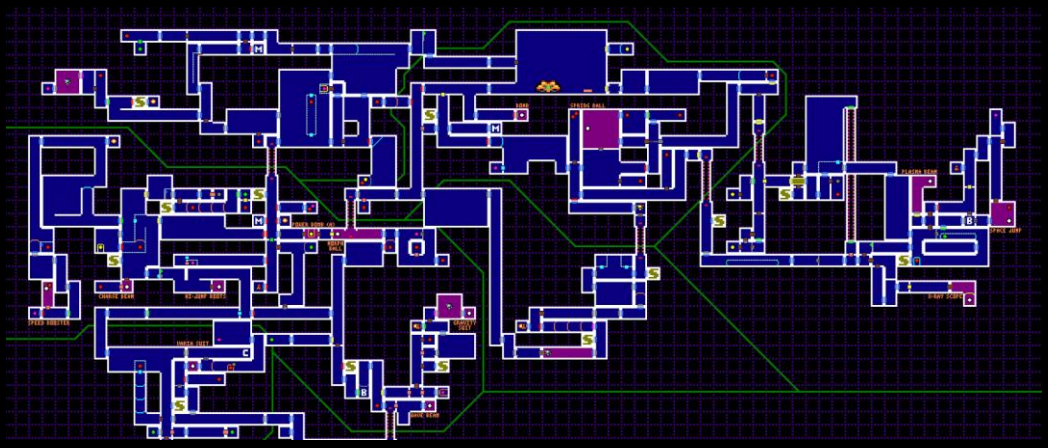


MetroidVania

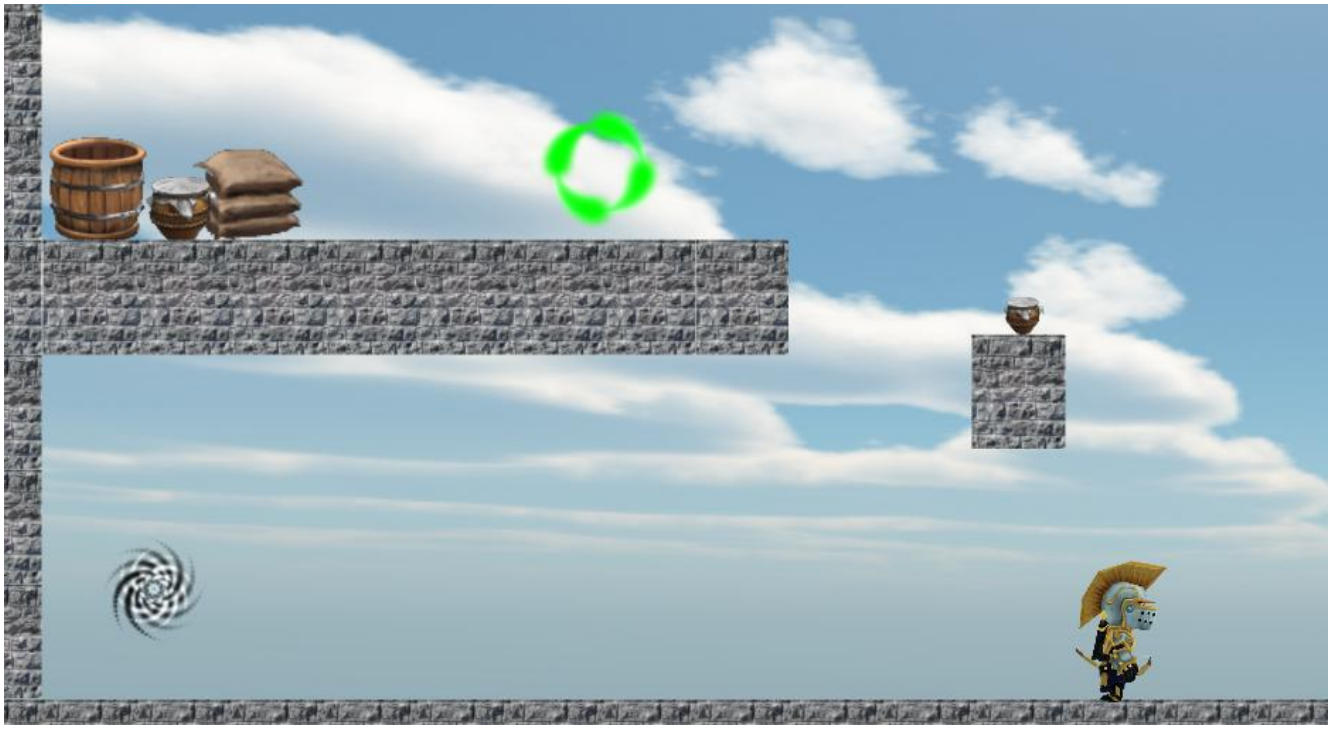
MetroidVania	
Bas van Rijn	**500531907**
Dimitri Meister	**500639631**
José Boon	**500651079**
Maikel van Munsteren	**500643698**

Date: 16-06-2015
Version: 1.0

Table of Contents

Introduction

This document describes the architecture of the game our team created during the course Design Patterns. The document describes which patterns were used/applied.
For each pattern there will be a description of its usage, or motivation, detailing the reasoning behind the implementation. There will also be an UML diagram and, where applicable, some code snippets.



GOF patterns

This section describes the GOF patterns we used during development of the game.

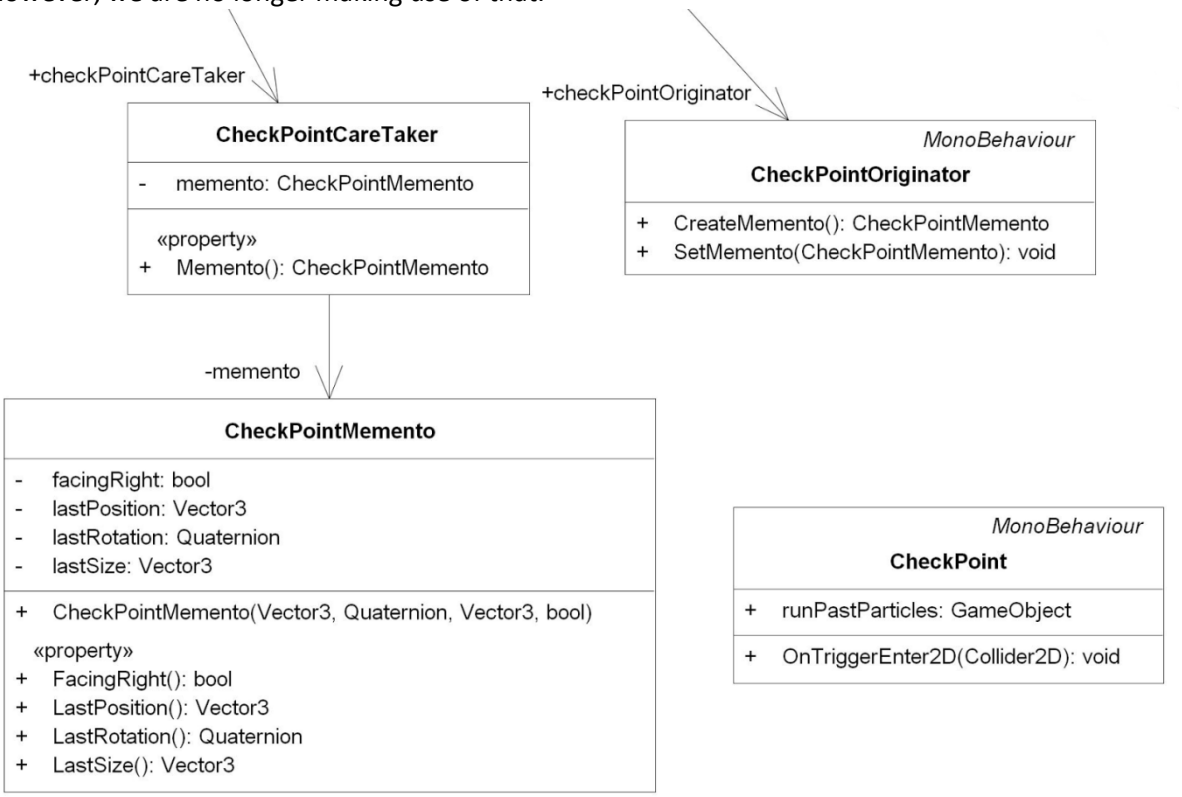
Memento

Usage:

This Pattern is used to create checkpoints for the player. The player can ‘walk over’ the checkpoint and, when he dies, respawn at the last checkpoint. While the memento can be used to create copies of the player (for example) we only save the player's orientation (position,rotation,scale). Since that is the only real variable we want to maintain.

UML:

The uml of the Memento pattern. Those lines are connected to the GameManager Singleton. Where they are both being used. The Checkpoint class is a MonoBehaviour that triggers the SetMemento that is available through the GameManager as seen in the snippets.
The facingRight is an old variable that still existed when the UML were created. In the snippets however, we are no longer making use of that.



Snippets:

```
40      public static void setCheckPoint()
41      {
42          if(instance.checkPointOriginator ==null)
43              instance.checkPointOriginator = GameObject.FindGameObjectWithTag("Player
44          if(instance.checkPointCareTaker == null)
45              instance.checkPointCareTaker = new CheckPointCareTaker();
46          instance.checkPointCareTaker.Memento = instance.checkPointOriginator.CreateM
47      }
48
49      public static void restoreCheckPoint()
50      {
51          instance.checkPointOriginator.SetMemento(instance.checkPointCareTaker.Memento
52      }
```

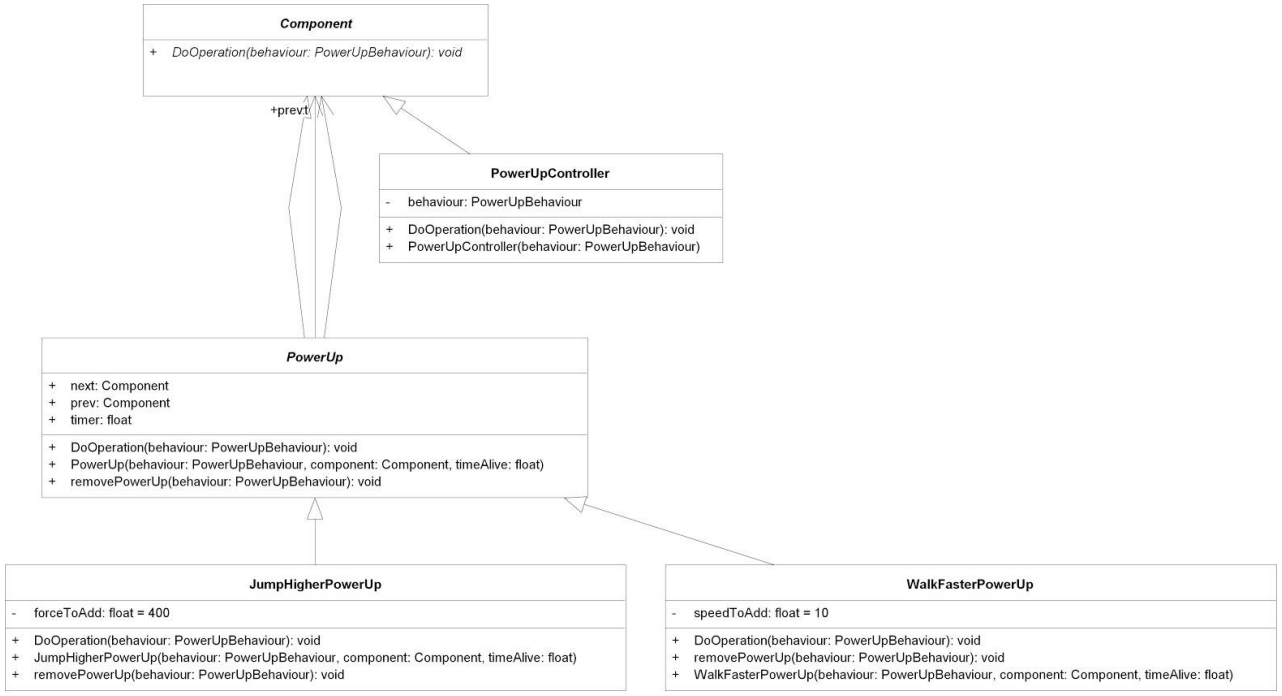
```
5  public class CheckPointOriginator : MonoBehaviour {
6
7      public CheckPointMemento CreateMemento()
8      {
9          return new CheckPointMemento(transform.position,transform.rotation,transform
10     }
11
12     public void SetMemento(CheckPointMemento memento)
13     {
14         transform.position = memento.LastPosition;
15         transform.rotation = memento.LastRotation;
16         transform.localScale = memento.LastSize;
17     }
18 }
```

Decorator

Usage:

The Decorator pattern is used for power ups. Our game lets the player be ‘a bit stronger’ for a moment. We currently have the power up to move faster and to jump higher. While creating this pattern we found out that it is not the best way to implement this pattern. Our powerups are temporary and we found out that the way a decorator works, is actually semi-permanent. You can make different combinations with the Decorator pattern but when you are done creating it is not designed to reverse the creation. For example: you can create coffee with milk and sugar, but when you decide you don’t want milk in your coffee, it is impossible(in real life) to get the milk out of your coffee. We worked around it and are now capable of removing the milk.

UML:



Snippets:

```
18      void Start ()
19      {
20          controller = new PowerUpController(this);
21          character = GetComponent<Player>();
22      }
23
24      // Update is called once per frame
25      void Update ()
26      {
27          controller.DoOperation(this);
28      }
29
30      public void AddPowerUp(PowerUpType type,float duration)
31      {
32
33          switch(type)
34          {
35              case PowerUpType.JUMPHIGHER:
36                  controller = new JumpHigherPowerUp(this,controller,duration);
37                  break;
38              case PowerUpType.WALKFASTER:
39                  controller = new WalkFasterPowerUp(this,controller,duration);
40                  break;
41              default:break;
42          }
43      }
```

```
6       public abstract class PowerUp : Component
7       {
8           public Component next;
9           public Component prev;
10          public float timer;
11          public PowerUp(PowerUpBehaviour behaviour,Component component,float timeAlive)
12          {
13              prev = component;
14              if(prev.GetType()==(typeof(PowerUp)))
15              {
16                  (prev as PowerUp).next = this;
17              }
18              timer = timeAlive;
19          }
20
21          public override void DoOperation(PowerUpBehaviour behaviour)
22          {
23              prev.DoOperation(behaviour);
24              timer-=Time.deltaTime;
25              if(timer<=0)removePowerUp(behaviour);
26          }
27
28          public virtual void removePowerUp(PowerUpBehaviour behaviour)
29          {
30              if(next != null && prev.GetType()==(typeof(PowerUp)))
31              {
32                  (prev as PowerUp).next = next;
33              }
34              if(prev != null && next != null)
35              {
36                  (next as PowerUp).prev = prev;
37              }
38              if(prev.GetType()==(typeof(PowerUpController)) && next == null)
39              {
40                  behaviour.controller = prev;
41              }
42          }
43      }
```

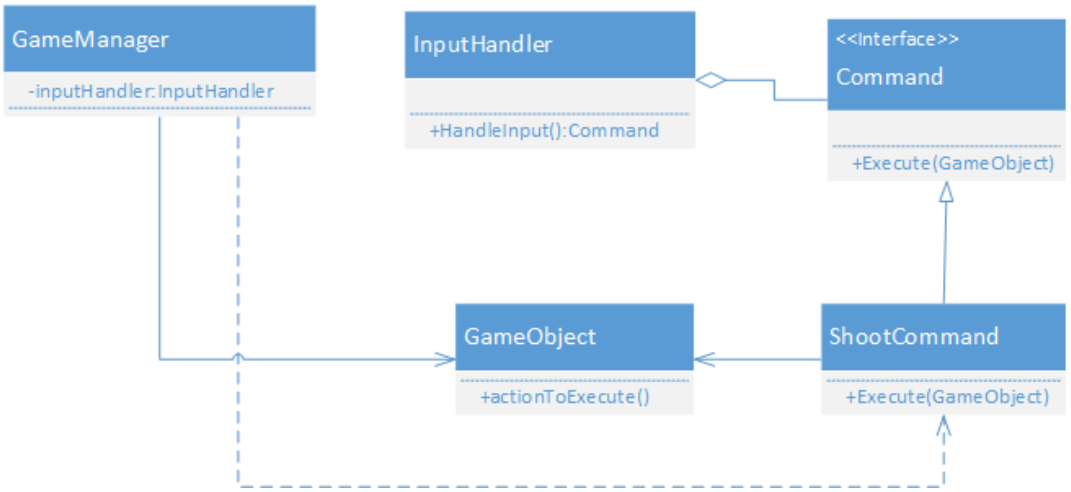
Command

Usage:

This pattern is used for controlling the player. Command makes adding new functionality to an gameactor relatively easy by creating a new object that encapsulates this functionality. One of our original plans was the ability to change the current player actor for a different actor, with different abilities. The player would then have make smart use of each controllable actor to progress through the game. The Command pattern makes it easy to swap between controllable actors.

UML:

For consideration of screen space we only display one implemented command, in this case the ShootCommand.



Snippets:

The execute function is probably the most interesting here as we make usage of the component pattern Unity uses here. We use the generic GameObject class as a parameter and fetch the required components we need from that object:

```
public void Execute (GameObject obj)
{
    ShootBehaviour script = obj.GetComponent<ShootBehaviour> ();
    if(script)
        script.Shoot();
}
```

The way we pass an GameObject to the execute function means that we can use it on any valid GameObject, granted that it has the required components.

We process the commands in the FixedUpdate function inside the GameManager and pass it the (current) player object.

```
Command command = inputHandler.HandleInput();
//check if there is ANY type of command being executed.
if(command!=null){
    command.Execute(player);
}
```

This would make our original intended plan of player swapping an easy task as we simply could have changed the object we pass in the Execute function.

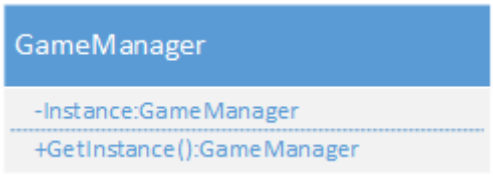
Singleton

Usage:

The Singleton is used for the game manager and the object pool. For both the game manager and the object pool there can only be one instance available in any scene. Both are queried relatively often and have only one instance, so the Singleton makes for an easy pattern to conveniently access these objects.

UML:

The Singleton has a rather small diagram:



Snippets:

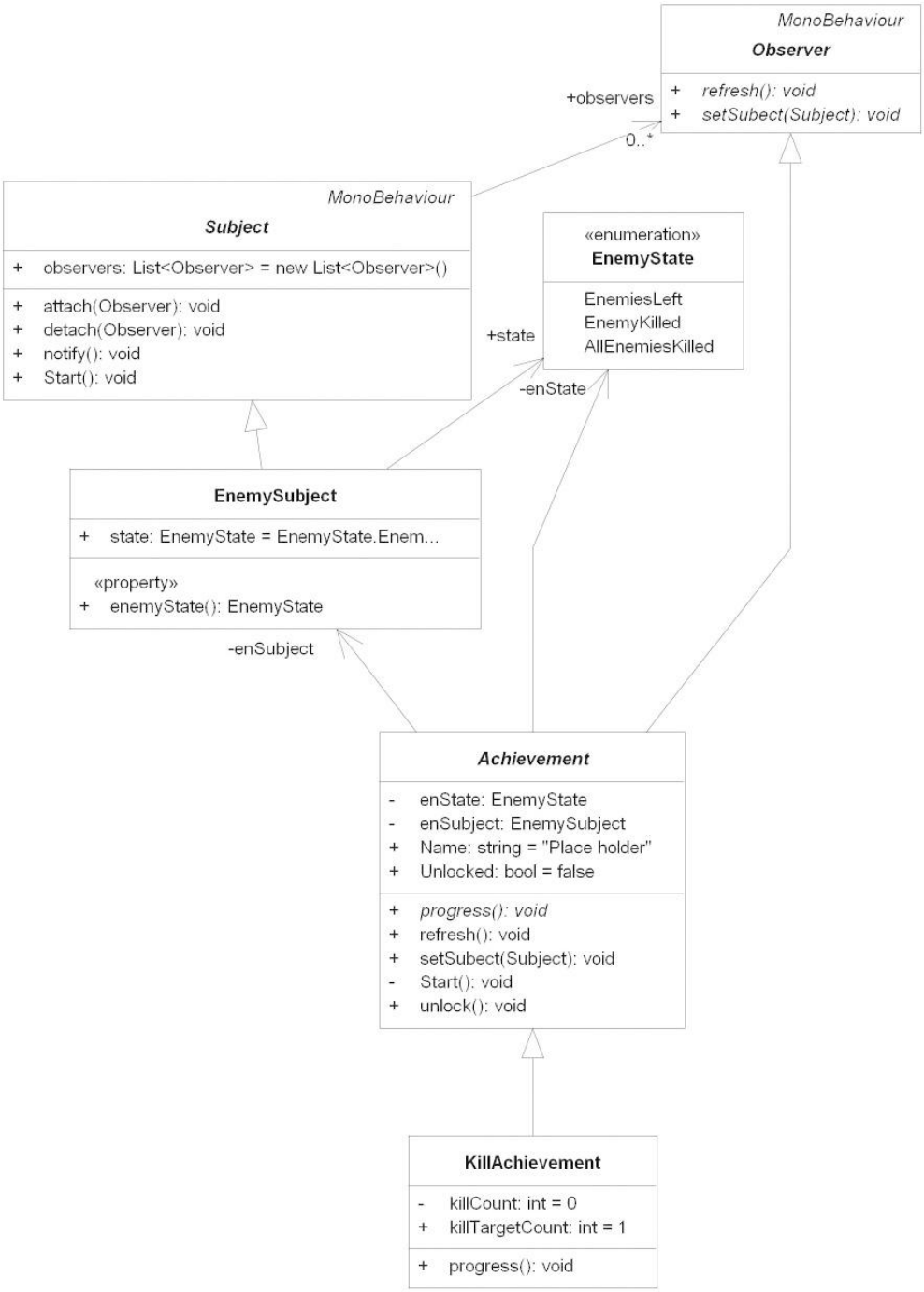
```
//Singleton instance
private static GameManager instance =null;

//Get the instance with an property
public static GameManager GetInstance{
    get{
        //If there is no instance yet, find it in the scene hierarchy
        if(instance == null){
            instance = (GameManager)FindObjectOfType(typeof(GameManager));
        }
        //Return the instance
        return instance;
    }
}

void Awake()
{
    instance = this;
}
```

Observer

Usage:

The observer pattern is being used for the achievement system. When the state of an object changes it should notify the attached observers. In our case when we have an enemy that has been killed it will notify the kill related achievement(s).

UML:



Snippets:

Attaching the achievements to our enemy subject.

```
public class AchievementManager : MonoBehaviour
{
    void Start ()
    {
        Achievement[] kAchievs = GameObject.Find("KillRelatedAchievements").GetComponents<Achievement>();
        GameObject enemies = GameObject.Find("Enemies");

        if(kAchievs.Length == 0 || !enemies)
            return;

        EnemySubject eSub = enemies.GetComponent<EnemySubject>();

        if(eSub != null)
        {
            for (int j = 0; j < kAchievs.Length; j++)
            {
                eSub.attach(kAchievs[j]);
            }
        }
    }
}
```

When an enemy has been killed it will change the subject state.

```
GameObject obj = GameObject.Find("Enemies");
if(obj)
{
     EnemySubject eSub = obj.GetComponent<EnemySubject>();
     if(eSub)
          eSub.enemyState = EnemyState.EnemyKilled;
}
```

Changing its state will also call the notify() function which loops through all the attached observers (achievements) and update them by calling the refresh() function.

```
public class EnemySubject : Subject
{
    // State is visible in inspector
    public EnemyState state = EnemyState.EnemiesLeft;

    // Set function also calls notify()
    public EnemyState enemyState
    {
        get { return state; }
        set
        {
            if(state != value)
            {
                EnemyState oldState = state;
                state = value;
                notify();

                if(transform.childCount > 0)
                    state = oldState;
            }
        }
    }
}
```

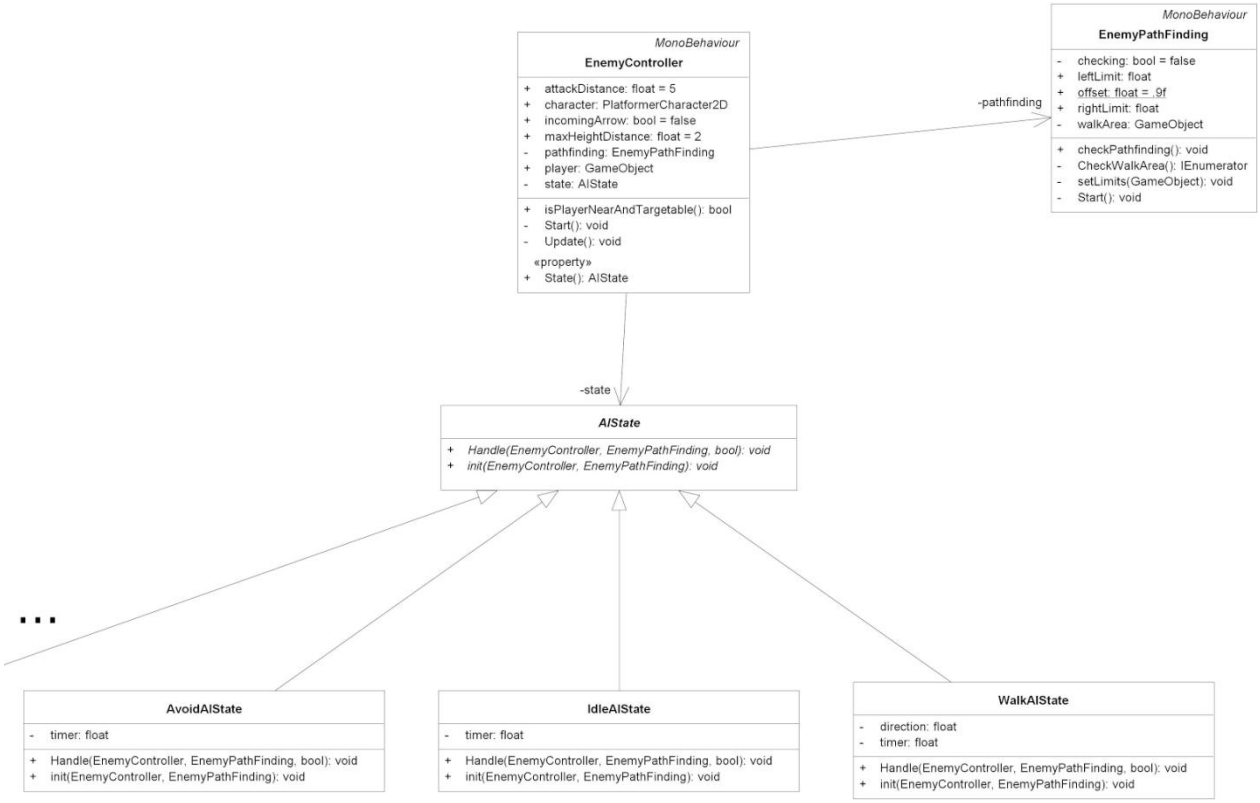
States pattern

Usage:

We use the State pattern for the AI of the enemies. The states describe the behaviour of the enemy when in that state and the determine when the enemy enters another state.

UML:

This is the UML, there is one more State on the left but that would be unreadable. The enemy also makes use of an EnemyPathFinding Object. This is just to see where it can and cannot walk.



Snippets:

```
20      // Use this for initialization
21      void Start () {
22          player =  GameObject.FindGameObjectWithTag("Player");
23          State = new IdleAIState();
24          pathfinding = GetComponent<EnemyPathFinding>();
25          character = GetComponent<PlatformerCharacter2D>();
26      }
28      // Update is called once per frame
29      void Update ()
30      {
31          //Currently disabled since it fails..
32          //state.Handle(this,pathfinding,incomingArrow);
33          state.Handle(this,pathfinding,false);
34          incomingArrow = false;
35      }
```

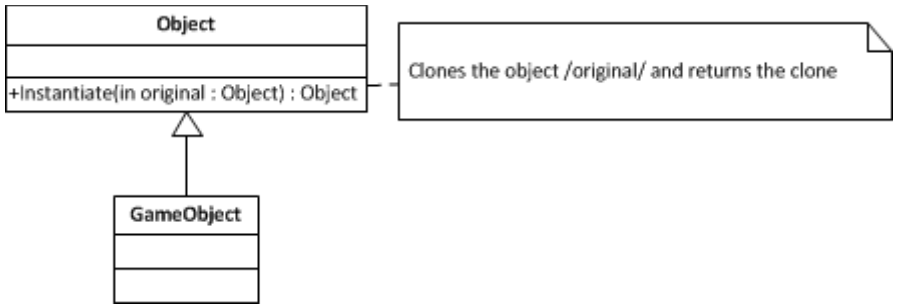
```
1  using UnityEngine;
2  using System.Collections;
3
4  public class IdleAIState : AIState
5  {
6      private float timer;
7      public override void init(EnemyController controller,EnemyPathFinding pathfinding)
8      {
9          timer = Random.Range(1,4);
10     }
11
12     public override void Handle(EnemyController controller,EnemyPathFinding pathfinding,bool incomingArrow)
13     {
14         timer-= Time.deltaTime;
15         //Check if we need to switch states
16         if(incomingArrow)
17             controller.State = new AvoidAIState();
18         else if(controller.isPlayerNearAndTargetable())
19             controller.State = new AttackAIState();
20         else if(timer<=0)
21             controller.State = new WalkAIState();
22
23         //Handle the state we are in
24         controller.character.Move(0,false,false,false);
25     }
26 }
27
```

Prototype

Usage:

The Prototype pattern is a pattern that is already within the Unity engine. We use it(Prefabs) to create curtain objects that we needed more than once. In our objectpool we use this pattern to create multiple clones of our objectPool prefab. In our case it was an arrow but it could be any object that we stored in a prefab.

UML:



Snippets:

We will start with the variable pooledObject. That is the prefab.

```
public GameObject pooledObject;
```

Instantiate ‘clones’ the prefab and places it in the game.

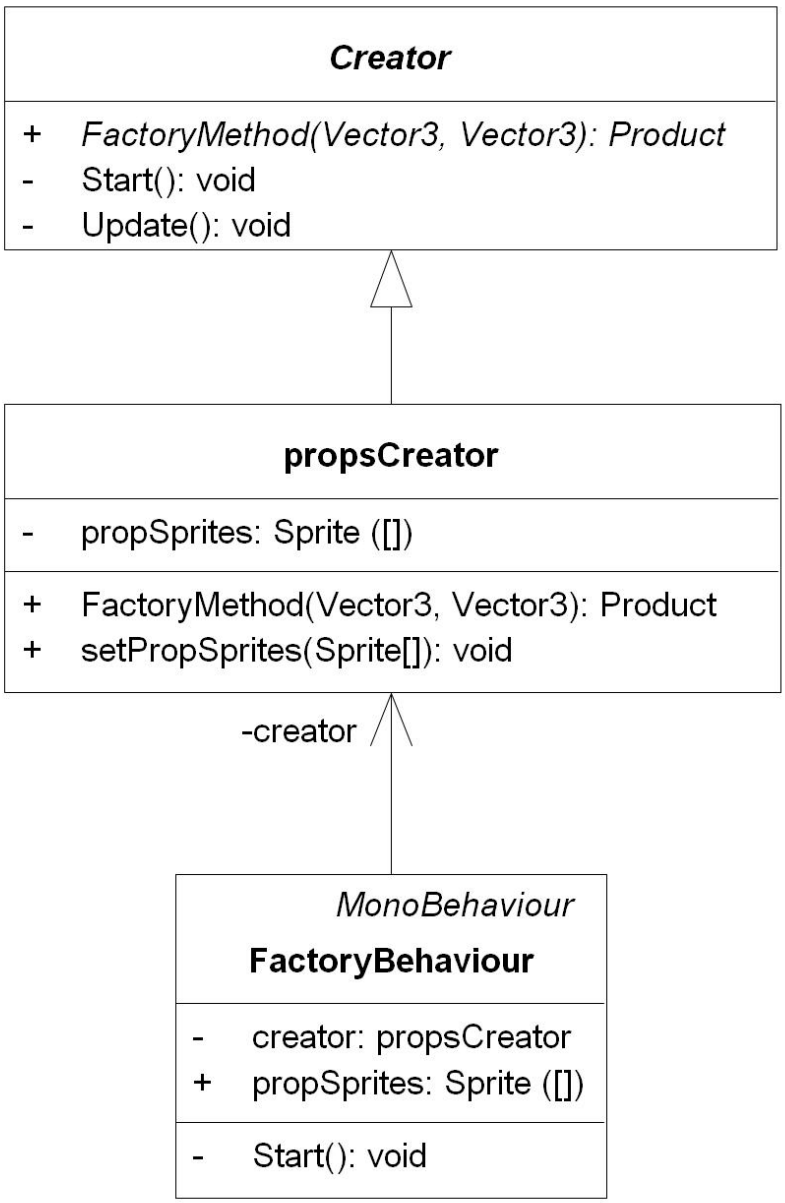
```
GameObject obj = (GameObject)Instantiate(pooledObject);
```

Factory Method

Usage:

The Factory Method pattern is used in the creation of props for our game. In our game there exists some temporary placeholders at the place where we want props(random objects that liven up your game). At the start of this game our Factory Method is used to generate all those props at the right place with one of the three predefined textures.

UML:



Snippets:

```
4 public class FactoryBehaviour : MonoBehaviour {
5     propsCreator creator;
6     public Sprite[] propSprites;
7     // Use this for initialization
8     void Start () {
9         creator = new propsCreator ();
10        GameObject[] tempProps = GameObject.FindGameObjectsWithTag("tempProp");
11        creator.setPropSprites(propSprites);
12        for (int i = 0; i < tempProps.Length; i++)
13        {
14            creator.FactoryMethod (tempProps[i].transform.position,tempProps[i].trans
15            Destroy(tempProps[i]);
16        }
17    }
18 }
```

```
4 public class propsCreator : Creator
5 {
6     Sprite[] propSprites;
7     public enum propTypes
8     {
9         Barrel,Sacks,Vase
10    }
11
12    public void setPropSprites(Sprite[] sprites)
13    {
14        propSprites = sprites;
15    }
16
17    public override Product FactoryMethod(Vector3 position, Vector3 scale)
18    {
19        GameObject gameObject = new GameObject();
20        gameObject.transform.position = position;
21        gameObject.transform.localScale = scale;
22        SpriteRenderer sr =gameObject.AddComponent<SpriteRenderer> ();
23        sr.sprite = propSprites [Random.Range (0, 3)];
24        return new propsProduct ();
25    }
26 }
```

Domain specific patterns

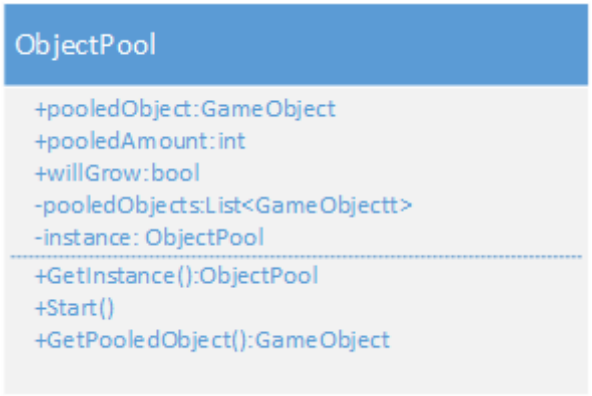
This section describes the domain specific patterns we used in the game. Some of these patterns were already highly integrated into the workflow Unity uses. In those cases we describe the logic in which Unity has implemented these patterns, rather than motivate its usage.

Objectpool

Usage:

The object pool is used for the arrows both the player and enemies fire. Instantiating objects is an expensive operation, and removal by the garbage collector can have negative impact on gameplay. Since arrows are fired frequently we avoid this (de)allocation of objects by instantiating them once and recycle them during gameplay, when they are needed.

UML:



Snippets:

There is always one object pool in the scene available(implemented as a singleton), every time an object from the pool is needed we call the getPooledObject() function:

```
GameObject arrow = ObjectPool.GetInstance.getPooledObject();
```

This function returns the first available gameobject by means of its active state. It is implemented as follows:

```
public GameObject getPooledObject()
{
    //find available object
    for (int i = 0; i < pooledObjects.Count; i++)
    {
        if (!pooledObjects[i].activeSelf)
        {
            pooledObjects[i].SetActive(true);
            return pooledObjects[i];
        }
    }
    //if no available found and growing is allowed, create a new object
    if (willGrow)
    {
        GameObject obj = (GameObject)Instantiate(pooledObject);
        pooledObjects.Add(obj);
        return obj;
    }

    //if all fails
    return null;
}
```

Update function

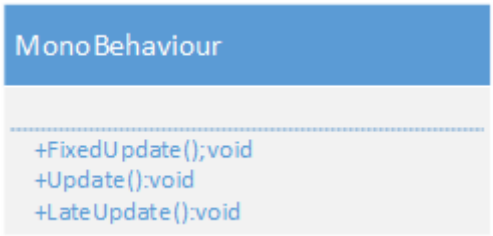
Usage:

In Unity most objects derive from MonoBehaviour. These objects implement an Update() function. For every object that is in the scene the Update() function is called by the GameLoop of Unity every frame. Basically the Update() function updates the behaviour, state etc. of each game object. For all our objects the bulk of the work is placed inside this function.

It is interesting to note that the Unity engine comes with 3 different types of this function:

- **Update:**
  Executes every frame.
- **FixedUpdate:**
  Used for physics updates, runs in a fixed speed.
- **LateUpdate:**
  Executes after all update functions are called, used for script execution ordering.

UML:



Snippets:

The Update function is rather simple:

```
// Update is called once per frame
void Update () {
    //Do work
}
```

The complexity of this pattern really depends on the work you place inside this function, rather than the pattern itself. The implementation for the Late and Fixed Update is, other than the name, identical.

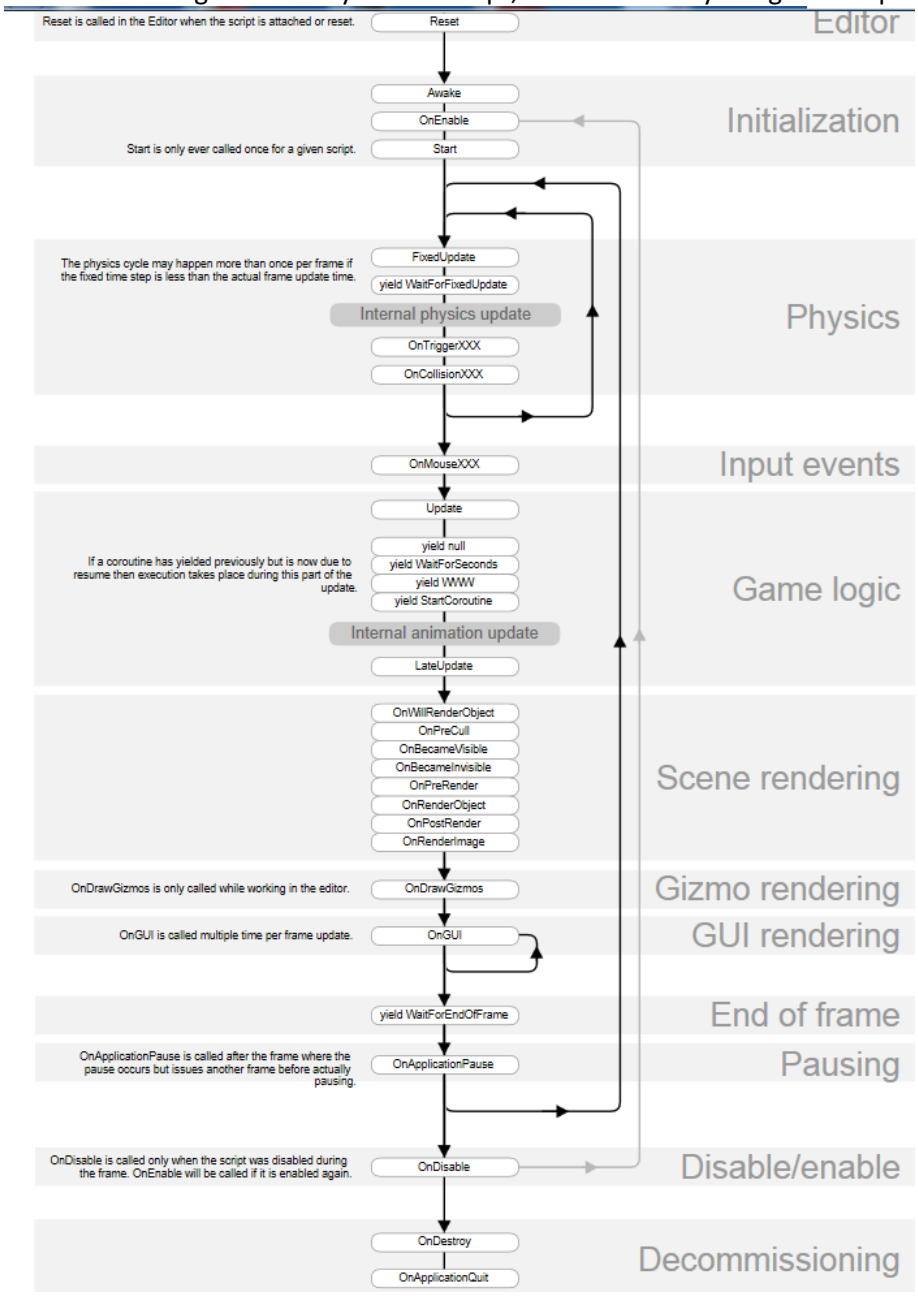
Game Loop

Usage:

This pattern is implemented by standard in Unity. This loop runs continuously during gameplay in order to process the state of game and the objects in the scene. The difference with the game loop in Unity, or any other engine, is that you don't own the loop like in a framework or library.

UML:

Instead of just showing the UML of a single function inside a class it is more illustrative to give an overview of the general lifecycle of a script, as controlled by the game loop.



Snippets:

Since the source code of the engine is in a “black box”, we don't have access to it. Therefore instead of showing a real snippet we attempted to illustrate the concept by our own code:

```
void GameLoop()
{
    //the loop generally keeps running during the lifetime of the game.
    while(true)
    {
        //this is the general idea of how the update is called for each object.
        foreach(GameObject obj in sceneObjects)
        {
            obj.Update();
        }
    }
}
```

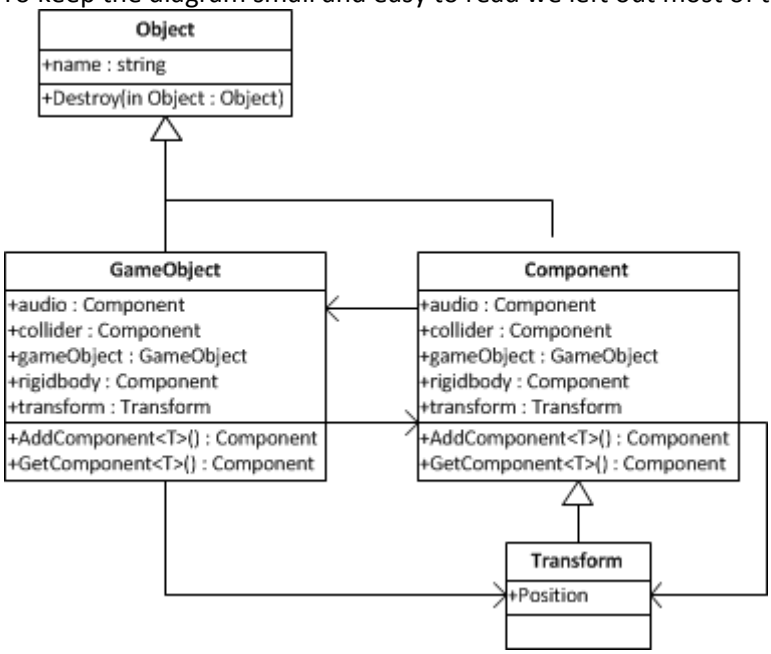
Component

Usage:

Unity works in a component based system and it resemblance the strategy pattern of the GOF. In unity when you create a game object you actually create a container for components. To add functionality or create a certain behaviour for this game object you add one or more components. A component can be a sound, rendering, script, physics, or ect. These components can also be added, removed, enabled and disabled on runtime (see snippet).

UML:

To keep the diagram small and easy to read we left out most of the other attributes and functions.



Snippets:

There are more functions to get components and it is also possible to get, add and remove components of other game objects but this is just a little example of the basics in unity.

```
// Get the component of the specified type that is attached to the gameobject
GetComponent<ComponentType>();
// Add a new component of the specified type to the gameobject
gameObject.AddComponent<ComponentType>();
// Use destroy to remove component from the gameobject (destroy is also used for removing other kind of objects)
Destroy(this);
```

Instead of using destroy to remove your component you can also use the enable function to stop a behaviour. Though when a script is disabled its public functions can still be called through other scripts that have reference to the disabled one.

```
// Disable and Enable your component
varComponentType.enabled = false;
varComponentType.enabled = true;
```