Realising and Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

D3.4 – Player-centric rule-and-pattern-based adaptation asset - INTEGRATION TUTORIAL

RAGE – WP3 – D3.4

DRAFT

Project Number	H2020-ICT-2014-1
Due Date	29 February 2016
Actual Date	January 2016
Document Author/s	Dessislava Vassileva
Version	0.3
Dissemination level	PU
Status	Draft
Document approved by	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644187

About the player-centric rule-and-pattern-based adaptation asset

The Player-centric rule-and-pattern-based adaptation asset uses metrics of player's performance, emotional status and/or playing style for realization of dynamical adaptation of various game features such as adaptation of player-driven game tasks and/or game assistance, dynamic adjustment of task difficulty, and/or adjustment of properties of audiovisual content and effects. The asset receives as input registration requests of player-centric metrics together with simple formal definitions of rules and patterns of variation of these metrics during the play time or their features such as mean, deviation and moving average within a desired time window. Next, it receives values of registered metrics and checks each incoming metric value for occurrence of a rule or a pattern defined for the metric or its feature. In case of finding such an occurrence, the asset fires a triggering event about this rule or pattern and executes its event handler, which is to be defined by the game developer depending on his/her goal to adapt specific game feature(s).

Document scope

The present document provides a concise functional description of the player-centric rule-and-pattern-based adaptation asset and, next, presents how to integrate and use in a Unity or C# based game the asset.

The document should be read together with the other tutorials about this asset, namely the Installation tutorial and the Game configuration tutorial.

Description of asset functionality

Due to the very complex and volatile nature of the player’s character, player-centric adaptation remains being a crucial issue for both entertainment and applied digital games. To be effective, player-centric adaptation should follow deterministic player models for tracking, measuring and analyzing player’s behavior.

The *Player-centric rule-and-pattern-based adaptation asset* uses metrics of player's performance, emotions and/or playing style for realization of dynamical adaptation of various game features such as generated content at micro and/or macro level [1], or task difficulty. More precisely, the player-centric metric may indicate three important issues:

1. player performance, which embraces knowledge and intellectual abilities of the player's including synthetic, analytical and practical skills [2], e.g. abilities for comprehension, memorization, evaluation, reasoning, decision making, and so on;
2. player affective (emotional) status, which represents emotional experiences, feelings and motivation of the player [3]. The affective status may be inferred by measuring facial, gestural, vocal, neural and/or psychophysiological bodily reactions [4];
3. playing style, which depends on player's personality and ways of thinking and reacting to game challenges [5].

The asset receives as input registration requests of player-centric metrics about individual performance, emotional status and/or playing style, together with simple formal definitions (using a simple but yet powerful syntax) of rules and patterns of variation of these metrics during the play time or their features such as mean, deviation and moving average within a desired time window. The idea of using patterns and rules for adaptation purposes is adopted from the game adaptation control framework using implicit derivation of the player character during the game play developed within the scope of the ADAPTIMES research project [12]. Next, it receives values of registered metrics and checks each incoming metric value for occurrence of a rule or a pattern defined for the metric or its feature. In case of finding such an occurrence, the asset fires a triggering event about this rule or pattern and executes its event handler, which is to be defined by the game developer depending on his/her goal to adapt specific game feature(s). Thus, the asset does not depend on any concrete digital game and provides the game developer with freedom to program any control over game adaptation. Various game features can be dynamically adapted, which fall into three main groups regarding the three main component of the Mechanics-Dynamics-Aesthetics (MDA) framework [13], namely game mechanics, dynamics and aesthetics as follows:

- adaptation of player-driven game tasks and/or game assistance such as helping instructions [6]. As well, the managed appearance of tasks and assistance in their performing during the game flow can be adjusted [7];
- dynamic adjustment of task difficulty – like in [8] where the adaptation control is based on the player's anxiety, or in [4] where adaptation is controlled according the skill level of the player;
- adjustment of properties of audiovisual content and effects, such as ambient light in rooms in a video game [9].

The asset works only on the client side. It can be used without any other asset provided the game developer is able to provide the player-centric metric(s) used as a basis for game adaptation. Otherwise, the developer should use another asset(s) for provisioning of the metric(s) values, such as the Real-time Emotion Detection Asset or the RT Arousal Detection Using Galvanic Skin Response Asset.

For understanding the installation process of the asset, refer to the Installation tutorial. For realizing how the asset is to be used for game adaptation together with a video game based on the Unity 3D game engine [14], see the Game integration tutorial.

Asset integration steps

The player-centric rule-and-pattern-based adaptation asset is implemented in C# and .NET framework, version 3.5. It is packaged in both variants: as an archive file containing necessary dll files for integration and as an archive files containing the source code. It can be integrated in a Unity project or other C# or .NET based game project as using only dll files. The main classes implementing the asset are:

- *PlayCentricStatisticExtractor* – it is an abstract class defining interfaces for communication with the asset. The code map diagram of this class is shown on fig. 1. The metod `public abstract Object PatternEventHandler(Object patternInput, Object gameObject)` is abstract and it should be overwritten for changing game features when the pattern with a particular **pattern** name has found. This class contains methods related to metrics, patterns and time window.

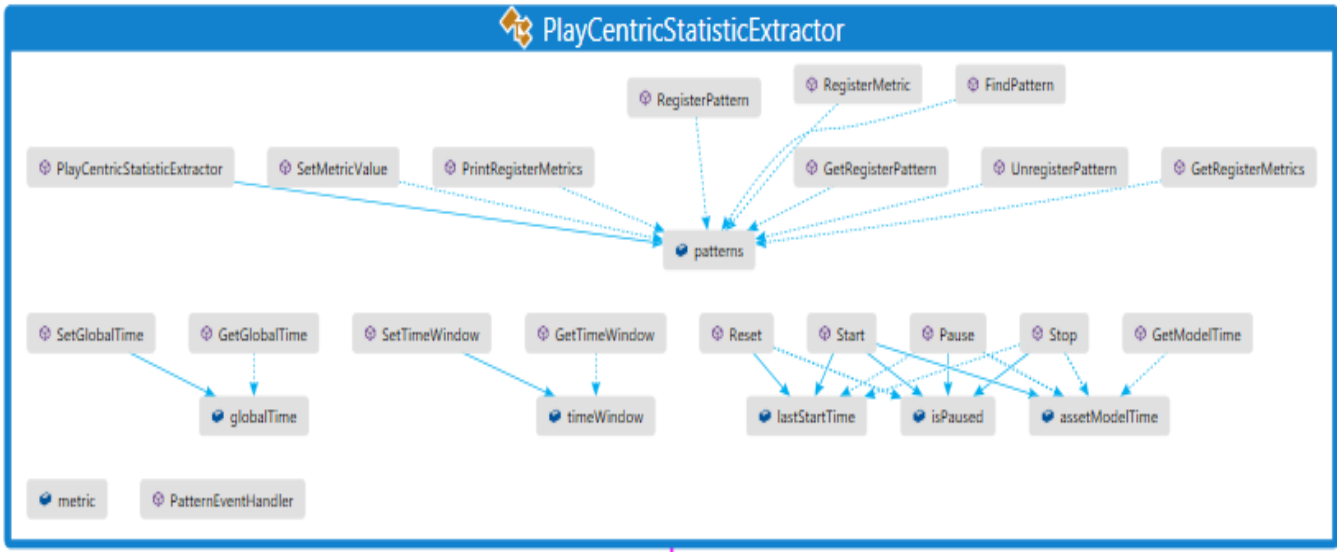

Fig. 1: The code map of PlayCentricStatisticExtractor.

- `PlayerCentricRulePattern` – this is the class responsible for implementing all functionality related to metrics and patterns. It includes methods for registered/unregistered of metric and patterns, for setting metrics value, for discovering patterns, etc. Its code map diagram is shown in fig. 2.

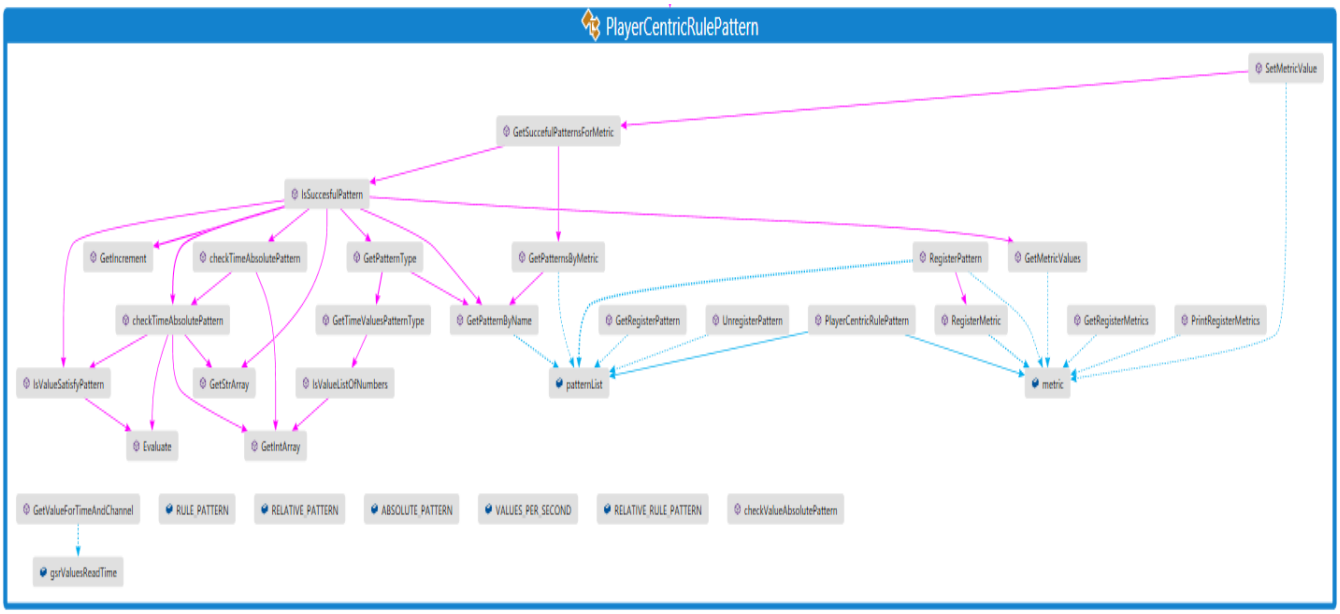

Fig. 2: The code map of PlayerCentricRulePattern.

- `RuleBasedAdaptation` and `RuleBasedAdaptationAssetSettings` – these classes are helping and they are responsible for setting/getting asset configuration and for defining some test cases. The code map and relationship between these two classes is shown in fig.3.

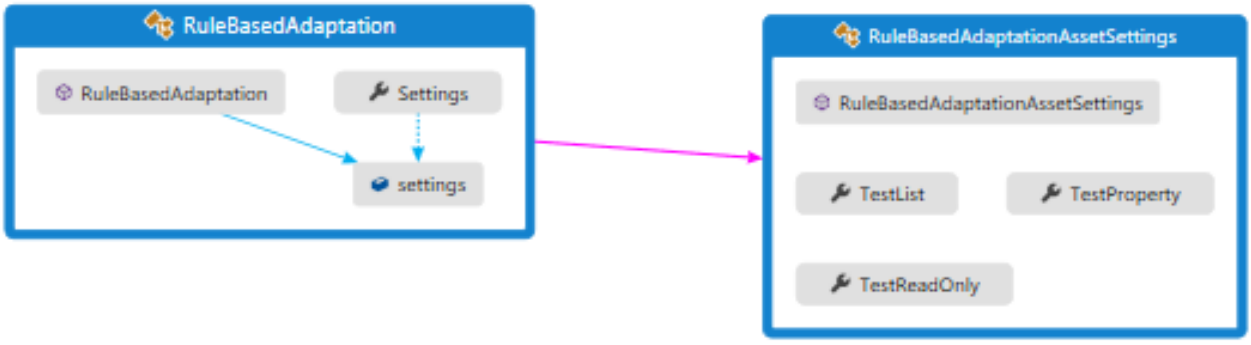

Fig. 3: The code maps and relationship between classes RuleBasedAdaptation and RuleBasedAdaptationAssetSettings.

The asset integration process includes following steps:

1. Download the zip file “PlayerCentricRuleBasedAdaptationDlls.zip” with dll files implementing the asset functionality;

2. Copy the asset dll archive file in a folder in your Unity project and uncompressed it there (fig. 4);

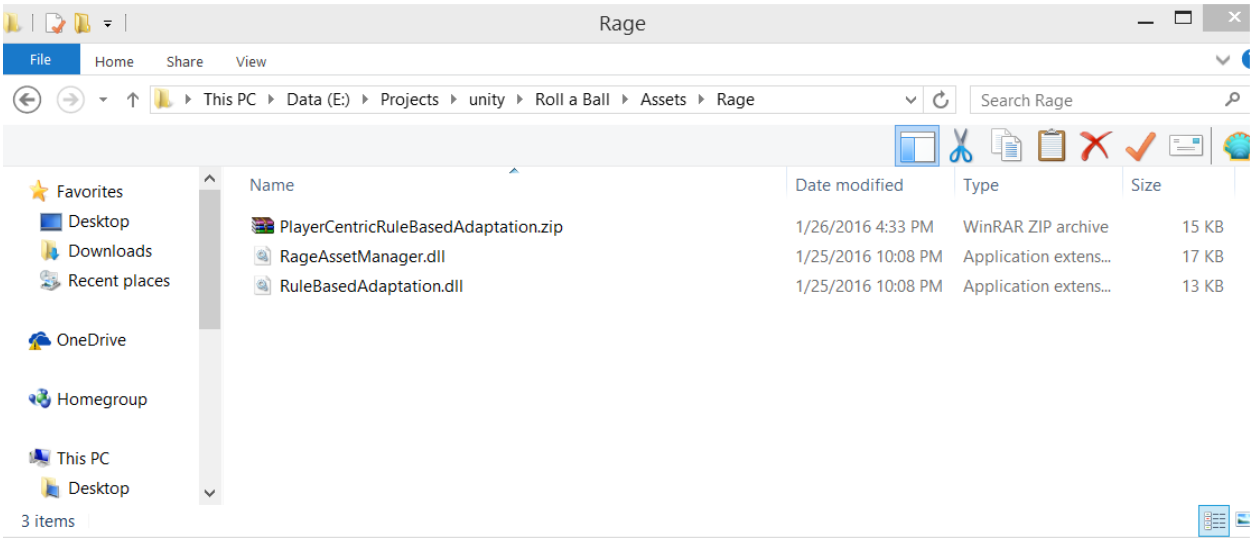

Fig. 4: Copy and uncompressed the archive file.

3. Import dll files to your Unity project (fig. 5 and fig. 6);

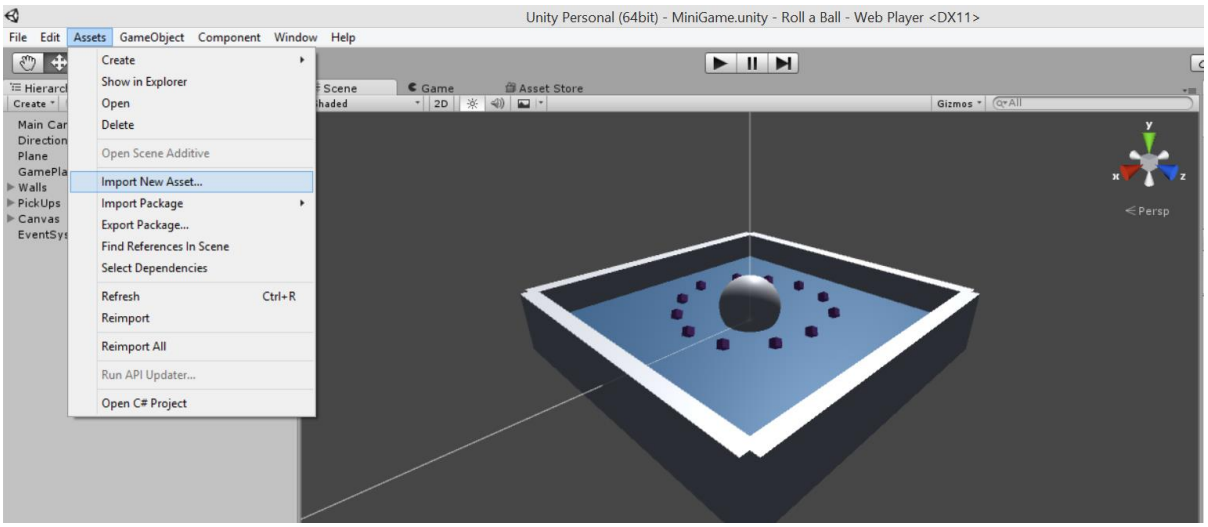

Fig.5: Import new asset in Unity 3D.

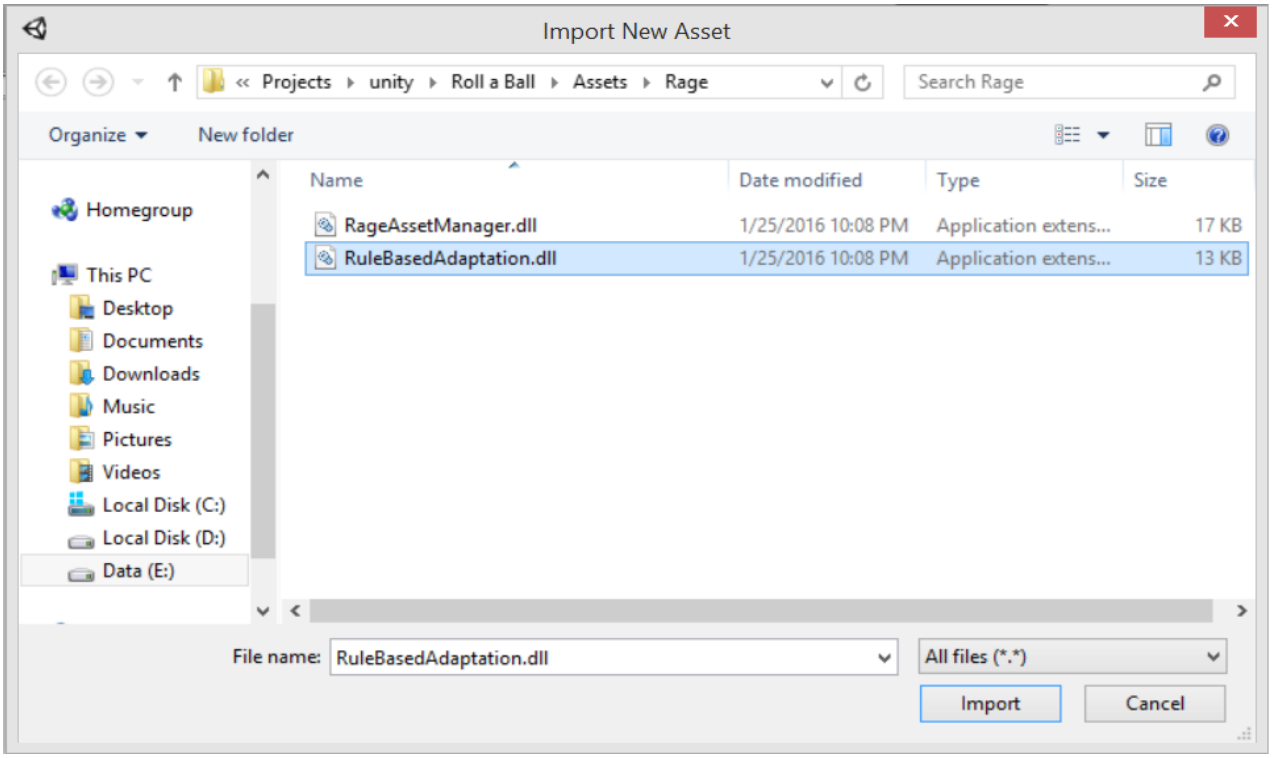

Fig. 6: Import dll files to the Unity project.

4. Add dll files to the references of your project in Visual Studio or other IDE that you use for your C# code (fig. 7);

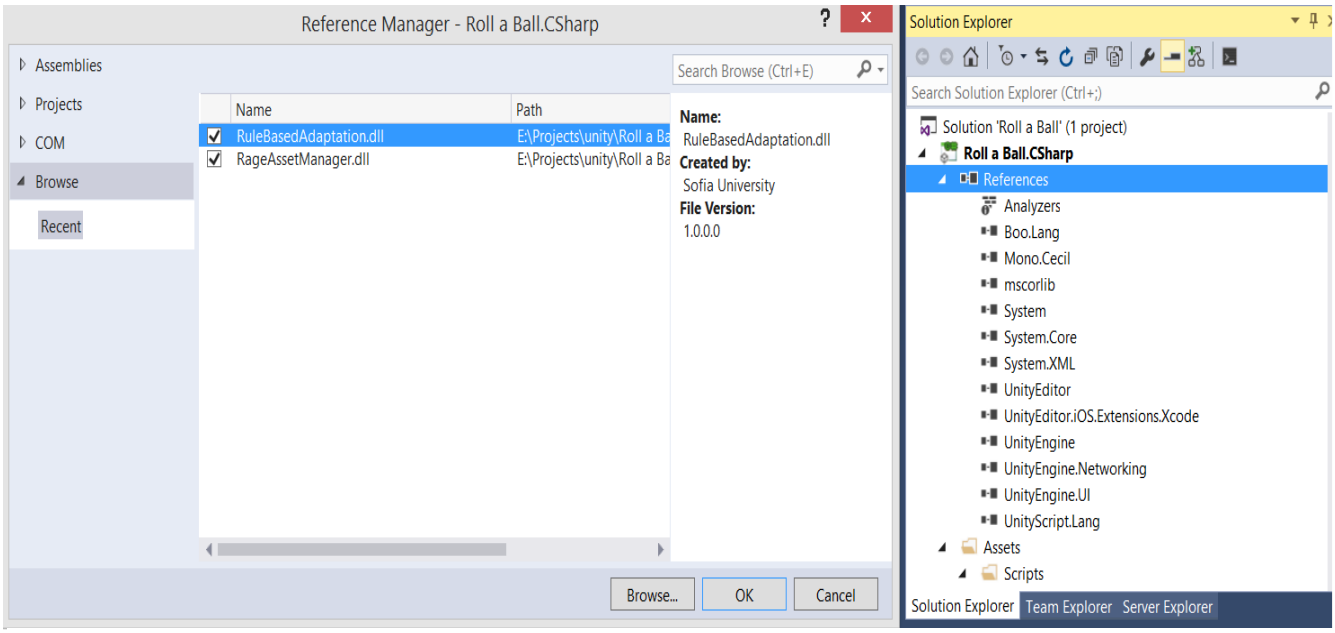

Fig. 7: Add dll files to the references of the Visual Studio project.

5. Add to your game project in Visual Studio a class extending the abstract class `Assets.Rage.PlayerCentricRulePatternBasedAdaptationAsset.PlayCentricStatisticExtractor` (fig. 8);

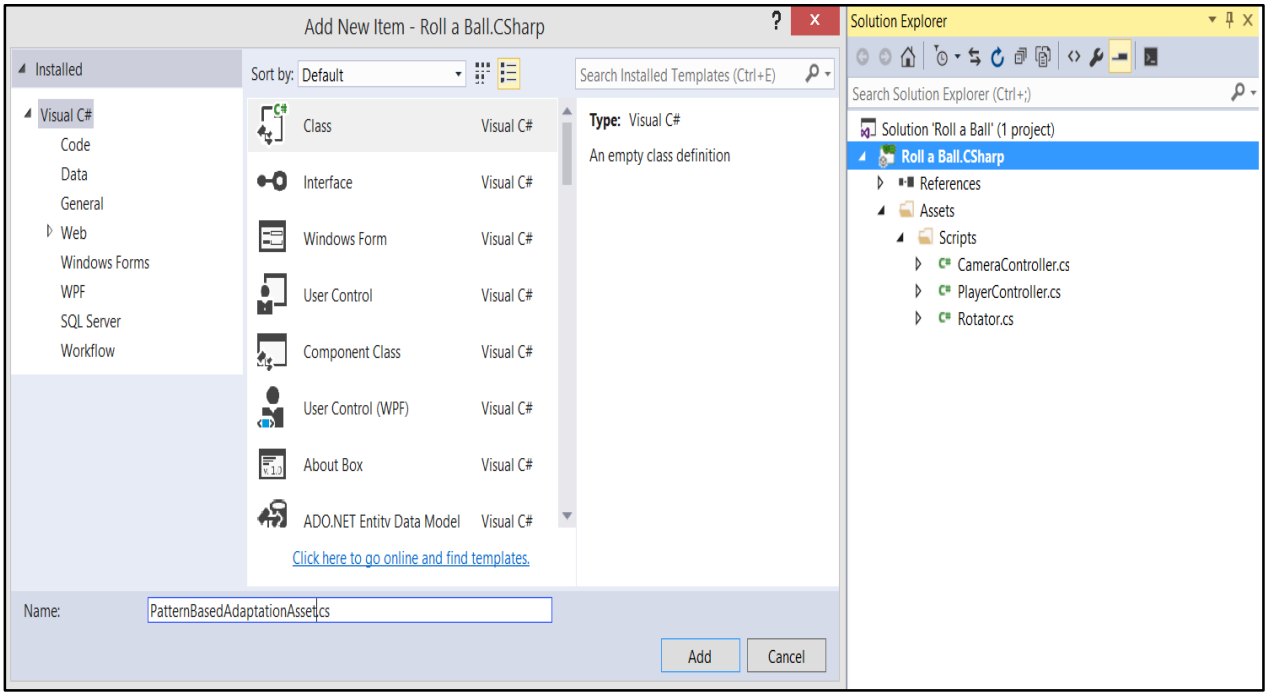

Fig. 8: Add to your game project in Visual Studio a class (in this example its name is PatternBasedAdaptationAsset) extending the abstract class.

6. Implement the method `public abstract Object PatternEventHandler(Object patternInput, Object gameObject).` This method has two parameters. The first of them patterInput contains the list of founded patterns, and the second parameter – gameObject – presents the game object, where the asset is used. The method should defined what action will trigger on the game object depending on founded patterns. An example of an implementation is given below.

```
using System;

namespace Assets.Rage.PlayerCentricRulePatternBasedAdaptationAsset

{

    class PatternBasedAdaptationAsset :
```

```
Assets.Rage.PlayerCentricRulePatternBasedAdaptationAsset.PlayCentricStatisticExtractor

    {

        public override Object PatternEventHandler(Object patternInput, Object
gameObject)

        {

            PatternAction((System.Collections.Generic.List<string>)patternInput,
(UnityEngine.GameObject) gameObject);

            return null;

        }

        public void PatternAction(System.Collections.Generic.List<string> patterns,
UnityEngine.GameObject gameObject)

        {

            if (patterns.Contains("Change color in blue"))

            {

                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.blue;

            }

            else if (patterns.Contains("Change color in green"))

            {

                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.green;

            }

            else if (patterns.Contains("Change color in green2"))

            {

                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.green;

            }
```

```
            else if (patterns.Contains("Change color in yellow"))
            {
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.yellow;
            }
            else if (patterns.Contains("Change color in white"))
            {
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.white;
            }
            else if (patterns.Contains("Change color in red"))
            {
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.red;
            }
            else if (patterns.Contains("Change color in cyan"))
            {
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.cyan;
            }
            else if (patterns.Contains("Change color in black"))
            {
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.black;
            }
            else if (patterns.Contains("Change color in magenta"))
            {
```

```
                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.magenta;

            }

            else if (patterns.Contains("Change color in blue2"))

            {

                gameObject.GetComponent<UnityEngine.Renderer>().material.color =
UnityEngine.Color.blue;

            }

            if(patterns.Contains("Change size"))

            {

                UnityEngine.Vector3 currentScale = gameObject.transform.localScale;

                gameObject.transform.localScale = currentScale - currentScale * 0.15f;

            }

        }

    }

}
```

7. It should be create an instance of the class extending *PlayCentricStatisticExtractor* (in our example from point 5. and 6. this is *PatternBasedAdaptationAsset*) in the class related to the game object that will use the asset. In the method `void Start()` has to be defined and registered all metrics (with the method `public bool RegisterMetric(String metricName)`) and patterns (with the method `public bool RegisterPattern(String patternName, String metricName, String featureName, String timeInterval, String values)`). When a metric value is changed, the new value has to be set for the correspond metric with the method `public List<String> SetMetricValue(String metricName, int value)`. This method returns the list with all patterns founded depending on the new metric value. At the end the event handler (`public void PatternAction (List<string> patterns, UnityEngine.GameObject gameObject)`) will be executed upon the list of rule/pattern name, acting as adaptation method changing some game features about game mechanics, dynamics and aesthetics [10]. An example of using *PatternBasedAdaptationAsset* is given below.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.SceneManagement;
using Assets.Rage.PlayerCentricRulePatternBasedAdaptationAsset;
using System.Collections.Generic;


public class PlayerController : MonoBehaviour
{


    public float speed;
    public Text countText;
    public Text winText;
    public Text sizeText;
    public Text timerText;
    private Rigidbody rb;
    private int count;
    private float time;
    protected bool paused;


    PatternBasedAdaptationAsset testMetricPattern;


    void Start()
    {
        rb = GetComponent<Rigidbody>();
```

```
        count = 0;

        SetCountText();

        SetSizeText();

        winText.text = "";

        timerText.text = "";

        paused = false;


        testMetricPattern = new PatternBasedAdaptationAsset();


        testMetricPattern.RegisterMetric("Number of hits");


        testMetricPattern.RegisterPattern("Change color in blue", "Number of hits",
"none", "GT(1)", "2");

        testMetricPattern.RegisterPattern("Change color in blue2", "Number of hits",
"none", "GT(1)", "10");

        testMetricPattern.RegisterPattern("Change color in yellow", "Number of hits",
"none", "GT(1)", "3");

        testMetricPattern.RegisterPattern("Change color in green", "Number of hits",
"none", "GT(1)", "4");

        testMetricPattern.RegisterPattern("Change color in green2", "Number of hits",
"none", "GT(1)", "11");

        testMetricPattern.RegisterPattern("Change color in white", "Number of hits",
"none", "GT(1)", "5");

        testMetricPattern.RegisterPattern("Change color in red", "Number of hits",
"none", "GT(1)", "6");

        testMetricPattern.RegisterPattern("Change color in cyan", "Number of hits",
"none", "GT(1)", "7");

        testMetricPattern.RegisterPattern("Change color in black", "Number of hits",
"none", "GT(1)", "8");
```

```
        testMetricPattern.RegisterPattern("Change color in magenta", "Number of hits",
"none", "GT(1)", "9");


        testMetricPattern.RegisterPattern("Change size", "Number of hits", "none",
"GT(1)", "GT(0)");
    }


    void Update()
    {
        if(!paused)
        {
            time += Time.deltaTime;
            //update the label value
            timerText.text = string.Format("Time: {0} ", time);
        }
    }


    void FixedUpdate()
    {
        if(!paused) {
            float moveHorizontal = Input.GetAxis("Horizontal");
            float moveVertical = Input.GetAxis("Vertical");


            Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);


            rb.AddForce(movement * speed);
```

```
        }
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("PickUp"))
        {
            other.gameObject.SetActive(false);
            count = count + 1;
            SetCountText();

            List<string> patterns = testMetricPattern.SetMetricValue("Number of hits",
count);
            testMetricPattern.PatternEventHandler(patterns, gameObject);
            SetSizeText();
        }
    }

    void SetCountText()
    {
        countText.text = "Cubes hit: " + count.ToString();
        if (count >= 12)
        {
            winText.text = "You won!";
            paused = true;
```

```
        }
    }

    void SetSizeText()
    {
        sizeText.text = "Ball size: " + gameObject.transform.localScale.x.ToString();
    }

    void OnGUI()
    {
        if (GUI.Button(new Rect(10, 90, 70, 30), "Replay"))
        {
            SceneManager.LoadScene("MiniGame");
        }

        if (GUI.Button(new Rect(10, 130, 70, 30), "Stop"))
        {
            Object[] objects = FindObjectsOfType(typeof(GameObject));
            foreach (GameObject go in objects)
            {
                go.SendMessage("OnPauseGame", SendMessageOptions.DontRequireReceiver);
            }
        }

        if (GUI.Button(new Rect(10, 170, 70, 30), "Resume"))
```

```
        {
            Object[] objects = FindObjectsOfType(typeof(GameObject));
            foreach (GameObject go in objects)
            {
                go.SendMessage("OnResumeGame", SendMessageOptions.DontRequireReceiver);
            }
        }
    }
    void OnPauseGame()
    {
        paused = true;
    }

    void OnResumeGame()
    {
        paused = false;
    }
}
```

Properties of patterns for player-centric rule-and-pattern-based adaptation asset are name of pattern, name of metric, name of feature, set of time values time and set of metric values. The size of the set of time values has to be equal to the size of the set of metric values. A pattern matched A metric if the pattern for each one value from the set of time values has corresponding metric value from the set of metric value. The player-centric rule-and-pattern-based adaptation asset supports three type of values for time and metric:

- Absolute – this type of value defines a set of integer numbers. A pattern example with absolute feature value using absolute time moments is given below:

```
{ name=”Happy pattern”, metric=”happiness”, feature=”moving average”, time=”120000 300000 600000”, values=”1 2 3” }
```

- Relative - this type of value defines a set of integer numbers after time t or greater than an integer value x. A pattern example with absolute feature value using relative time moments after time t is given below:

```
{ name=”GSR mean pattern”, metric=”GSR”, feature=”average”, time=”t t+3000 t+6000 t+9000”, values=”16 20 24 20” }
```

- Rule based – this type of value defines a set of integer numbers that matched a boolean expression. A pattern example for checking if the metric about quiz result is between 20 and 30 points between the third minute (after 180000 ms) and before the eighth minute (before 480000 ms) is given below:

```
{ name=”A quiz points rule”, metric=”Quiz result”, feature=”none”, time=”GT(180000) AND LT(480000)”, values=”GT(20) AND LT(30)” }
```

The syntax of pattern definitions is described in details in the configuration tutorial.

8. The asset timer can be set/reset at any time (the time is given in milliseconds) in order to synchronize it to the game engine by using the *public void SetGlobalTime(int synchronizationTime)* method, as far as setting the moving average time window by using the *public void SetTimeWindow(int milliseconds)* method (as explained below).

Example:

```
PlayCentricStatisticExtractor test = new PlayCentricStatisticExtractor();

//the global time will be 2 minutes

test.SetGlobalTime (120000);

//the time window will be equal to 40 seconds

test.SetTimeWindow (40000);
```

Examples of use in games

The player-centric adaptation technology continues being very promising and attractive for both entertainment and applied video games. State current examples if it is already in use. Various event handlers can be developed for game adaptation based on changes in player’s character, namely:

- Player's performance – including knowledge, motivation, skills and abilities
- Player's affective state - emotions and arousal;
- Player's style – such as killer/achiever/explorer/socializer of Bartle [11] or conqueror/manager/wanderer/participant styles of Bateman and Boon [5]

The player character is to be implicitly derived during the play process, instead of explicit ways like using self-reports. The asset can be used for adapting various features regarding game mechanics, scenarios, dynamics, or aesthetics, as programmed in the event handlers by the game developer.

Value

It is not possible to predict all possible types of features to be adapted even for a single complex game. Therefore, for not losing generality and for preserving the blackbox communication (the asset has no information about the game logic, scenarios, dynamic variables, etc.), the game developer is free to overwrite the default event handler as he likes. Thus, the asset offers:

- Independence from game logic, dynamics, and scenarios – includes independence from the adaptivity scale (micro/macro adaptivity [1]);
- Independence from game engine used;
- Easy use and integration – no specific domain expertise is required to use the asset, however, the game developer should be able to define reasonable and consistent patterns and rules of metric variation. Thus, the asset can be used by trainers, pedagogues and psychologist for creating various adaptive applied games;
- Flexibility - patterns and rules of metric variation can be set and cancel for monitoring during run time;
- Monitoring of heterogeneous player-centric metric – asset user is able to specify various metric about player's performance, emotional status and playing style. For example, for the playing style can be observed at given together several metric and specific playing style can be inferred based on their change.

Dependencies

The asset can be used in isolations provided the player-centric metrics are known by the developer at run time. Otherwise, the game developers may use it together the Real-time Emotion Detection Asset or

the RT Arousal Detection Using Galvanic Skin Response Asset, in order to provide emotion or arousal metric, respectively.

Technical details

The asset has only a client component (in form of dll - RuleBasedAdaptation.dll). Its run-time requirements are as follows:

- Windows Vista or higher
- Microsoft .NET Framework 3.5

Source project requirements:

- Microsoft Visual Studio Professional to import the solution (project).
- Support for C#.

Licensing

No specific licensing is required.

References

1. Kickmeier-Rust, M. D., Albert, D. Educationally adaptive: Balancing serious games. Int. Journal of Computer Science in Sport, 2012, 11(1), pp.1-10.
2. Tremblay, J., Bouchard, B., Bouzouane, A. Adaptive Game Mechanics for Learning Purposes-Making Serious Games Playable and Fun, Proc. of CSEDU (2), 2010.
3. Tijs, T., Brokken, D., IJsselsteijn, W. Creating an emotionally adaptive game. ICEC 2008, LNCS 5309, Springer Berlin Heidelberg. pp.122–133.
4. Fairclough, S., Gilleade, K. Construction of the biocybernetic loop: a case study. Proc. of the 14th ACM Int. Conf. on Multimodal interaction, ACM, 2012, October, pp. 571-578.
5. Bateman, C., Boon, R. 21st Century Game Design, vol. 1. Charles River Media, Londan, 2005.
6. Murphy, C., Chertoff, D., Guerrero, M., Moffitt, K. Design Better Games: Flow, Motivation, and Fun. Design and Development of Training Games: Practical Guidelines from a Multidisciplinary Perspective, 2014, p.1773.

7. Sweetser, P., Johnson, D. M., Wyeth, P. Revisiting the GameFlow model with detailed heuristics. Journal of Creative Technologies, 2012 (3).
8. Rani P., Sarkar N., Liu C. Maintaining optimal challenge in computer games through real-time physiological feedback. Proc. of the 11th Int. Conf. on Human Computer Interaction, 2005, pp.184–192.
9. Grigore, O., Gavat, I., Cotescu, M., Grigore, C. Stochastic algorithms for adaptive lighting control using psycho-physiological features. Int. J. of Biology and Biomedical Engineering 2, 2008, pp.9–18.
10. Parnandi, A., Gutierrez-Osuna, R. A comparative study of game mechanics and control laws for an adaptive physiological game. J. on Multimodal User Interfaces, 2014, pp.1-12.
11. Bartle, R. Hearts, Clubs, Diamonds, Spades: Players Who suit MUDs, 1996, http://mud.co.uk/richard/hcds.htm
12. Bontchev, B. Methods of adaptation control by implicit derivation of the player character during the game play, Deliverable D3, Version 1.0, ADAPTIMES – WP3 – D3
13. Hunicke, R., LeBlanc, M., Zubek, R. MDA: A Formal Approach to Game Design and Game Research, 2005, CiteSeerX: 10.1.1.79.4561
14. Smith, M., Queiroz, C. Unity 4.x Cookbook, Packt Publishing, 2013