frameTimer = animationTime;
        fade = 0;
        spriteRenderer = gameObject.GetComponent<SpriteRenderer>();
        triggeredAnimation = sneaking;

        dataMetric.obstacle = DataMetricObstacle.Obstacle.Panther;

        if (isEndBoss)
        {
            GetComponent<Collider2D>().enabled = false;
            dataMetric.obstacle = DataMetricObstacle.Obstacle.EndBoss;
        }
    }

    public override void Update()
    {
        spriteRenderer.color = new Color(255, 255, 255, fade);
        if (!dead)
        {
            AnimatePanther();
            Movement();

            if (Vector2.Distance(transform.position, 
GameObject.FindWithTag("Blindguy").transform.position) <= attackDistance)
            {
                Attack();
            }
            else if (Vector2.Distance(transform.position, 
GameObject.FindWithTag("Blindguy").transform.position) <= revealDistance)
            {
                Reveal();
            }
            if (this.transform.position.x <= 
(GameObject.FindGameObjectWithTag("Blindguy").transform.position.x + offset))
            {
                speed = 0;
                speedCap = 0;
                dead = true;
            }
        }
        else
        {
            if (isEndBoss)
            {
                transform.localScale -= new Vector3(-Time.deltaTime, Time.deltaTime, 
0);
                if (transform.localScale.y <= 0)
                    Destroy(gameObject);
            }
            fade = 1;
        }
    }


    public override void Awake()
    {

    }

    public override void Attack()
    {
        if (!dead)
        {
            transform.rotation = Quaternion.Euler(0, 0, rotate);
            triggeredAnimation = leaping;
        }
    }


    public override void OnBecameVisible()
    {
        dataMetric.spawnTime = Time.timeSinceLevelLoad.ToString();
    }
}
```
Another class that uses these methods is the panther, along with a lot of other enemies that define their own behaviour within these functions

State

In Blind Guide, the player can choose between three different abilities to use in order to destroy obstacles and enemies. Seeing as only one of these powers can be active at a time, it is required that the player keeps track of the current power in order to decide which power to activate when the player presses the 'Use Power' button.
We accomplish this behaviour by using the state pattern: we create an array of available powers and switch between these powers whenever the player presses the 'Switch Power' button. By having an integer that tells us which power we're currently using, we can activate the correct power whenever the player attacks.
The first power the player can use is telekinesis. The player can pull certain objects, that are marked appropriately with a tag, towards him and then throw them away to prevent the Blind Guy from being harmed.
The second and third power the player can use are quite similar; the player can breathe fire or ice in order to burn or freeze objects. Seeing as these powers are quite similar, we only need to define one state for both and simply give the spawned object (flame or ice particles) a tag that will define its type. Objects decide what happens to them when collision with the attacks is registered.

Structure



Implementation

```
public class PlayerPowerHandler : MonoBehaviour {
        PlayerPower[] powers;
        int currentPower = 0;
        public GameObject icePowerPrefab, firePowerPrefab;

        void Start() {
                //set the available powers
                 powers = new PlayerPower[] {
                        new PlayerPowerTelekinesis(this.gameObject),
                        new PlayerPowerBreath(this.gameObject, firePowerPrefab),
                        new PlayerPowerBreath(this.gameObject, icePowerPrefab) };
        }

        // Update is called once per frame
        void Update() {
                if (Input.GetButtonDown("Switch Power")) {
                        //deactivate current power
                        powers[currentPower].Deactivate();
                        //go to next power if available, otherwise pick the first
                        currentPower = currentPower < powers.Length-1 ?currentPower+1 :0;
                        //actiate the new power
                        powers[currentPower].Activate();
                }

                powers[currentPower].Update();
        }
}
```
The PlayerPowerHandler class keeps track of the current state and calls its update method every frame

```
public abstract class PlayerPower {
        //keep track of player object
        protected GameObject player;
        public PlayerPower(GameObject player) {
                this.player = player;
        }
        //frame-by-frame logic
        public abstract void Update();
        //call when power is selected
        public abstract void Activate();
        //call when power is deselected
        public abstract void Deactivate();
}
```
The base class for each power contains functions that must be implemented

```
public class PlayerPowerTelekinesis : PlayerPower {
	private float pullDistance = 6.0f;
	GameObject pulledObject;

	public PlayerPowerTelekinesis(GameObject player) : base(player) { }

	public override void Activate() {
		//send a message to the player to trigger sprite switches and sounds
		player.SendMessage("SwitchPower", "Telekinesis");
	}

	public override void Deactivate() {
		//make sure the pulled object doesn't return when we switch back
		pulledObject = null;
	}

	public override void Update() {
		if (Input.GetButton("Use Power")) {
			GravitationalPull();
		}
		if (Input.GetButtonUp("Use Power")) {
			GravitationalPush();
		}
	}

	//called when power is active
	private void GravitationalPull() {
		//find all pullable objects in the scene
		GameObject[] pullableObjects =
			GameObject.FindGameObjectsWithTag("PullableObject");

		//iterate through each object
		for (int i = 0; i < pullableObjects.Length; i++) {
			//check their distance from the player
			if (Vector3.Distance(pullableObjects[i].transform.position,
				player.transform.position) < pullDistance) {
				pulledObject = pullableObjects[i];
				//pull object towards the player
				pullableObjects[i].GetComponent<Rigidbody2D>().velocity =
					(player.transform.position -
					pullableObjects[i].transform.position) * 10;
			}
		}
	}

	//called when power is deactivated
	private void GravitationalPush() {
		if (pulledObject != null) {
			//throw object away, towards the mouse cursor
			pulledObject.GetComponent<Rigidbody2D>().velocity =
			GlobalGuide.AimTowardsMouse(pulledObject.transform.position)
				* 10;
		}
	}
}
```
Telekinesis pulls objects towards the player and throws them away towards the cursor

```
public class PlayerPowerBreath : PlayerPower {
	private GameObject powerObject, powerPrefab;

	public PlayerPowerBreath(GameObject player, GameObject icePowerPrefab)
		: base(player) {
		this.powerPrefab = icePowerPrefab;
	}
	public override void Activate() {
		//trigger spreite switches, animations and sounds
		player.SendMessage("SwitchPower", "Ice");
	}

	public override void Deactivate() {
		//destroy the spawned power when we switch
		GameObject.Destroy(powerObject);
	}

	public override void Update() {
		if (Input.GetButtonDown("Use Power")) {
			//spawn the power object
			powerObject = (GameObject)GameObject.Instantiate(powerPrefab,
				player.transform.position,
				player.transform.rotation);
			//make sure it follows the player position and rotation
			powerObject.transform.parent = player.transform;
			//place it slightly in front of the player
			powerObject.transform.localPosition = new Vector2(-2.0f, 0.0f);
		}
	}
}
```
Fire and ice attacks use the same behaviour and spawn a prefab containing animations and collision detection

```
public void OnTriggerEnter2D(Collider2D target) {
	if (target.gameObject.tag == "IceAttack") {
		IceDeath();
	}
}
```
Objects that can be killed by fire or ice check for a tag when they detect collision (built-in unity call)

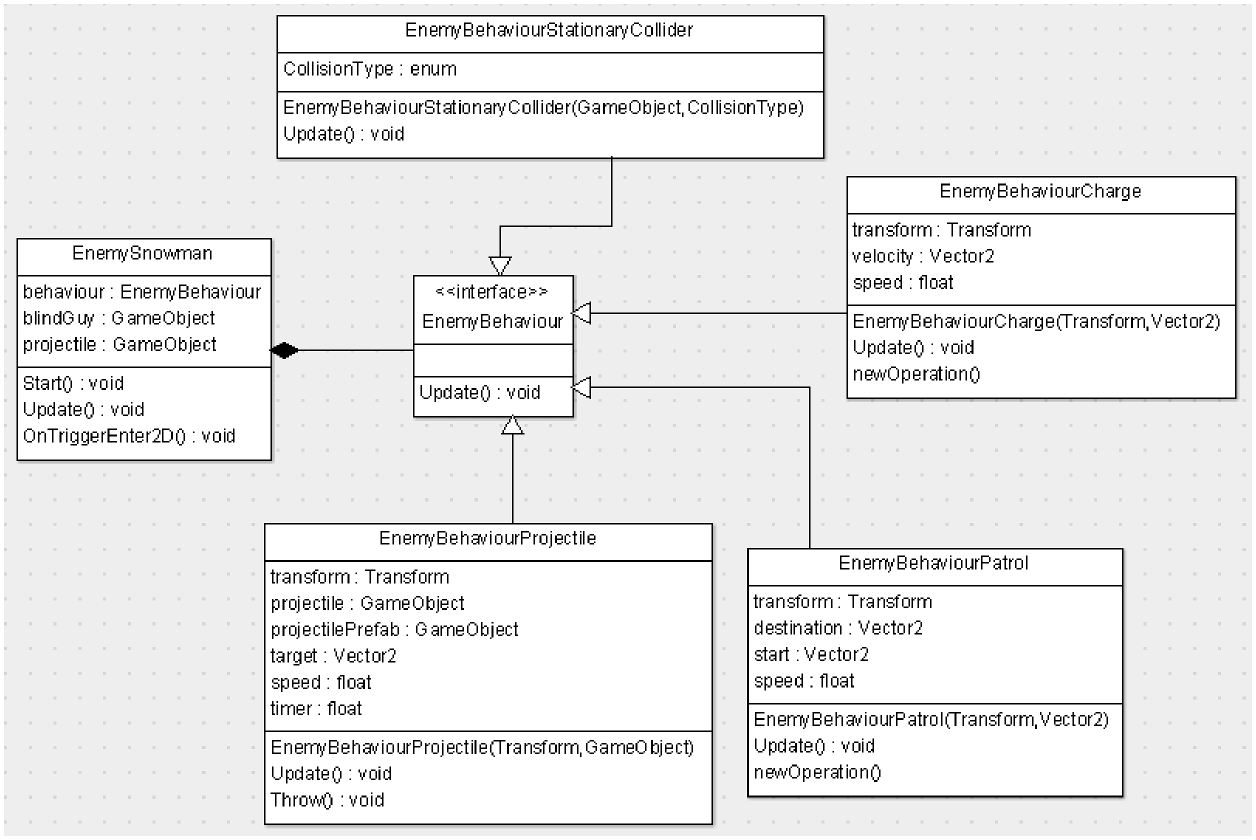
Strategy

Enemies in the game can have different behaviour depending on the situation. This means that we should be able to apply behaviour to enemies during runtime and that we should have a system that allows us to separate general behaviour from specific behaviour. Seeing as all enemies have either one of the following traits, we can easily accomplish this using the strategy pattern:

- Stationary collider object that kills the blind guy on collision
- Patrolling enemy that moves between to specified positions
- Projectile-throwing object that throws projectiles towards the blind guy
- Charing object that moves towards the blind guy at a certain speed

These behaviours can be contained within their own class, all inheriting from a base EnemyBehaviour class. Specific enemies then use these different behaviours to handle their behaviour without having to duplicate code each time two objects have the same behaviour.

Structure



Implementation

```
public class EnemySnowman : MonoBehaviour {
       private EnemyBehaviour behaviour;
       private GameObject blindGuy;
       public GameObject projectile;
       public Sprite moltenSprite;
       private bool canSwitchAttack = false;
```

```
        // Use this for initialization
        void Start () {
               blindGuy = GameObject.FindWithTag("Blindguy");
               behaviour = new EnemyBehaviourPatrol(transform,
                      transform.position + transform.TransformDirection(Vector2.right)
                      * 8.0f);
        }

        // Update is called once per frame
        void Update () {
               behaviour.Update();
               //throw projectiles at the blind guy when he's near
               if (Vector2.Distance(blindGuy.transform.position,
                      transform.position) < 10
                      && canSwitchAttack) {
                      behaviour = new EnemyBehaviourProjectile(transform, projectile);
                      canSwitchAttack = false;
               }
        }

        void OnTriggerEnter2D(Collider2D coll) {
               //switch to a collision object when we melt
               if (coll.gameObject.tag == "FireAttack") {
                      //we can no longer attack
                      canSwitchAttack = false;
                      GetComponent<SpriteRenderer>().sprite = moltenSprite;
                      behaviour = new EnemyBehaviourStationaryCollider(
                             this.gameObject,
                             EnemyBehaviourStationaryCollider.CollisionType.Ice);
                      //make this a pullable object for telekinesis
                      this.gameObject.tag = "PullableObject";
                      GetComponent<Rigidbody2D>().isKinematic = false;
               }
        }
}
```
The snowman only needs to know when to switch between different behaviours; the behaviour logic is handled in its own class

```
public abstract class EnemyBehaviour {
        public abstract void Update();
}
```
EnemyBehaviour base class only needs an Update method

```
public class EnemyBehaviourCharge : EnemyBehaviour {
	private Transform transform;
	private Vector2 velocity;
	private float speed = 10.0f;

	public EnemyBehaviourCharge(Transform callerTransform,
		Vector2 chargeDestination) {
		transform = callerTransform;
		//calculate velocity once based on destination
		velocity = (chargeDestination - (Vector2)transform.position).normalized;
	}

	public override void Update() {
		//move with charge velocity
		transform.position += (Vector3)velocity
			* Time.deltaTime
			* speed;
	}
}
```
Charge behaviour can be used to make an object roll towards the blind guy

```
public class EnemyBehaviourProjectile : EnemyBehaviour {
	private Transform transform;
	private GameObject projectile, projectilePrefab;
	private Vector2 target;
	private float speed = 15.0f;
	private float timer;

	public EnemyBehaviourProjectile(Transform callerTransform,
		GameObject projectileToThrow) {
		transform = callerTransform;
		projectilePrefab = projectileToThrow;
		Throw();
	}

	public override void Update() {
		//throw a projectile every couple of seconds
		timer -= Time.deltaTime;
		if (timer <= 0) {
			Throw();
		}
	}

	private void Throw() {
		timer = 5.0f;
		projectile = (GameObject)GameObject.Instantiate(projectilePrefab,
			transform.position,
			transform.rotation);
		target = GameObject.FindWithTag("Blindguy").transform.position;
	}
}
```
Some enemies can throw projectiles at the player. Projectile logic is handled in separate projectile scripts, because some behave differently

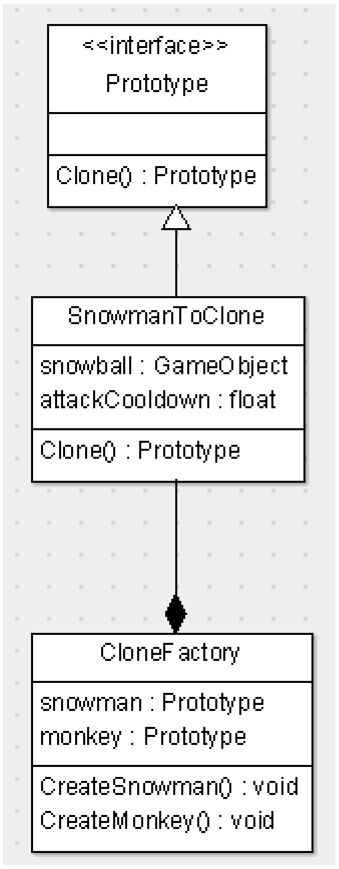
```
public class EnemyBehaviourStationaryCollider : EnemyBehaviour {
	public enum CollisionType { Fire, Ice, Normal }
	public EnemyBehaviourStationaryCollider(GameObject caller, CollisionType type) {
		//apply one of three different types of deaths
		switch (type) {
			case CollisionType.Fire:
				caller.AddComponent<SetFlameDeath>();
				break;
			case CollisionType.Ice:
				caller.AddComponent<SetFrozenDeath>();
				break;
			case CollisionType.Normal:
				caller.AddComponent<SetDizzyDeath>();
				break;
		}
	}

	public override void Update() {
		//collision is handled by the Set<Type>Death components
	}
}
```
A stationary collider needs to know what kind of death it should apply on collision

```
public class EnemyBehaviourPatrol : EnemyBehaviour {
	private Transform transform;
	private Vector2 destination, start;
	private float speed = 3.0f;

	public EnemyBehaviourPatrol(Transform callerTransform,
		Vector2 patrolDestination) {
		transform = callerTransform;
		destination = patrolDestination;
		start = transform.position;
	}

	public override void Update () {
		//move towards patrol destination
		if (Vector2.Distance(transform.position, destination) > 1.0f) {
			transform.position += ((Vector3)destination
				- transform.position).normalized
				* Time.deltaTime
				* speed;
		}
		//turn around if we reach the destination
		else {
			Vector3 current = destination;
			destination = start;
			start = current;
		}
	}
}
```
When using the patrol behaviour, the player does not need to be attacked, the enemy simply movies between two points

Prototype

The aim of this pattern is to solve memory when you are executing your code cloning your objects instead of creating new one so you just need to copy the object. We apply that to create enemies for our game.

Structure



Implementation

```
public class SnowmanToClone : Prototype
{
    public GameObject snowball;
    public GameObject rollingStoneIce;
    public float attackCooldown, attackDistance, retreatDistance, speed, throwSpeed,
            frameTime;
    float timer, animationTimer, attackTimer;
    public Sprite[] sprites;
    SpriteRenderer spriteRenderer;
    int currentSprite = 0;
    public bool goLeft, attack2;
    private bool dead = false;
    public AudioClip throwball;
    DataMetricObstacle dataMetric = new DataMetricObstacle();

    //here is where you get your object cloned
    public override Prototype Clone()
    {
        return (Prototype)this.MemberwiseClone();
    }
}
```
Snowman returns a copy of itself when someone wishes to clone it

```
public abstract class Prototype : MonoBehaviour {

    // the interface to acces to SnowmanToClone, method overrided in SnowmanToClone
    public abstract Prototype Clone();
}
```
The prototype base class provides a clone function that must be implemented

```
public class CloneFactory : MonoBehaviour {
    public Prototype snowman, monkey;
    //We create a snowman and we clone it if it already exists and same with monkey
    void CreateSnowman() {
        if (snowman == null) {
            snowman = Instantiate(snowman.gameObject).GetComponent<Prototype>();
        }
        else
        {
            Instantiate(snowman.Clone().gameObject);
        }
    }

    void CreateMonkey()
    {
        if (monkey == null)
        {
            monkey = Instantiate(monkey.gameObject).GetComponent<Prototype>();
        }
        else
        {
            Instantiate(monkey.Clone().gameObject);
        }
    }
}
```
The clone factory clones objects on request of, for example, the procedural content generator

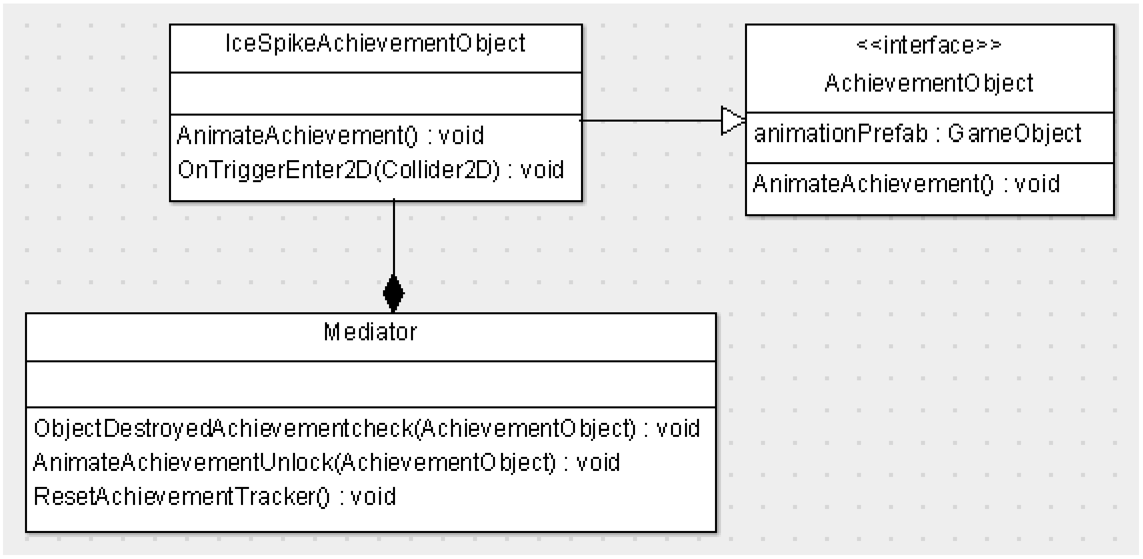
Mediator

Players in Blind Guide can unlock achievements by doing certain tasks. One such task is killing 10 obstacles in a row, without letting the player die. Doing so unlocks a certain animation for the object that was destroyed last. In order to accomplish this, we do not want the achievement tracker and the objects to be linked to each other. That's why we implement a mediator pattern to serve as a middle-man.

The mediator will receive calls from obstacles when they are destroyed. The mediator will then tell the achievement tracker that an object has been destroyed and the tracker should be updated. If enough obstacles were destroyed in a row, the achievement tracker will tell the mediator that an achievement has been unlocked. The mediator can then tell the object that it should play an animation for unlocking the achievement.

The mediator can be expanded to also play a sound, send messages to the server if online achievement tracking is ever implemented, and anything else you can come up with. All this can be done without having to touch the achievement tracker or the obstacle code.

Structure


IceSpikeAchievementObject
AnimateAchievement() : void
OnTriggerEnter2D(Collider2D) : void
<<interface>>
AchievementObject
animationPrefab : GameObject
AnimateAchievement() : void
Mediator
ObjectDestroyedAchievementcheck(AchievementObject) : void
AnimateAchievementUnlock(AchievementObject) : void
ResetAchievementTracker() : void

Implementation

```
public abstract class AchievementObject : MonoBehaviour {
	//the prefab containing the animation to be played
	public GameObject animationPrefab;
	//trigger the animation sequence for an unlocked achievement
	public abstract void AnimateAchievement();
}
```

Other obstacles can have a component that inherits from this class to handle the unlocking of achievements

```
public class IceSpikeAchievementObject : AchievementObject {
		public override void AnimateAchievement() {
		//instantiate the object containing the animation to be played
		Instantiate(animationPrefab, transform.position, transform.rotation);
	}

	void OnTriggerEnter2D(Collider2D coll) {
		//this object can be destroyed by fire
```

```
              if (coll.gameObject.tag == "FireAttack") {
                     //see if we should play an animation upon destruction
                     FindObjectOfType<Mediator>()
                            .ObjectDestroyedAchievementcheck(this);
              }
       }
}
```

When the ice spike triggers an achievement, it creates an object that plays the animation without any other behaviour in the game

```
public class Mediator : MonoBehaviour {
       public void ObjectDestroyedAchievementcheck(AchievementObject obj) {
              //update the achievement tracker

       FindObjectOfType<AchievementTracker>().UpdateObjectDestructionTracker(obj);
       }

       public void AnimateAchievementUnlock(AchievementObject obj) {
              //play an animation because an achievement was unlocked
              obj.AnimateAchievement();
       }

       public void ResetAchievementObjectTracker() {
              FindObjectOfType<AchievementTracker>().ResetObjectDestructionTracker();
       }
}
```

The mediator connects the achievement tracker and the achievement objects and calls their functions when needed

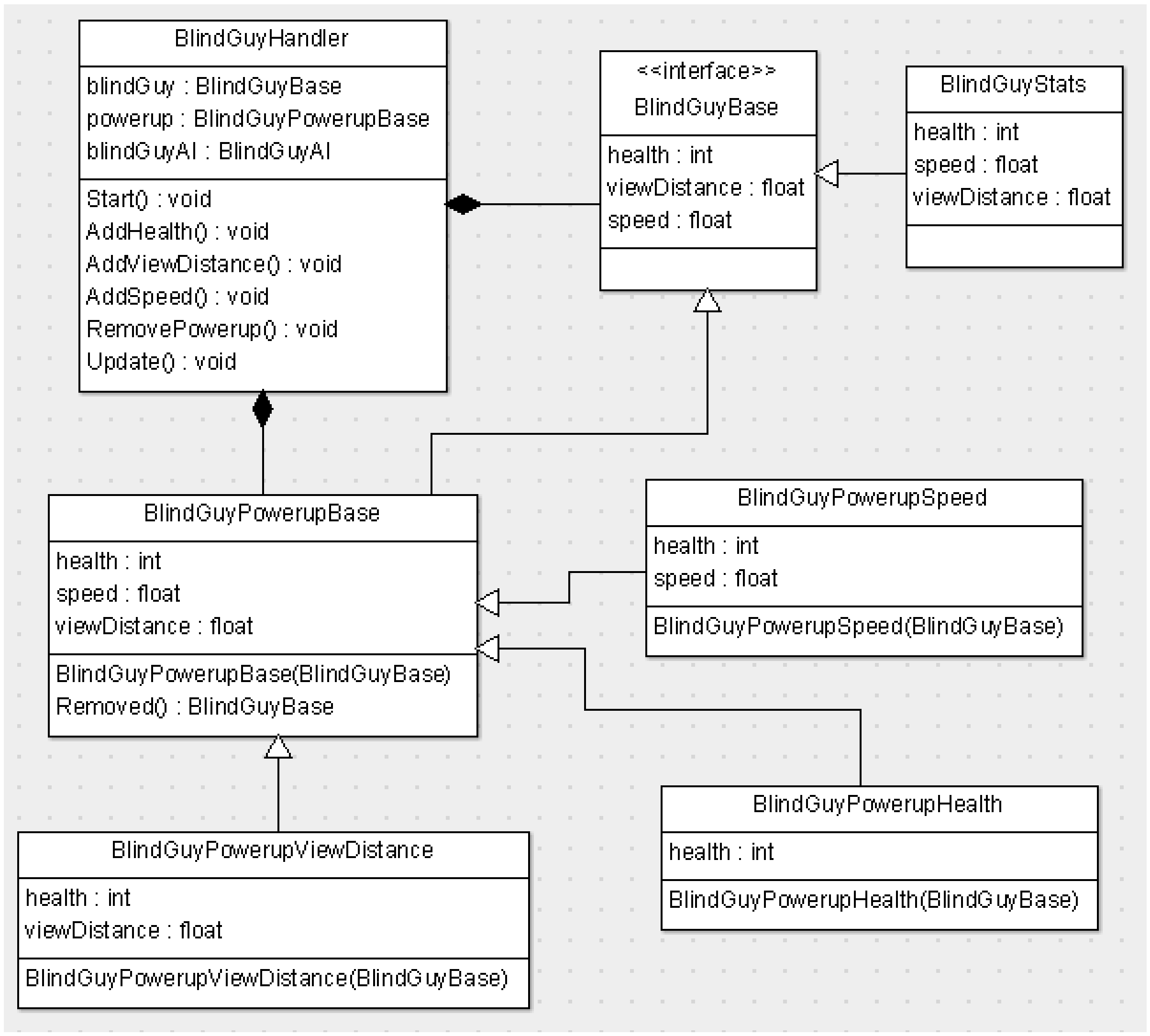
```
public class AchievementTracker : MonoBehaviour {
       //keep track of how many obstacles have been destroyed so far
       static int objDestructionCount = 0;
       bool objDestructionCountUnlocked = false;

       //update the amount of obstacles detroyed so far
       public void UpdateObjectDestructionTracker(AchievementObject obj) {
              objDestructionCount++;
              if (objDestructionCount > 10 && !objDestructionCountUnlocked) {
                     //play an animation sequence on the object we destroyed
                     FindObjectOfType<Mediator>().AnimateAchievementUnlock(obj);
                     objDestructionCountUnlocked = true;
              }
       }

       //reset the destruction tracker
       public void ResetObjectDestructionTracker() {
              objDestructionCount = 0;
       }
}
```

The achievement tracker decides when an achievement unlock trigger should be called, based on the data is has collected so far

Decorator

The player can pick up certain objects that act as a power-up to the blind guy. These objects can be found by killing objects or are hidden behind screen obstructions in the levels. Picking up one of these power-ups increases the blind guy's health and sometimes also provides an extra boost to, for example, the blind guy's speed or the view distance of the camera. Seeing as these power-ups should stack endlessly (limits will be in the form of object placement, not code) we need a way to add and remove power-ups during runtime.
We can accomplish this using the decorator pattern: we add a power-up whenever the player picks up a power-up object and we remove the last power-up when the blind guy takes damage.
By giving the blind guy a component that handles the power-up states, we can let scripts communicate to trigger collision events with power-up pickups and hostile obstacles.

Structure



Implementation

```
public abstract class BlindGuyBase {
	public abstract int health { get; }
```

```
        public abstract float viewDistance { get; }
        public abstract float speed { get; }
}
```
Base class for the blind guy stats and the blind guy power-ups

```
public class BlindGuyStats : BlindGuyBase {
        public override int health {
                get { return 0; }
        }

        public override float speed {
                get { return 2.0f; }
        }

        public override float viewDistance {
                get { return 5.5f; }
        }
}
```
BindGuyStats provides base values for the blind guy, which the power-ups can expand upon

```
public class BlindGuyPowerupBase : BlindGuyBase {
        private BlindGuyBase blindGuy;
        public BlindGuyPowerupBase(BlindGuyBase blindGuy) {
                this.blindGuy = blindGuy;
        }

        //return the blindguy without the latest power-up attached
        public BlindGuyBase Removed() {
                return blindGuy;
        }

        public override int health {
                get { return blindGuy.health; }
        }

        public override float speed {
                get { return blindGuy.speed; }
        }

        public override float viewDistance {
                get { return blindGuy.viewDistance; }
        }
}
```
The base class of the power-ups provides the base values of the player

```
public class BlindGuyPowerupHealth : BlindGuyPowerupBase {
        public BlindGuyPowerupHealth(BlindGuyBase blindGuy) : base(blindGuy) { }
        public override int health {
                get { return base.health + 1; }
        }
}
```
Increase the health of the blind guy

```
public class BlindGuyPowerupSpeed : BlindGuyPowerupBase {
        public BlindGuyPowerupSpeed(BlindGuyBase blindGuy) : base(blindGuy) { }
        public override float speed {
                get { return base.speed + 0.5f; }
        }
```

```
	public override int health {
		get { return base.health + 1; }
	}
}
```
Increase the speed of the blind guy

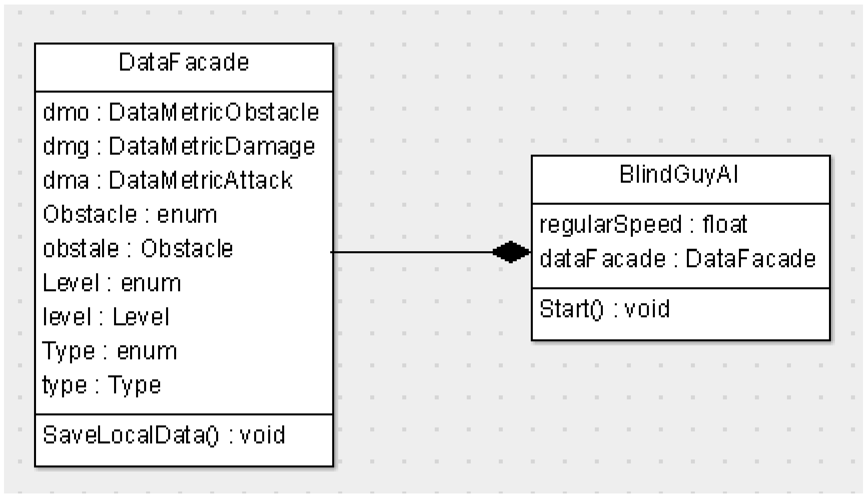
```
public class BlindGuyPowerupViewDistance : BlindGuyPowerupBase {
	public BlindGuyPowerupViewDistance(BlindGuyBase blindGuy) : base(blindGuy) { }
	public override float viewDistance {
		get { return base.viewDistance + 0.5f; }
	}

	public override int health {
		get { return base.health + 1; }
	}
}
```
Increase the view distance of the blind guy so the player has more time to react to danger

```
public class BlindGuyHandler : MonoBehaviour {
	BlindGuyBase blindGuy;
	BlindGuyPowerupBase powerup;
	BlindGuyAI blindGuyAI;

	void Start () {
		//create a new blind guy stats object
		blindGuy = new BlindGuyStats();
		//give the powerup a base to work with
		powerup = new BlindGuyPowerupBase(blindGuy);
		blindGuyAI = GetComponent<BlindGuyAI>();
	}

	public void AddHealth() {
		/* apply a new power-up and give the current
		 * power-up as base to expand upon */
		powerup = new BlindGuyPowerupHealth(powerup);
	}

	public void AddViewDistance() {
		powerup = new BlindGuyPowerupViewDistance(powerup);
	}

	public void AddSpeed() {
		powerup = new BlindGuyPowerupSpeed(powerup);
	}

	public void RemovePowerup() {
		powerup = (BlindGuyPowerupBase)powerup.Removed();
	}

	void Update () {
		//update blind guy values
		blindGuyAI.health = powerup.health;
		blindGuyAI.viewDistance = powerup.viewDistance;
		blindGuyAI.speed = powerup.speed;
	}
}
```
Added to the blind guy as a component, this class handles all communication between power-ups and the components that use them

Façade

The aim here is to make a class that allow some classes just access to an intermediate and that intermediate access to the other class that the first one wanted to access. We have three different types of data to be saved, we are going to create a class that allow the other classes just to call that one to save data.

Structure



Implementation

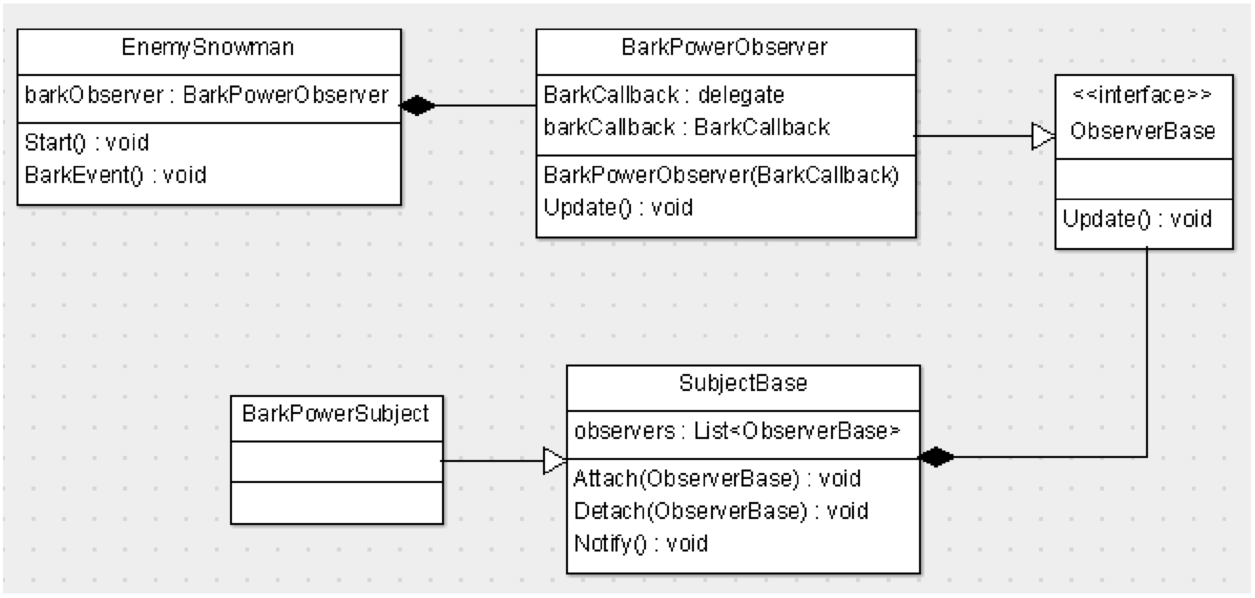
```
public class DataFacade {
        DataMetricObstacle dmo = new DataMetricObstacle();
        DataMetricGame dmg = new DataMetricGame();
        DataMetricAttack dma = new DataMetricAttack();
        //class obstacle
        public enum Obstacle {
            Monkey, FlyingEnemy, Lavaman,
            Panther, Snake, Snowman, FallingRock, FireFinish, Geyser, EndBoss,
            IcePool, Icicle, Lavafall, RollingBoulder, RollingBoulderSurprise, Coconut,
LavaBall, SnowBall
        }
        public Obstacle obstacle;
        public string spawnTime;
        public string defeatedTime;
        public string howItDied;
        //for obstacle and attack
        public int gameID;
        //class Game
        public enum Level { Tutorial, Fire1, Fire2, Fire3, Ice1, Ice2, Ice3, Jungle1,
Jungle2, Jungle3 }
        public int session;
        public string starttime;
        public string endTime;
        public Level level;
        public int playerDied;
        public string howPlayerDied;
        //class Attack
        public enum Type { Fire, Ice, Telekinesis, Destruction }
        public string attackTime;
        public Type type;

        //depending the type of data got you access to one or other class
        public void SaveLocalData() {
            if(spawnTime != null) {
                dmo.saveLocalData();
            }

            else if( session != 0) {
                dmg.saveLocalData();
            }

            else {
                dma.saveLocalData();
            }
        }
    }
```
All data saving is handled in the façade class

```
public class BlindGuyAI : MonoBehaviour {
    public float stopTimer = 0;
    public float speed = 2;

    float regularSpeed;

    Vector3 positionOffset, blindGuySize;

    public Sprite[] burned;
    public Sprite[] dizzy;
    public Sprite[] frozen;
    public Sprite[] stunned;
    public Sprite[] walking;
    int curSprite;
    public float animationTime = 1;
    float frameTimer;
    public AudioClip freezeDeath, flameDeath, dazedDeath;
    DataFacade dataFacade = new DataFacade();

    Sprite[] triggeredAnimation;

    bool dying = false;
    public int health;
    public float viewDistance;

    void Start () {
        frameTimer = animationTime;
        regularSpeed = speed;
        dataFacade.starttime = System.DateTime.Now.ToString();
        dataFacade.level = (DataFacade.Level) Application.loadedLevel;
    }
}
```
Blind guy is one of the examples of saving data using the façade

Observer

The player has the option to make the blind guy pause momentarily. The player character does this by letting out a loud bark, which makes the blind guy look around for a bit while he's paused. Seeing as this is a loud bark, certain enemies should be able to hear this bark as well, and react to it. So, we want a way for certain objects to be notified whenever the player uses his bark function.
We can accomplish such behaviour by using the observer pattern: we have a subject (the bark power event caller) and observers that are notified when the bark event is triggered. This allows multiple objects to be notified at the same time, without having to use the expensive built-in SendMessage method provided by Unity3D. We keep track of objects that want to be notified by storing them in a list inside the subject. Objects that want to be notified can then pass a callback method onto the observer, which will then be executed when the bark power event is triggered.

Structure



Implementation

```
public abstract class ObserverBase {
        public abstract void Update();
}
```

The observer base class provides an Update method.

```
public class BarkPowerObserver : ObserverBase {
        //create a delegate for callbacks
        public delegate void BarkCallback();
        //callback function is stored in this delegate
        private BarkCallback barkCallback;

        public BarkPowerObserver(BarkCallback callbackFunction) {
                //save the callback method
                barkCallback = callbackFunction;
        }

        public override void Update() {
                //execute the callback method
                barkCallback();
        }
}
```
The bark power observer requires a callback method to be assigned when it is created. The callback is then executed when the player barks.

```
public abstract class SubjectBase : MonoBehaviour {
        private List<ObserverBase> observers = new List<ObserverBase>();

        public void Attach(ObserverBase ob) {
                observers.Add(ob);
        }

        public void Detach(ObserverBase ob) {
                observers.Remove(ob);
        }

        public void Notify() {
                foreach (ObserverBase ob in observers) {
                        ob.Update();
                }
        }
}
```
The subject base class has a list of observers and methods to add, remove and notify observers.

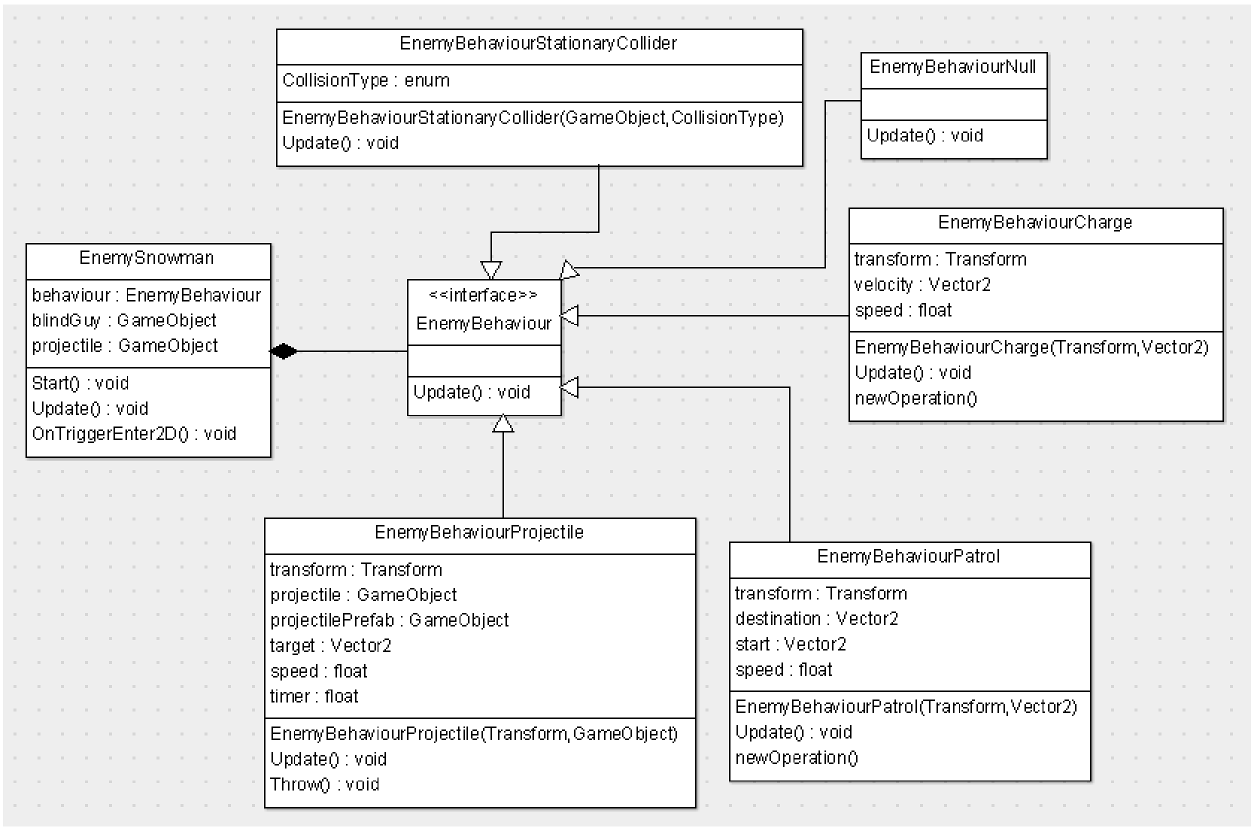
```
public class BarkPowerSubject : SubjectBase {

}
```
The BarkPowerSubject class requires no extra code to work, but we still have it so we can more easily use it as a component in Unity.

```
public class EnemySnowman : MonoBehaviour {
        private BarkPowerObserver barkObserver;
        void Start () {
                //create a new observer object and assign a callback function
                barkObserver = new BarkPowerObserver(BarkEvent);
                //add the observer to the list of subject observers for bark
                FindObjectOfType<BarkPowerSubject>().Attach(barkObserver);
        }

        void BarkEvent() {
                //attack the player when bark is triggered
                AttackPlayer();
        }
…
}
```
The snowman reacts to barks and goes into attack mode when the bark is heard.

Null Object

We are currently using the strategy pattern to define enemy behaviour at runtime, but what happens when we want an object to remain in the level but be harmless (have no behaviour)? One solution for this is to use a null object as a placeholder for enemy behaviour. We assign the null object as soon as the object is declared 'dead'.
Using the null object, we can still 'bring the object to life' if we ever decide to implement such behaviour. We could simply accomplish this by detecting whether the user is casting a fire attack on a 'dead' fire-based object and then call it back to life without having to instantiate a new version of it.

Structure



Implementation

```
public class LavaFall : MonoBehaviour {
	private EnemyBehaviour behaviour;
	public Sprite frozenSprite;

	void Start () {
		behaviour = new EnemyBehaviourStationaryCollider(
			this.gameObject,
			EnemyBehaviourStationaryCollider.CollisionType.Fire);
	}

	void Update() {
		behaviour.Update();
	}
	void OnTriggerEnter2D(Collider2D coll) {
		if (coll.gameObject.tag == "IceAttack") {
			//we don't want to destroy the object
			behaviour = new EnemyBehaviourNull();
			//we simply want it to no be harmful and change its sprite
			GetComponent<SpriteRenderer>().sprite = frozenSprite;
		}
	}
}
```
The lava fall object remains in the scene after it's destroyed, but it has no behaviour; it's just there as decoration

```
public abstract class EnemyBehaviour {
	public abstract void Update();
}
```
The EnemyBehaviour class again for reference

```
public class EnemyBehaviourNull : EnemyBehaviour {
	public override void Update() {
		//do nothing
	}
}
```
The null enemy behaviour does nothing in its update method

Coding Conventions

To our coding conventions we agree in following the next rules:

- When we create a variable we use camel casing

```
    private EnemyBehaviour behaviour;
    public Sprite frozenSprite;
```

- Our functions will start with capital letter and every other word will also be with capital letter

```
    public void AddHealth() {
              /* apply a new power-up and give the current
               * power-up as base to expand upon */
              powerup = new BlindGuyPowerupHealth(powerup);
        }

        public void AddViewDistance() {
              powerup = new BlindGuyPowerupViewDistance(powerup);
        }
```

- When you need brackets to get into a *"while"*, an *"if"* or a function the brackets will be next to it

```
    public void AddViewDistance() {
              powerup = new BlindGuyPowerupViewDistance(powerup);
    }
```

- Our classes will ever start with capital letters

```
public class LavaFall : MonoBehaviour {

public abstract class EnemyBehaviour {
```

- When we need to make a comment that comment will be ever above the thing you are going to comment

```
    void Start () {
              //create a new blind guy stats object
              blindGuy = new BlindGuyStats();
              //give the powerup a base to work with
              powerup = new BlindGuyPowerupBase(blindGuy);
              blindGuyAI = GetComponent<BlindGuyAI>();
        }
```