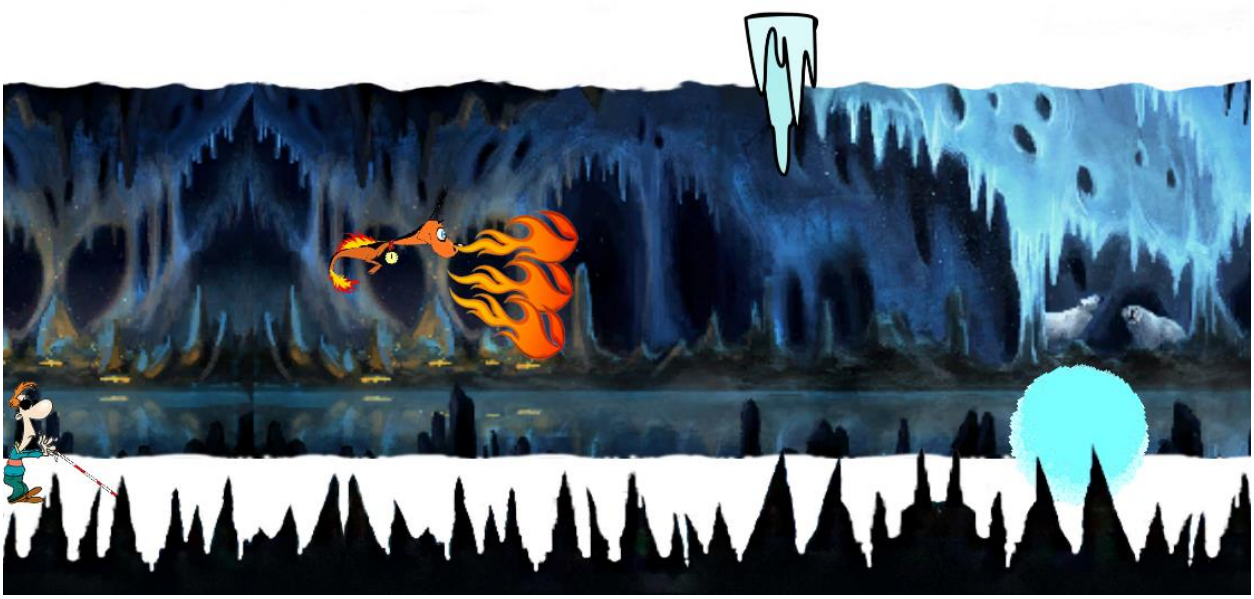
Blind Guide Redesign Patterns

Students:
Victor Fortea Martinez, 500747258
Wasili Prattis, 500689732

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.



Contents

Introduction

The game we picked for this assignment is Blind Guide. It's a game that was made by Wasili and his 2nd year project group. It was developed Unity and coded in C#. The reason we picked this game is because we also want to use it for our Automated Game Design project and because we wanted to see if we can redesign a game that was made with the same skill level we had about a year ago. This way we can at least prove to ourselves that we have learned something useful during this semester.

Powers

In Blind Guide, the player plays the role of a ghostly guide dog that wants to help his owner get home. There are multiple obstacles along the way, which can all be cleared using the player's special powers. The player has three different powers: fire, ice and telekinesis. Fire and ice can melt and freeze stuff, while the telekinesis power allows the player to pick up certain objects and throw them away.

Bark

Besides powers, the player also has the option to pause the blind guy momentarily using a bark ability. This gives the player more time to clear objects out of the way. Certain enemies in the game can react to this bark ability, causing them to go into offensive mode when the player uses it.

Power-ups

Initially, the blind guy can only take one hit during the entire game; taking a hit means game over. This can however change by picking up certain power-ups for the blind guy. Power-ups can provide various enhancements on the blind guy's behaviour, as well as give extra health. For now, power-ups can give the blind guy a boost in movement speed and the ability to look further ahead than normally. Taking a hit causes the blind guy to lose his most recent power-up.

Combination attacks

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. For example, some objects need to be melted first and then thrown outside the screen to destroy them.

Data collection

During a playthrough, data is collected and saved locally. The data is used at the end of each level to generate the content of the next level based on the player's playstyle. This behaviour is not fully implemented yet, but it will be implemented for the Automated Game Design course. Using this course to provide a better code base to work with will help us drastically in realising the dynamic content generation feature.

Optimization

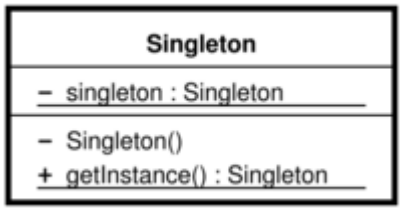
The game will eventually be uploaded to a website, where it will be playable as a browser game. We need to ensure that our game is as optimized as possible in order for it to work properly, because browser games tend to be slower than their executable counterparts. To realize this, we use patterns that are meant to optimize certain operations in a game.

Singleton

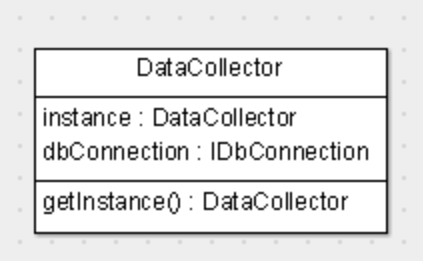
Problem: The data collector in the game needs to be accessible by all objects that need to store data locally. Storing the data can be done by calling a function inside the data collector class that opens a local database and writes data to it. Seeing as the local file should only be accessed by one instance of the data collector, it should be assured that only one instance of the data collector exists. Not doing so would lead to unpredictable results inside the local database.

Solution: The singleton pattern provides a way of making a class globally available, while also ensuring that there is never more than one instance of this class in the game. By checking whether the singleton is null before we return its instance, we can ensure that there is always an instance available when we need one by creating one when the instance is requested.

Structure



Singleton diagram, as found on https://en.wikipedia.org/wiki/Singleton_pattern



Our implementation of the singleton pattern

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

Problem: Our game is meant to work as browser game, which means we need to pay close attention to memory usage and performance. We want to find a way to put less pressure on objects that share the same properties in our game, because every bit of optimization will help us have a more polished game.

Solution: A lot of obstacles in our game share the same methods, which means we can apply the flyweight pattern to have them use less memory by having them share a base class that provides the methods they need.

Structure

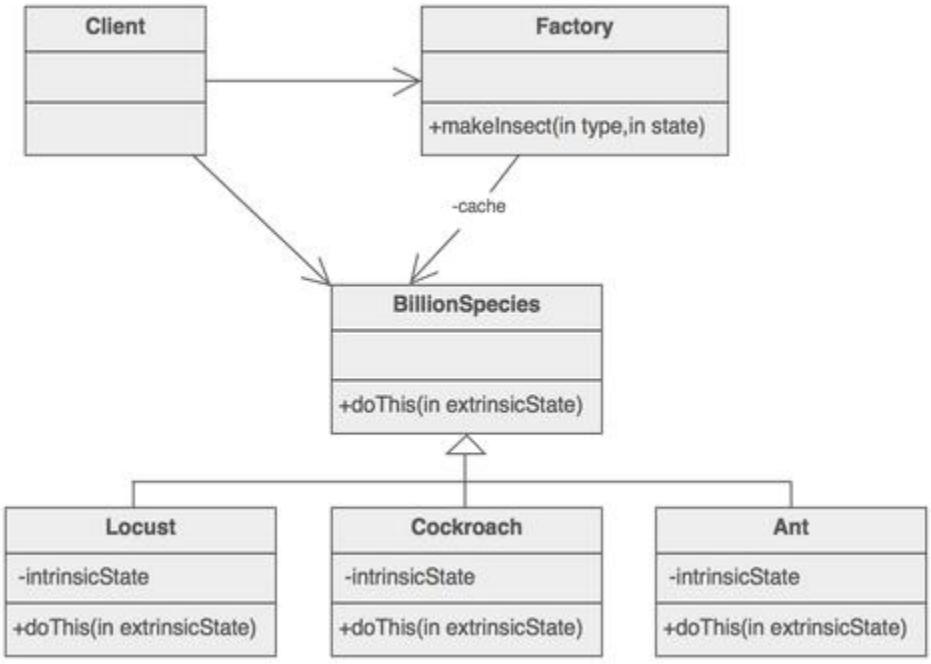
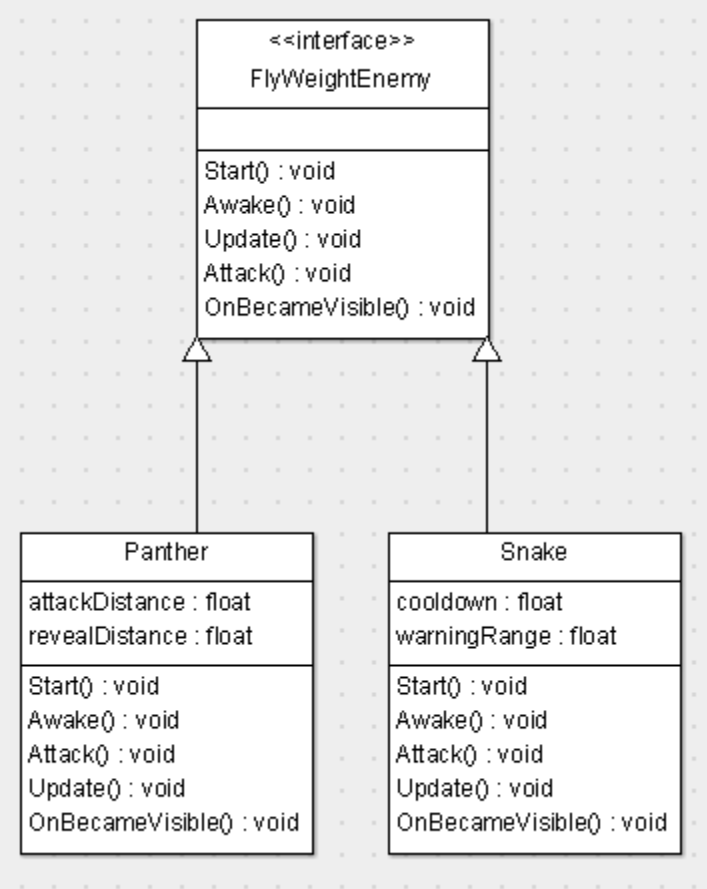


Diagram of the flyweight pattern, as found on https://sourcemaking.com/design_patterns/flyweight



Our implementation of the flyweight pattern. Enemies use FlyWeightEnemy's functions.

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

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

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

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

        dataMetric.obstacle = DataMetricObstacle.Obstacle.Panther;
    }

    public override void Update()
    {
        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;
        }
```

```
    }


    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();
    }
}
```

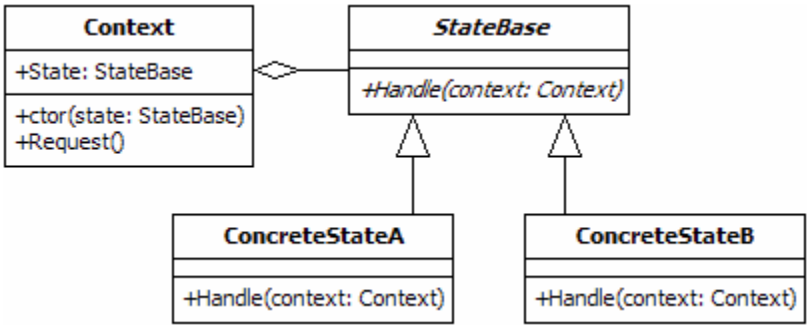
Another class that uses these methods is the panther, along with a lot of other enemies that define their own behaviour within these functions

State

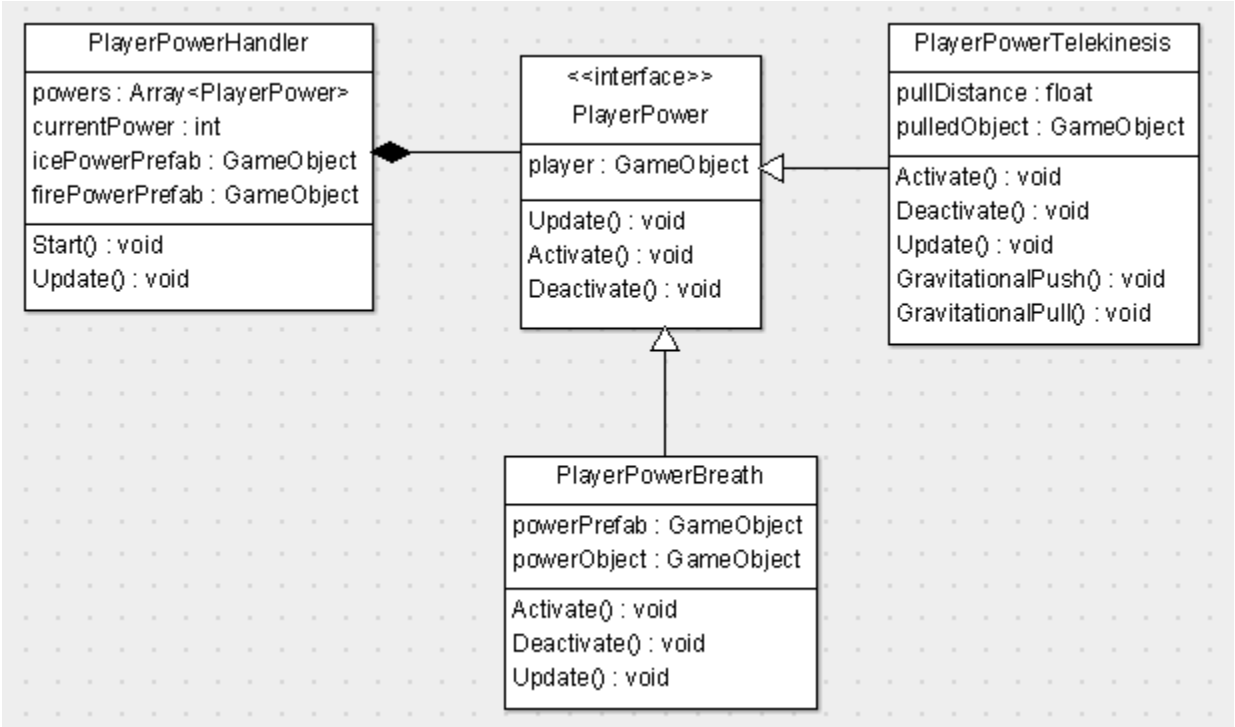
Problem: The player can choose to use one of three different powers, but only one power can be active at a time. We want behaviour of the powers to be defined in their own class, so we can easily add new powers in the future to expand our game.

Solution: Use the state pattern to decide which power the player is currently using and define the behaviour of this power within each power's own class. To prevent duplicate code, we have the fire and ice power share the same class: PlayerPowerBreath. The only difference here is the sprite of the power and the result of the power hitting an object, which is handled inside each obstacle separately. Sprites and collision can all be defined in a prefab, which will be passed along when the power is called. This way we combine the strength of the Unity engine with the strength of the state design pattern to simplify our code. We store the powers inside an array at initialization to ensure that the order of power switching is always the same, and handle activation and deactivation behaviour with custom functions.

Structure



The state design pattern, as described on http://www.blackwasp.co.uk/state.aspx



Our implementation of the state pattern. Fire and Ice powers share the same class.

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() {
		//pull objects towards the player
	}

	//called when power is deactivated
	private void GravitationalPush() {
		//push objects towards the mouse cursor position
	}
}
```

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

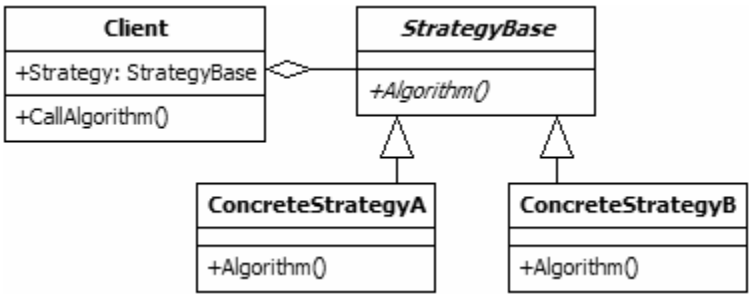
Problem: Enemies in the game have different behaviour, which can be changed during runtime by events such as collision with another object or being hit one of the player's powers. The behaviour is pretty standard for all enemies, because enemies can either:

- Shoot projectiles towards the blind guy
- Patrol between two points until the player is near or an event is triggered
- Stand still and simply detect collisions with the blind guy
- Charge towards the blind guy and deal damage upon collision

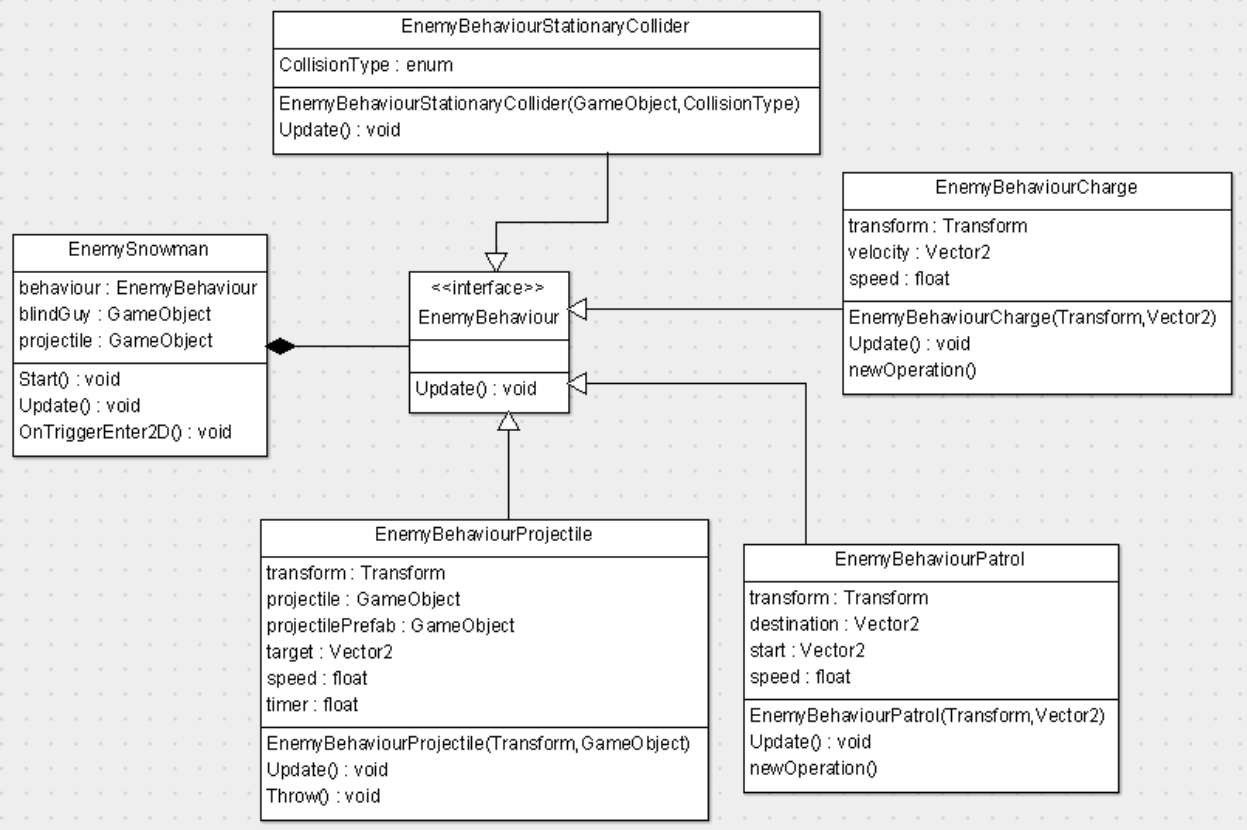
We want to implement these behaviours with future additions in mind.

Solution: Use the strategy pattern to assign behaviours to each enemy, without having to worry about duplicating tons of code for all enemies that share the same behaviour. This also allows us to change behaviour on the fly, so events such as being hit a player power can cause behaviour to change with a single reassignment of the behaviour variable.

Structure



Strategy pattern example found on http://www.blackwasp.co.uk/strategy.aspx



Our implementation of the strategy pattern. The snowman has multiple behaviours it uses during its lifetime.

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;
			//change behaviour to stationary collider
			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;
                }
        }
}
```

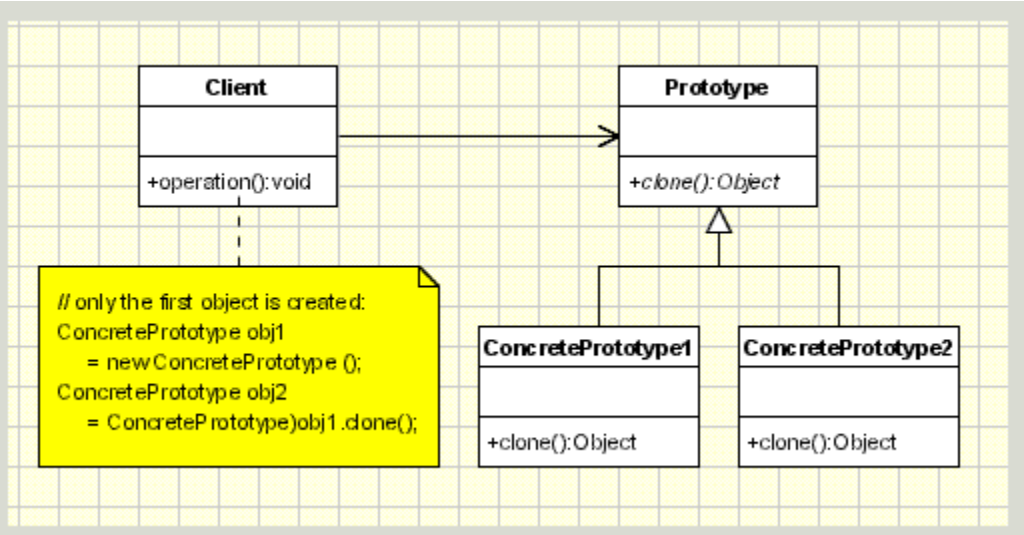
When using the patrol behaviour, the player does not need to be attacked, the enemy simply movies between two points

Prototype

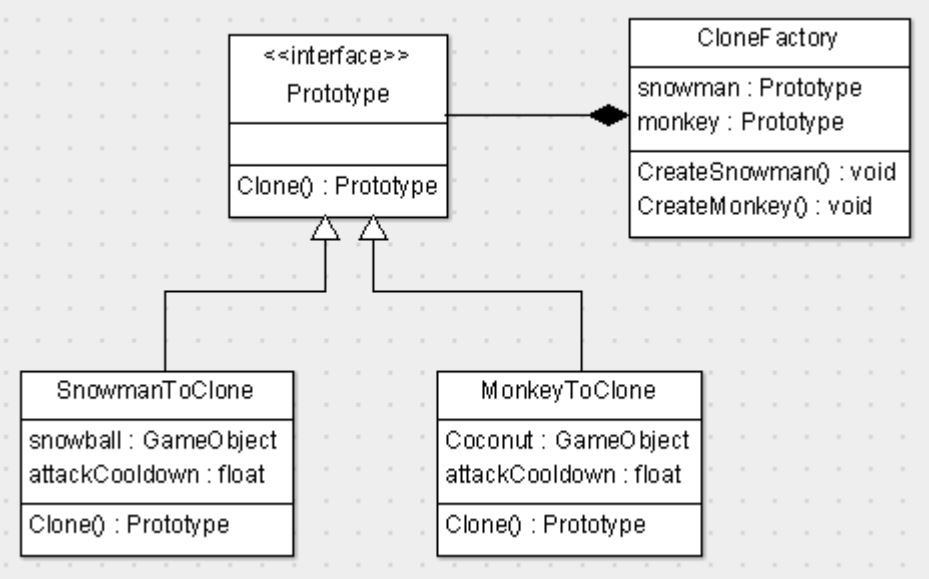
Problem: During the content generation process, we want levels to be loaded as quickly and efficiently as possible, because browser games tend to be slow by nature. Optimizing this loading process as much as we can, is essential for getting the game experience to be as smooth as possible.

Solution: Seeing as the content generator will be instantiating objects multiple times per level, we can optimize the instantiation process by using the prototype pattern: we create an object once and then we simply place a clone of it in a different position whenever we want to place the same object somewhere else.

Structure



Description of the prototype pattern found on http://www.oodesign.com/prototype-pattern.html



Our prototype pattern allows the CloneFactory to clone existing objects with Prototype as their base class.

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

Problem: We want to be able to track and unlock achievements without adding the code for it to objects that can trigger such achievements. We need a way to have objects do their thing, without having to worry about achievements until an event is triggered that requires action to be taken.

Solution: The use of the mediator pattern allows us to have a middle-man that handles communication between objects and the achievement system. By using a mediator, we can also add other features in the future, such as an online save function, progress notifications, sound effects and social media notifications without having to change the base code of each object. All we need to do is tell the mediator that something happened and the mediator will take care of the rest for us.
Once the achievement tracker, or any other system that is triggered by the event, wants to communicate a new message back to the triggering object, we can do so through the mediator again. All we need to do is pass along the instance of the object that triggered the event.

Structure

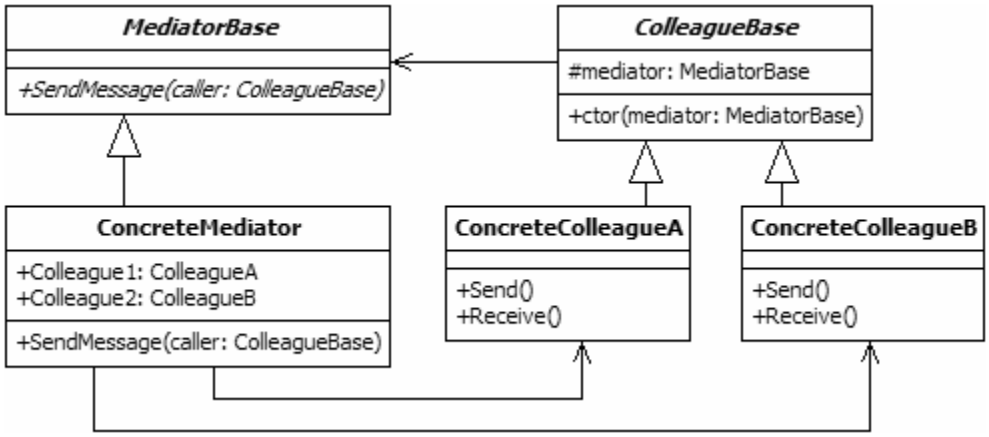
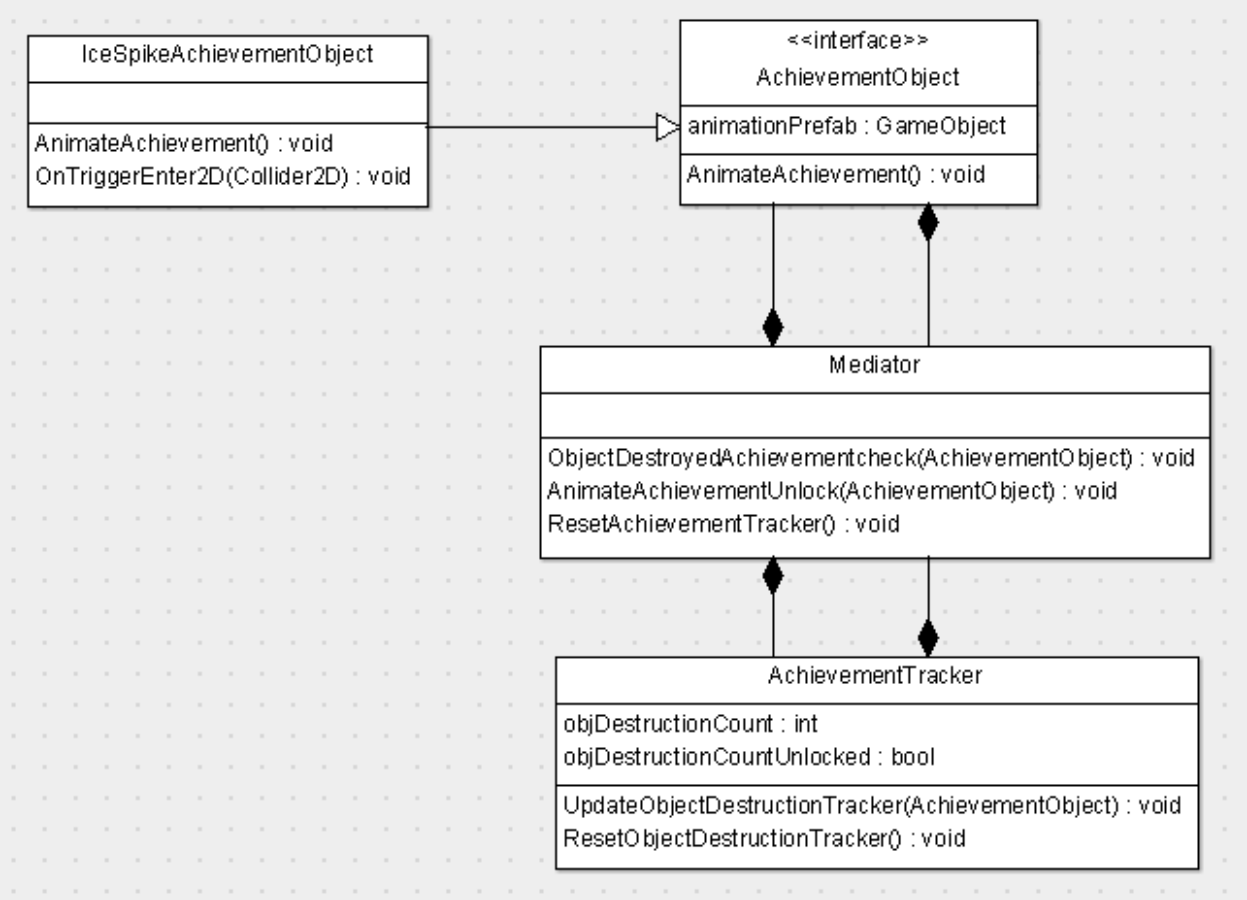


Diagram of the mediator pattern, as described on http://www.blackwasp.co.uk/mediator.aspx



Our implementation of the mediator pattern, without a mediator base class.

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;
	}
}
```

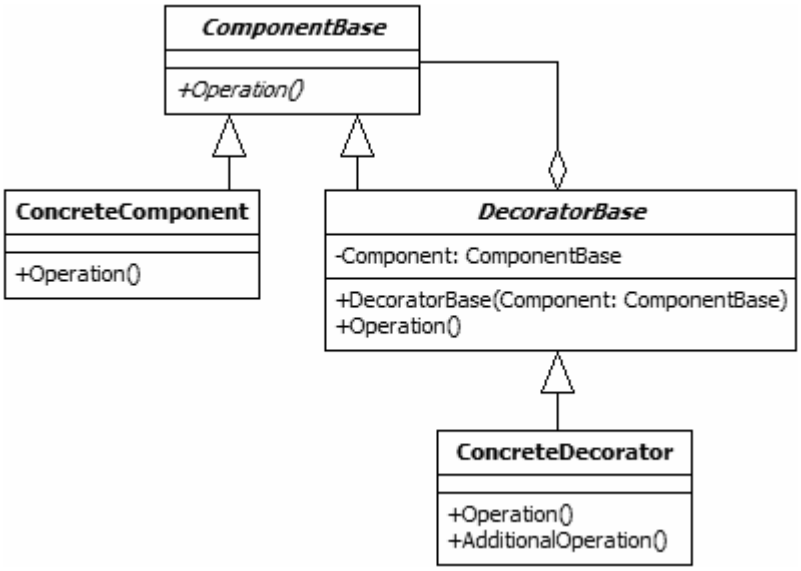
The achievement tracker decides when an achievement unlock trigger should be called, based on the data is has collected so far

Decorator

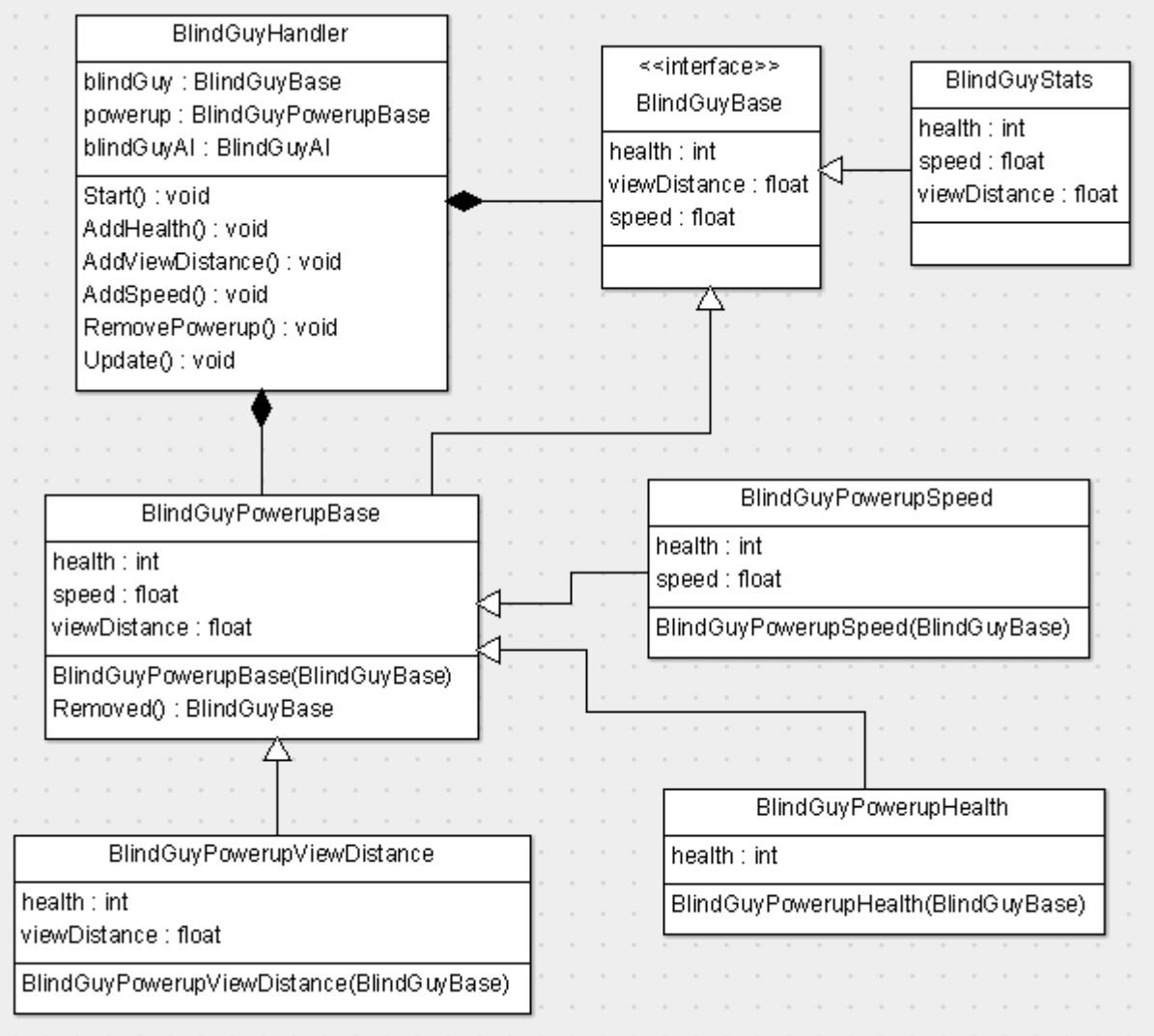
Problem: The player can pick up power-ups for the blind guy. Power-ups enhance certain values of the blind guy and always increase the blind guy's heath by one. We want to implement power-ups without having to keep track of them inside the base code of the blind guy. We also want the blind guy to lose his last power-up without adjusting the remaining acquired power-ups.

Solution: The decorator pattern allows us to enhance behaviour without touching the base code of the behaviour. When a player picks up a power-up, we add a decorator to our blind guy that increases the desired values. By using the decorator pattern's structure, we can easily remove the current power-up and still keep the remaining power-ups. Our handler class can keep track of the power-ups we have and also pass the values on to the class that actually updates the blind guy's behaviour.

Structure



Overview of the decorator pattern, found on http://www.blackwasp.co.uk/decorator.aspx



Our implementation of the decorator pattern, inspired by the example given on blackwasp.co.uk

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;
	}
}
```

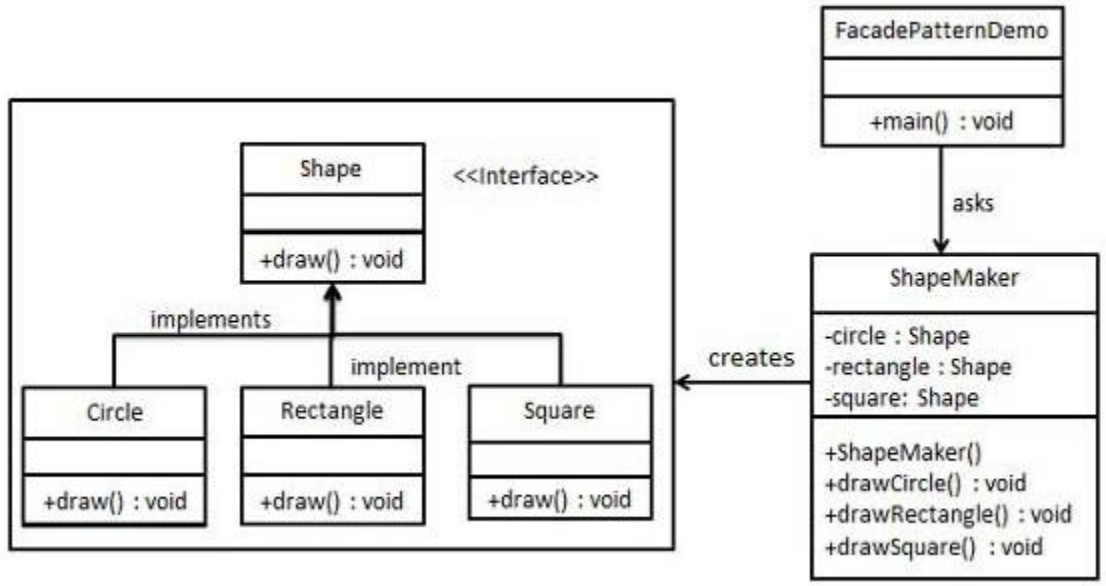
Added to the blind guy as a component, this class handles all communication between power-ups and the components that use them

Façade

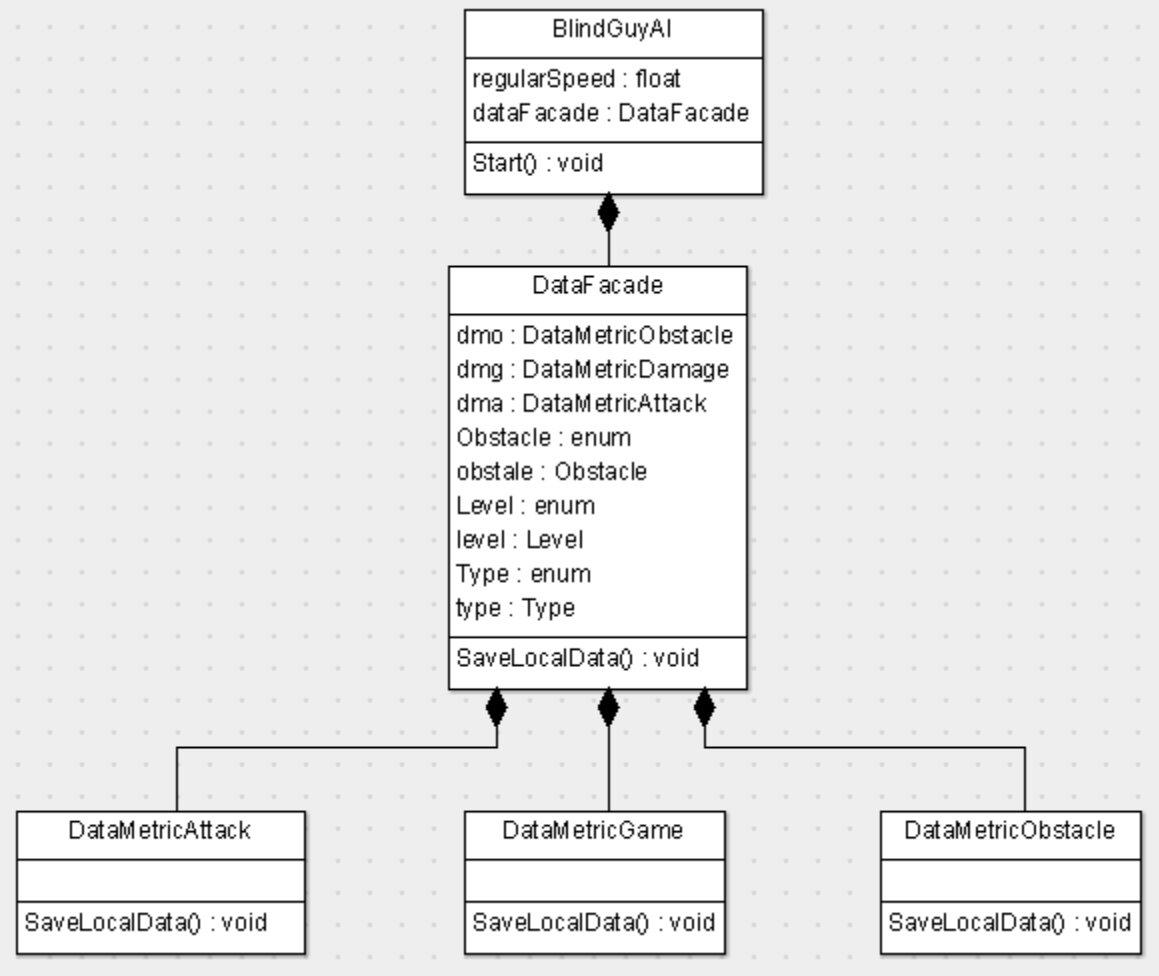
Problem: Data collection is a big feature that requires a lot of classes to save data locally. If we were to call the data collector class every time data needs to be saved, we would be opening a connection to a database nearly every second. This would obviously be very painful on the performance of the game, so we need a way to control when data is saved locally, while still being able to collect it whenever an event is triggered. We also do not want to complicate the code by handling the local data saving inside each object's class.

Solution: The façade pattern allows us to separate the complex local saving data from the object code. When an event is triggered. We tell the façade class that we want to save some data. We send the data to the façade, which then keeps the data locally until it is time to send the data over to the data collector class and save it in a database.

Structure



Façade example from http://www.tutorialspoint.com/design_pattern/facade_pattern.htm



Example of the blind guy using the DataFacade class to save local data.

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;
        //for obstacle and attack
        public int gameID;
        //class Game
        public enum Level { Tutorial, Fire1, Fire2, Fire3, Ice1, Ice2, Ice3, Jungle1,
Jungle2, Jungle3 }
        public Level level;
        //class Attack
        public enum Type { Fire, Ice, Telekinesis, Destruction }
        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 {
    DataFacade dataFacade = new DataFacade();

    void Start () {
        dataFacade.starttime = System.DateTime.Now.ToString();
        dataFacade.level = (DataFacade.Level) Application.loadedLevel;
    }
}
```

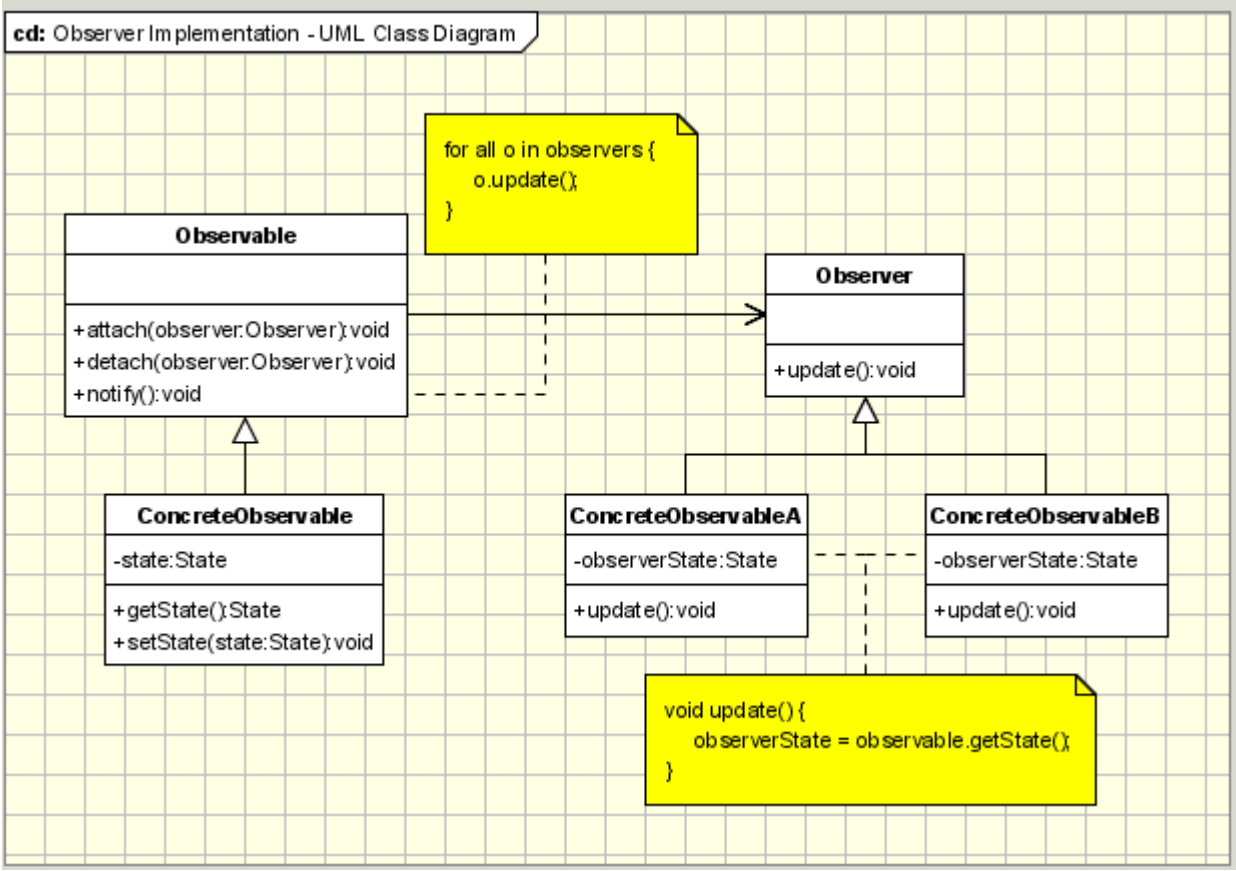
Blind guy is one of the examples of saving data using the façade

Observer

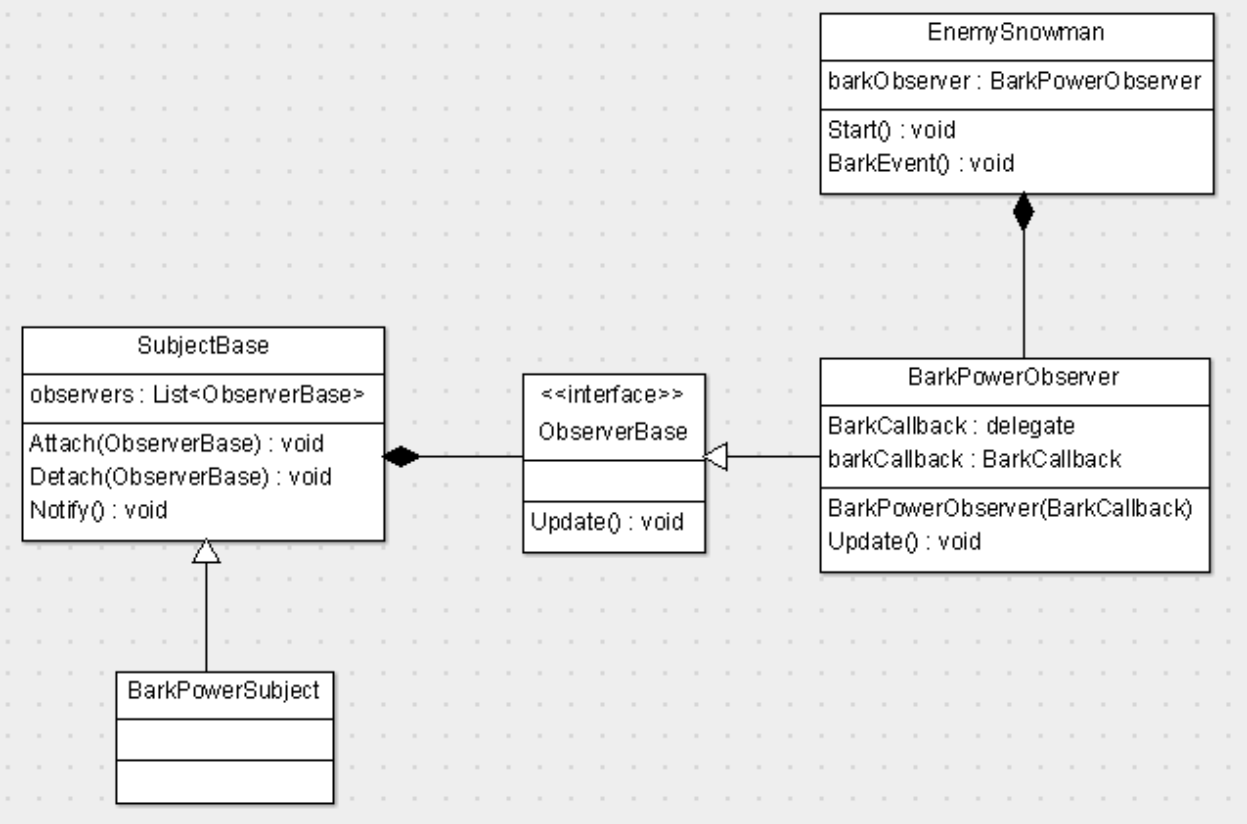
Problem: Tell certain objects that the player has triggered his bark ability, without having to notify all objects in the scene about this event. Only a couple of specific objects need to be notified, so we want to minimize the amount of receivers for this call as much as we can.

Solution: Use the observer pattern to register objects that react to the bark ability as observers. When the bark ability is used, these observers will get their assigned callback method triggered so they can execute their desired behaviour. By keeping track of the observers in a list, we make sure that only the objects that can react to the bark ability will get notified by it. By having the observer provide a callback method, we always ensure that the right function is called.

Structure



Example of observer pattern, found on http://www.oodesign.com/observer-pattern.html



Our implementation of the observer pattern uses a callback method to notify observers.

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();
        }
…
}
```

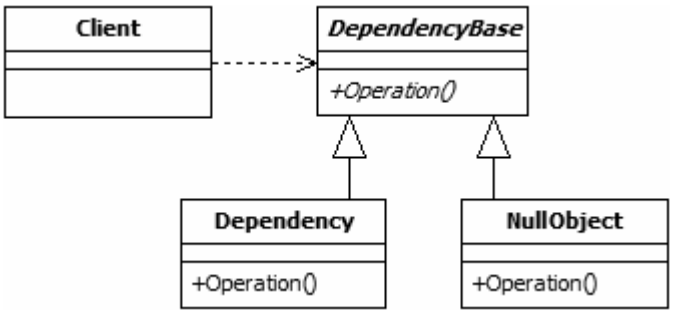
The snowman reacts to barks and goes into attack mode when the bark is heard.

Null Object

Problem: The strategy pattern used for handling enemy behaviours assumes that there is always a behaviour available. However, some objects simply do nothing once they have been destroyed; a static sprite with no collision could be sitting there with no interaction available for it. We do not want this sort of object to be replaced by a new game object, though. We want some objects to be able to recover from their 'defeated' state as a result of certain events, such as the player spitting fire on a defeated fire enemy to bring it back to life. Thus, we need to find a way to allow objects to have empty behaviours.

Solution: The null object pattern allows an object to be null, without causing null reference exceptions when their function is called. We can use this behaviour to create a 'dummy' behaviour with an empty update method. This way, we can simply reassign the null behaviour to a new behaviour type when the object is revived.

Structure



Null Object pattern found on http://www.blackwasp.co.uk/NullObject.aspx

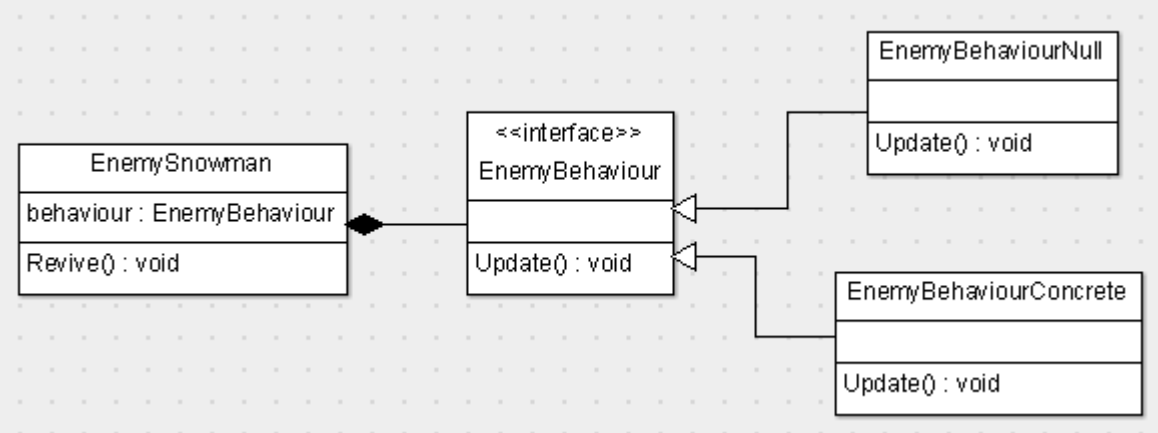


Diagram of the implementation of the null object pattern in our game. Less detailed than strategy for readability.

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

Since coding conventions was one of the things listed in the criteria, here are the rules we used when we implemented our design patterns:

- 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>();
        }
```

Conclusion

The game was already finished and fully functional in Unity before we started. However, we decided to use this game for our Automated Game Design project, which forced us to do some big overhauls in the code in order to get it working properly with our changes. We also added some new features and had to optimize the game to get it working as a browser game. Picking this game as our Design Patterns project allowed us to make it easier for us to add the Automated Game Design changes we wanted in the game. So for us, this was like killing two birds with one stone.

Changes

Most of the patterns required the reconstruction of entire classes to get them to work properly. For example, all enemies had to be reworked in order to have them make use of the strategy design pattern for defining their behaviour more easily. The redesign of the enemies took a lot less time than the initial programming required to get them to work when the project was developed by a team of 6 second-year students. The difference in time is so significant that I wish this course was given to us in our first or second year at the HvA.

Additions

Unity allows us to easily add and remove component from our objects, which made is easier for us to add patterns to existing objects without having to worry about breaking existing code. This allowed us to add new features that make the game more fun, while also allowing us to implement the patterns we wanted to implement because they seemed interesting.

Sources

The internet provided us with a lot information on all patterns, and there were many different sources we could pick examples from. We made sure to read at least two different sources each time we wanted to implement a pattern, in order to ensure that we understand the pattern correctly. The source we used the most was http://www.blackwasp.co.uk/gofpatterns.aspx. This website provides nice examples of each pattern, with code samples written in C#. They also provide both a general example and a specific implementation for each pattern, which makes it easier to understand.

Last words

The knowledge we gained from this course is, in our opinion, essential for us as programmers. Knowing how to apply patterns to get better structure in your code base and being able to tell when a pattern should or should not be used is what differentiates a good programmer from a code monkey. Seeing as we used a project that represents our level of expertise about one year ago, we are happy to see the big difference in readability and performance between the old version and the redesigned version of the game. It makes us feel like we've learned something useful, something we'll remember for the rest of our career, rather than something that we'll forget about within a few months.

End of report