Tutorial for the Adaptation and Assessment (TwoA) asset – C#

Asset version: 1.2.5
Document version: 1.0
Document date: 2018.01.09
Document author: Enkhbold Nyamsuren (Enkhbold.Nyamsuren@ou.nl; E.Nyamsuren@gmail.com)

Table of Contents

1 Introduction 2
2 Setting up the development environment. 4
2.1 Installing a Visual Studio Community. 4
2.2 Creating a root folder. 4
2.3 Downloading the RAGE client-side asset architecture. 5
2.4 Downloading the TwoA asset. 5
3 Creating a Hello World 6
3.1 Creating a new project. 6
3.2 Importing and referencing the RAGE architecture project. 6
3.3 Importing and referencing the TwoA asset. 8
4 Using TwoA with both response time and Boolean accuracy 11
4.1 Entering player data. 11
4.2 Entering game scenario data 12
4.3 Using TwoA's adaptation 13
4.4 Using TwoA's assessment 14
5 Using TwoA with continuous accuracy 19
6 Learning with TwoA 21
6.1 Designing game scenarios 21
6.2 Learning iteration. 22
6.3 Interpreting difficulty ratings 22
6.4 Interpreting a skill rating 23

1 Introduction

The Adaptation and Assessment (TwoA) asset provides two main functionalities:

- assessment of the player's skill and game difficulty
- adaptation of the game difficulty to player's skill

Both assessment and adaptation are done in real time, automatically, and stealthy without interrupting the gameplay. The assessment of the player's skill can be used to track the player's learning progress and for the adaptation of difficulty. The difficulty adaptation is pedagogically beneficial by keeping the player motivated and providing smoother learning experience.

The TwoA asset works in a manner similar to matchmaking algorithms such as Elo and TrueSkill. However, instead of matching two human players, the TwoA asset tries to match a human player to a game content with difficulty level that matches the player's skill level. Both player skill and game difficulty are represented with numerical ratings.

***Assumed game structure*:** It is assumed that a game consists of one or more individually playable scenarios. Scenarios can be played sequentially without any strict order, and the player can play one scenario at the time. The same scenario can be replayed by the same player one or more times. It is further assumed that scenarios vary in difficulty and demand different levels of skills from the player. Finally, it is assumed that a scenario has a learning content.

***Difficulty rating*:** Each scenario will have a quantitative difficulty rating in the TwoA asset. A higher rating will indicate higher difficulty that may demand higher skills from a player.

Skill rating: A player will also have a quantitative skill rating in the TwoA asset. A higher rating of a player will indicate higher skill. Player's skill rating can be updated after each scenario played by the player. If the player underperforms in a scenario then player's rating will decrease and increase otherwise. Player's skill rating will be estimated based on player's performance metrics. If the option for dynamic ratings for scenarios is enabled then difficulty ratings of a scenario will be recalculated after each playthrough. If the player underperforms in a scenario then scenario's rating will be increased. Scenario's rating will be decreased if the player performs better than expected.

1. A new player will be assigned a rating close to 0. The TwoA asset will automatically re-estimate this skill rating after each scenario played by the player.
2. Assigning quantitative difficulty ratings to scenarios can be done by one of following ways:
    2.1. Assign initial random ratings (e.g. 0) to scenarios and let the TwoA asset to adaptively estimate real difficulty ratings during gameplays with real players.
    2.2. Assign initial random ratings to scenarios and let the TwoA asset to adaptively estimate real difficulty ratings during a pre-test experiment with the human participant.
    2.3. Ratings are assigned by an experienced teacher (expert).
3. Setting difficulty ratings for scenarios to be either fixed or dynamic:
    3.1. Once set for scenarios, fixed ratings will not change (will not be re-estimated by the TwoA asset). This option is helpful if difficulty ratings of scenarios are well known and not expected to change.
    3.2. If ratings are dynamic then scenario's rating will be re-estimated after playthrough based on player's performance metrics. This option is helpful if scenario's difficulty is not known clearly. The TwoA asset will be able to estimate scenario's true difficulty rating after several playthroughs.

Player's performance metrics:
Performance metrics indicate how well a player performed within a game scenario. The TwoA asset can accept one of two following performance metrics:

- A combination of response time and accuracy
- Accuracy only

A combination of response time and accuracy: Response time is an amount of time a player required to finish the scenario. The value of accuracy is 1 if the player was able to finish a scenario successfully and 0 otherwise. There are can be various ways to measure the accuracy: (1) overall outcome of all activities within a scenario are evaluated as 0 or 1; (2) not being able to finish the game within certain time threshold; (3) making more than a certain number of errors; (4) making more than a certain number of suboptimal decisions; etc.

Accuracy only: Accuracy can be any value between 0 and 1. For example, a value of 1 indicates a complete victory while 0 indicates a complete loss. A value of 0.5 can be used to indicate a draw.

Use cases:

1. The TwoA asset will track learning rate of students using a non-intrusive assessment based on player's performance metrics (for details see **Appendix 1**).
2. The TwoA asset can be used to identify possible learning content gaps in scenarios (for details see **Appendix 2**).
3. The TwoA asset will recommend a next scenario most suitable to the current skill level of the player.

Past uses of the assessment and adaptation algorithm in TwoA**:**

TwoA asset is content agnostic meaning that it will NOT require any domain knowledge modeling effort. Therefore, it is ideal for assessing complex skills that are hard to express explicitly. It also makes the TwoA asset easy to reconfigure if new scenarios are added to the game. For examples of practical use of the assessment and adaptation algorithm used in the TwoA, you can refer to:

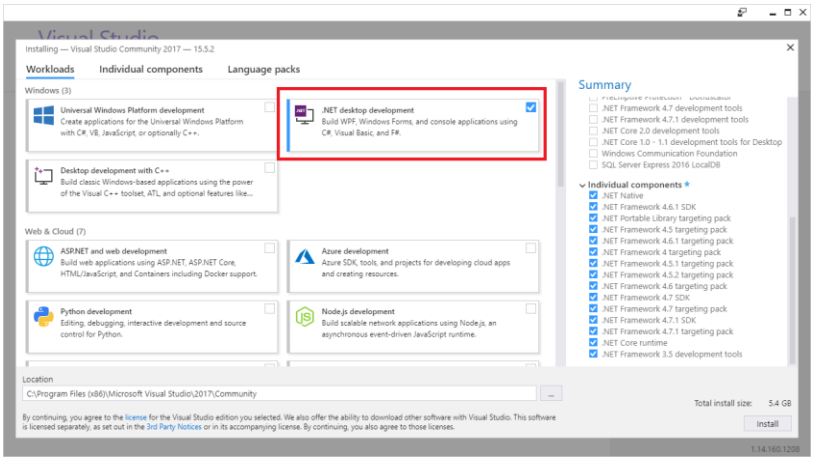
- http://www.taalzee.nl/ (In Dutch) - Serious games for language skill practice; Taalzee provides detailed information about the strengths and weaknesses of players and their development compared to peers
- http://www.statistiekfabriek.com/ (In Dutch) - Statistics Factory provides a fun environment for practicing statistics at their own level. The program contains more than 2000 statements about probability theory, descriptive and inferential statistics.
- http://www.rekentuin.nl/ (In Dutch) (http://www.mathsgarden.com/ in English) - Serious games for training various skills including mathematical reasoning, spatial reasoning, working memory capacity, logical reasoning.

[references to the benefits of balancing difficulty]
[examples based on the Mathgarden and Qwirqle]

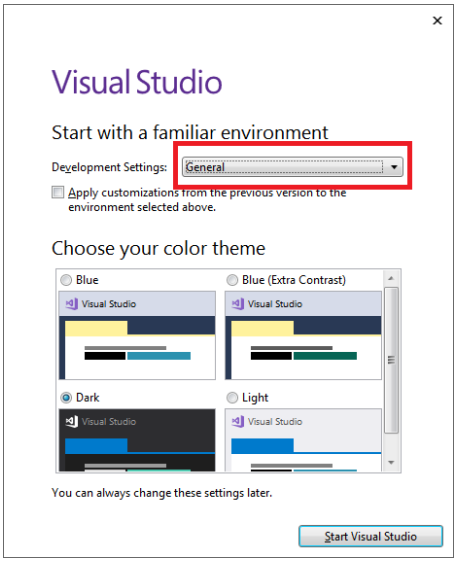
2 Setting up the development environment.

2.1 Installing a Visual Studio Community.

Since the TwoA asset is written in C#, Visual Studio is the most suitable development environment for it. Microsoft provides a free version of the Visual Studio named a Community edition. It can be downloaded via following link: https://www.visualstudio.com/vs/community/ . During the installation, it may ask you to select a workload as shown in the image below. The option "*.NET desktop development*" should be selected. This tutorial uses the Visual Studio Community 2017. If you are using a latter or an earlier version of the Visual Studio then there might be slight differences in the user interfaces.



During the first start of the Visual Studio, a prompt window may appear requesting to select the development settings (image below). You can select "*General*" settings so that your UI is consistent as much as possible with this tutorial. You can also pick your preferred color theme. This tutorial uses the Dark therem.



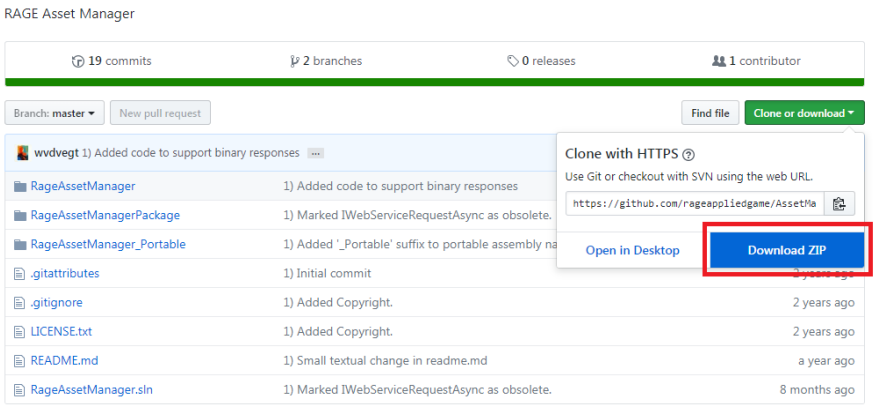
2.2 Creating a root folder.

Create a folder called "*tutorial*". It does not matter where the folder is created. It can be a place that is most convenient for you.

2.3 Downloading the RAGE client-side asset architecture.

The latest version of the RAGE client-side asset architecture (RAGE architecture for short) can be downloaded via https://github.com/rageappliedgame/AssetManager .

3.1. Follow the above link to open the Github repository.
3.2. Click the "*Clone or download*" button in the top-right corner.
3.3. Select the option "*Download ZIP*" and download the zip file to the "*tutorial*" folder. The downloaded zip file should be named something like "*AssetManager-master.zip*".
3.4. Extract the content of the zip file.



We are mainly interested in the content of the "*RageAssetManager*" subfolder. Its path should resemble: "*tutorial\AssetManager-master\RageAssetManager*". The "*RageAssetManager*" folder contains a raw source code for the RAGE architecture. The source code is packaged as a Visual Studio project and can be easily integrated to another project.

2.4 Downloading the TwoA asset.

The TwoA asset can be downloaded via following link: https://github.com/rageappliedgame/HatAsset . Follow the same steps as you did with the RAGE architecture to download and extract the content from the zip file. Once, you are done the "*tutorial*" folder should contain a folder "*HatAsset-1.2-noXML*" (the name of the subfolder may be different depending on the version of the TwoA asset). The "*HatAsset-1.2-noXML*" folder should contain multiple subfolders explained below:

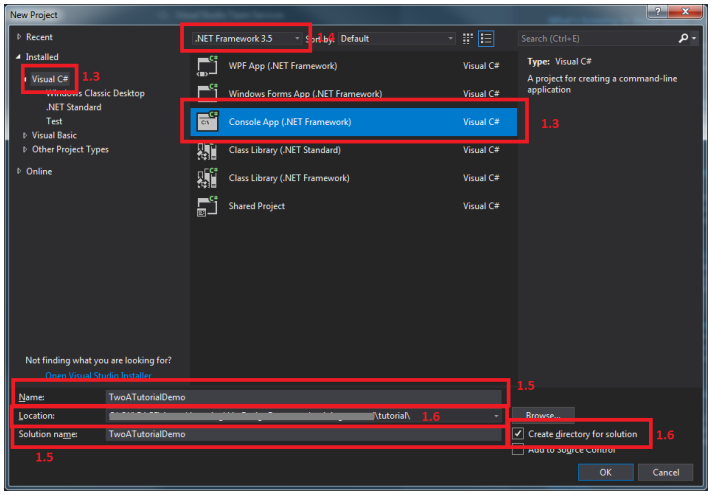
- *TwoA* – contains raw source code for the TwoA asset. The source code is packaged as a Visual Studio project.
- *TwoA_Portable* – contains the portable version of TwoA's Visual Studio project. We will not use it in this tutorial.
- *binary* – contains binary version of the TwoA asset compiled as DLL file: "*TwoA.dll*". The folder also contains another file "*RageAssetManager.dll*". It is a version of the RAGE architecture with which the TwoA asset was tested before its latest release. I recommend using the up-to-date version of the RAGE architecture available on Github. Only in a case of severe compatibility issues between the TwoA asset and the latest version of the RAGE architecture, I recommend using the "*RageAssetManager.dll*" instead.
- *TestApp* – contains a simple dummy project demonstrating the use of the TwoA asset such as instantiation and calls to its API. We will ignore this dummy project in this tutorial and going to create a new one from scratch.
- *manual* – contains documentation for the TwoA asset. The "*TwoA-DesignDocument-v1.2*" provides a technical description of the API. The "*TwoA-UseCaseDocument-v1.2*" provides descriptions of cases of how the asset can be used in games. Some of these cases will be also discussed in this document.

3 Creating a Hello World

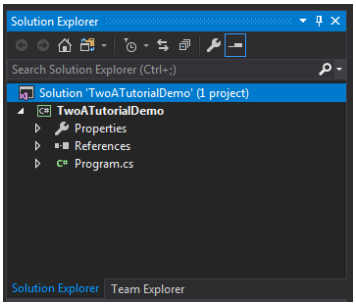
In this section, we will create a new console project and integrate the RAGE architecture and the TwoA asset to it.

3.1 Creating a new project.

1.1. Start Visual Studio.
1.2. In the *File* menu, select *New* -> *Project*.
1.3. In the New Project window, select "*Visual C#*" from the left tree and the option "*Console App (.NET Framework)*" in the right panel.
1.4. Select "*.NET Framework 3.5*" from the options box at the top of the window. .NET Framework 3.5 is the earliest version of the .NET Framework the TwoA asset is compatible with.
1.5. In the "*Name:*" field, input the project name as "*TwoATutorialDemo*". Make sure that the "*Solution name:*" field has the same name as the project name.
1.6. In the "*Location*" field, select the "*tutorial*" folder we created earlier. Make sure that the "*Create directory for solution*" is checked.
1.7. Press the Ok button and wait until the project is created.



The Visual Studio should create a solution called *TwoATutorialDemo* with one project also named *TwoATutorialDemo*. The image below shows how the *Solution Explorer* (on the right side of the main coding window) should look like after the solution and project were created:



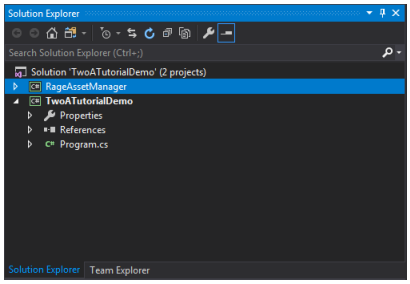
3.2 Importing and referencing the RAGE architecture project.

Now, we will import the RAGE architecture project to the *TwoATutorialDemo* solution. Follow steps below:

2.1. Right-mouse click on the "*Solution 'TwoATutorialDemo'*" in the Solution Explorer to bring a pop-up menu.
2.2. In the popup menu, select "*Add -> Existing Project*".

2.3. In the "*Add Existing Project*" dialogue, select following file "*tutorial\AssetManager-master\RageAssetManager\RageAssetManager.csproj*".

After the Step 2.3, the *Solution Explorer* should resemble the image below. Notice that the second project named *RageAssetManager* was added to the solution.



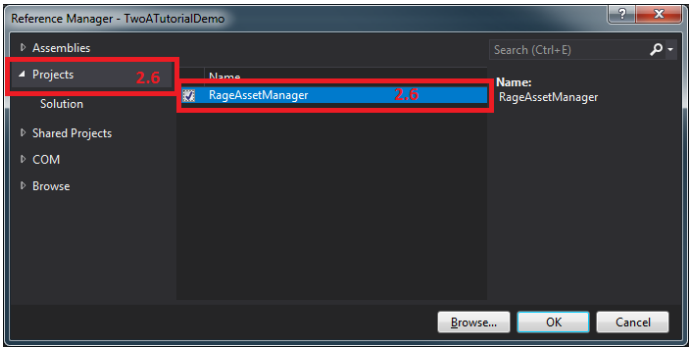
Now, we will reference from the *TwoATutorialDemo* project to the *RageAssetManager* project.

2.4. In the *Solution Explorer*, expand the *TwoATutorialDemo* project and right-mouse click on the "*References*".

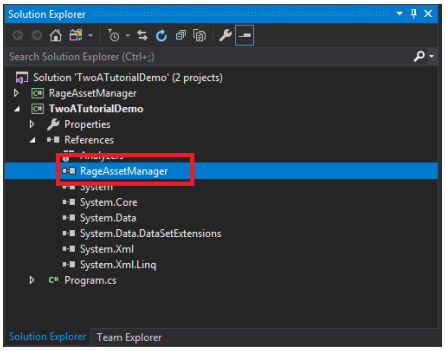
2.5. In the pop-up menu, select "*Add Reference…*".

2.6. In the "*Reference Manager*" dialogue, select "*Projects*" and check the box next to the *RageAssetManager* option as shown in the image below.

2.7. Click the OK button.



The reference to the *RageAssetManager* project should have appeared in the *Solution Explorer* as shown in the image below.



Finally, we add a namespace for the *RageAssetManager* project within the code of the *TwoATutorialDemo* project.

2.8. Double click on the *Program.cs* (within *TwoATutorialDemo* project) to open it.

2.9. Add "`using AssetPackage;`" statement to the code as shown below.

Code snippet C1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AssetPackage; // [SC] 2.9 new code here

namespace TwoATutorialDemo
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

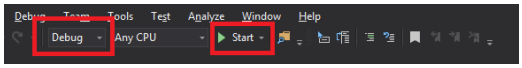
Now, let's compile and run the project to make sure that there are no errors.

2.10. Add following two lines to the Main method and save the changes.

Code snippet C2

```
static void Main(string[] args)
{
    Console.WriteLine("\nPress any key to exit."); // [SC] 2.10 new code here
    Console.ReadKey(); // [SC] 2.10 new code here
}
```

2.11. Set the project to the Debug mode and click the Start button as shown in the image below.



If everything is done right then a console program should appear. Press any key to quite the program. Two or three warning may appear during the compilation. These warning can be ignored for now.

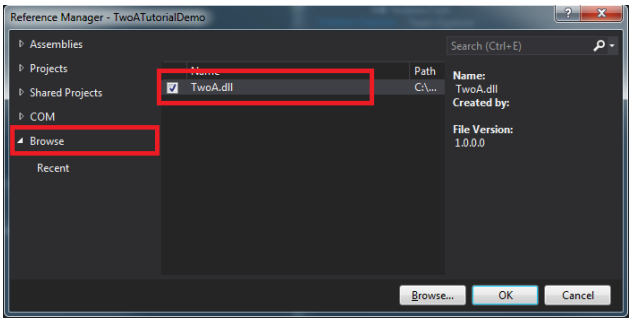
Output O1: for code snippet C2

```
Press any key to exit.
```

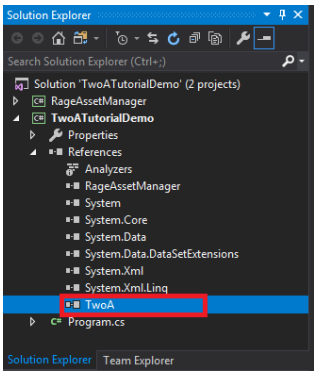
3.3 Importing and referencing the TwoA asset.

The project for the TwoA asset can be imported in the similar manner as the RAGE architecture. However, this time we will import directly the pre-compiled DLL file for the TwoA asset.

3.1. In the *Solution Explorer*, right-mouse click on the "*References*" of the *TwoATutorialDemo* project.
3.2. In the pop-up menu, select "*Add Reference…*".
3.3. In the "*Reference Manager*" dialogue, click on the "*Browse…*" button.
3.4. In the "*Select the files to reference…*" dialogue, navigate to the "*tutorial\HatAsset-1.2-noXML\binary\TwoA.dll*" file and click the *Add* button.
3.5. A reference to the *TwoA.dll* file should appear in the *Browse* section of the *Reference Manager*.
3.6. Click Ok to close the *Reference Manager*.



The *Solution Explorer* should also show the new reference to the TwoA library.



Finally, we add a namespace for the *TwoA* library within the code of the *TwoATutorialDemo* project.

Code snippet C3

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AssetPackage;
using TwoANS; // [SC] adding the namespace for the TwoA library

namespace TwoATutorialDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            TwoA twoA = new TwoA(); // [SC] creating a TwoA instance

            Console.WriteLine("\nPress any key to exit.");
            Console.ReadKey();
        }
    }
}
```

At this point, we are ready to develop a console application that makes use of the *TwoA* asset. Let's create an instance of the TwoA class within the *Main* method. This instance gives access to TwoA's API.
As discussed earlier, TwoA provides two modes of assessment/adaption. The *DifficultyAdapter* class offers assessment and adaptation that relies both on player's accuracy and response time. The *DifficultyAdapterElo* class offers assessment and adaptation that relies only on accuracy. The code below prints information about the two adapters.

Code snippet C4

```
static void Main(string[] args)
{
```

```
    TwoA twoA = new TwoA(); // [SC] creating a new TwoA instance

    Console.WriteLine("Adapter type: " + DifficultyAdapter.Type);
    Console.WriteLine("Adapter description: " + DifficultyAdapter.Description);
    Console.WriteLine("\nAdapter type: " + DifficultyAdapterElo.Type);
    Console.WriteLine("Adapter description: " + DifficultyAdapterElo.Description);

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Output O2: for code snippet C4

```
Adapter type: Game difficulty - Player skill
Adapter description: Adapts game difficulty to player skill. Skill ratings are evaluated for individual players.
Requires player accuracy (0 or 1) and response time. Uses a modified version of the CAP algorithm.

Adapter type: SkillDifficultyElo
Adapter description: Adapts game difficulty to player skill. Skill ratings are evaluated for individual players.
Requires player accuracy (value within [0, 1]) observations. Uses the Elo equation for expected score estimation.

Press any key to exit.
```

4 Using TwoA with both response time and Boolean accuracy

This section demonstrates how TwoA uses both accuracy and response time for its assessment and adaptation. First, we need to enter dummy data for a player and game scenarios.

4.1 Entering player data.

We will expand the body of the *Main* method as shown in the code below. *PlayerNode* class stores data about a player. Its constructor takes following three arguments:

- Type of assessment and adaptation to be performed. In this case, it is based on both accuracy and response time.
- ID of a game in which assessment and adaptation is performed for the player with given ID.
- ID of the player.

Note that the combination of game ID and player ID should be unique within the adaptation mode.
"*Rating = 6*" sets the player's skill rating to 6. This step is optional. If an explicit value for the skill rating is not provided then the rating defaults to 0.01.
Finally, the new instance of the *PlayerNode* is registered with TwoA by calling the *TwoA.AddPlayer* method and passing the instance as an argument.

Code snippet C5

```
static void Main(string[] args)
{
    ... // [SC] this part is same as before

    // [SC] adding player data
    twoA.AddPlayer(
        new PlayerNode(DifficultyAdapter.Type, "dummy game", "dummy player") {
            Rating = 6
        }
    );

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Once registered with TwoA, player data can be easily retrieved back as shown in code below. The "*TwoA.Player*" method retrieves the *PlayerNode* instance matching the supplied combination of adaptation type, game ID, and player ID. The next six lines print values of the player's parameters that are used to calculate player's skill rating. Similar to the rating, these parameters are initialized with predefined values and updated automatically by TwoA. We strongly discourage manual changes to these values.

Code snippet C6

```
static void Main(string[] args)
{
    ... // [SC] this part is same as before

    // [SC] Retrieving a player node by player ID
    PlayerNode playerNode = twoA.Player(DifficultyAdapter.Type, "dummy game", "dummy player");
    // [SC] Printing player parameter
    Console.WriteLine("\nPlayer ID: " + playerNode.PlayerID);
    Console.WriteLine("Skill rating: " + playerNode.Rating);
    Console.WriteLine("Play count: " + playerNode.PlayCount);
    Console.WriteLine("K factor: " + playerNode.KFactor);
    Console.WriteLine("Uncertainty: " + playerNode.Uncertainty);
    Console.WriteLine("Last played: " + playerNode.LastPlayed.ToString(TwoA.DATE_FORMAT));

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Output O3: for code snippet C6

```
Player ID: dummy player
Skill rating: 6
Play count: 0
K factor: 0.0375
Uncertainty: 1
Last played: 2018-01-09T08:25:57

Press any key to exit.
```

4.2 Entering game scenario data

We assume that the game has three scenarios of different difficulties: easy, medium, and hard. We need to add data for each of three scenarios as shown in code snippet below. The *ScenarioNode* class stores data for a single. Its constructor takes following three arguments:

- Type of assessment and adaptation to be performed. In this case, it is based on both accuracy and response time.
- ID of a game in which assessment and adaptation is performed for the player with given ID.
- ID of the scenario.

The combination of game ID and scenario ID should be unique within the adaptation mode.

Rating is a property that stores scenario's difficulty rating. This step is optional. If an explicit value for the skill rating is not provided then the rating defaults to 0.01.

TimeLimit is a duration of time within which the player should complete the scenario. It is measured in milliseconds. While this parameter has a default value, we recommend explicitly setting this parameter with an appropriate value used in the game. In this dummy example, we assume that each scenario should be completed within 15 minutes.

The instance of the *ScenarioNode* class is passed to the *TwoA.AddScenario* method to register the scenario with TwoA. This process is repeated for each scenario.

Note that the easy scenario was assigned the lowest difficulty rating, and the hard scenario was assigned the highest difficulty rating. These differences in ratings reflect relative differences in difficulties among scenarios. In this dummy example, we assume that we know these relative difficulties and can quantify them in ratings. In a real case, such assumptions are not necessary, and ratings can be initialized with default values. Over time, TwoA will be able to calculate more accurate ratings.

Finally, the code snippet includes an example of retrieving *ScenarioNode* instance by game and scenario IDs. The code is self-explanatory.

At this point, TwoA knows that there is one player who plays the dummy game. TwoA also know that the dummy game has three playable scenarios.

Code snippet C7

```
static void Main(string[] args)
{
    ... // [SC] this part is same as before

    // [SC] Adding data for the 1st scenario
    twoA.AddScenario(
        new ScenarioNode(DifficultyAdapter.Type, "dummy game", "easy scenario") {
            Rating = 1.2,
            TimeLimit = 900000
        }
    );
    // [SC] Adding data for the 2nd scenario
    twoA.AddScenario(
        new ScenarioNode(DifficultyAdapter.Type, "dummy game", "medium scenario") {
            Rating = 1.6,
            TimeLimit = 900000
        }
    );
    // [SC] Adding data for the 3rd scenario
    twoA.AddScenario(
        new ScenarioNode(DifficultyAdapter.Type, "dummy game", "hard scenario") {
            Rating = 5.5,
            TimeLimit = 900000
```

```
        }
    );

    // [SC] Retrieving a scenario node by scenario ID
    ScenarioNode scenarioNode = twoA.Scenario(DifficultyAdapter.Type, "dummy game", "hard scenario");
    // [SC] Printing player parameters
    Console.WriteLine("\nScenarioID: " + scenarioNode.ScenarioID);
    Console.WriteLine("Rating: " + scenarioNode.Rating);
    Console.WriteLine("Play count: " + scenarioNode.PlayCount);
    Console.WriteLine("K factor: " + scenarioNode.KFactor);
    Console.WriteLine("Uncertainty: " + scenarioNode.Uncertainty);
    Console.WriteLine("Last played: " + scenarioNode.LastPlayed.ToString(TwoA.DATE_FORMAT));
    Console.WriteLine("Time limit: " + scenarioNode.TimeLimit);

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Output O4: for code snippet C7

```
ScenarioID: hard scenario
Rating: 5.5
Play count: 0
K factor: 0.0375
Uncertainty: 1
Last played: 2018-01-09T08:28:49
Time limit: 900000

Press any key to exit.
```

4.3 Using TwoA's adaptation

When a player starts the game, the game has to decide which scenario the player should play. This decision can be delegated to TwoA. This is the adaptive capability of the TwoA asset. TwoA will try to select a scenario with difficulty rating that matches the best the player's skill rating. Such matching is regulated by the parameter *P*, a desired probability that a player will be able to successfully complete a scenario. *P* takes a value between 0 and 1. The default (and recommended) value is 0.75 meaning that a player should have 75% chance of successfully completing a scenario and 25% chance of failing in the scenario. By comparing skill and difficulty ratings, TwoA recommends a scenario where the player's chance of success is closest to *P*.

Skill and difficulty ratings are measured along the same scale and, therefore, comparable to each other. If the player's skill rating is higher than the scenario's difficulty rating then the player has above 50% chance of successfully completing the scenario. If the player's skill rating is lower than the scenario's difficulty rating then the player has below 50% chance of successfully completing the scenario. If the player's skill rating is equal to the scenario's difficulty rating then the player has 50% chance of completing the scenario.

Let's explore adaptation using our dummy example with three game scenarios. Code snippet C8 shows example adaptation code. The game can request TwoA recommendation for a scenario by calling the *TwoA.TargetScenarioID* method. As arguments, the method requires the previously explored values: adaptation type, game ID, and player ID. Behind the scene, the method retrieves the instance of the *PlayerNode* class, extracts player's skill rating, and uses the rating to match the scenarios. The method returns ID of the scenario recommended by TwoA.

To clarify some inner workings of TwoA, the code snippet calls the *TwoA.TargetScenarioID* method 10 times consecutively. In real case, this method should be called only once for each recommendation request from a game. In our dummy example, the hard scenario (difficulty rating 5.5) is the closest match to the dummy player (skill rating 6). As a result, the hard scenario should be the most frequently recommended scenario after 10 calls of the method. Occasionally, TwoA will recommend the other scenarios. This controlled randomness in recommendations is important for the player's efficiency of learning. It allows player to consolidate new knowledge via replays of earlier easier scenarios and acquire new knowledge via introduction of more challenging scenarios where player's success rate less than *P*.

Code snippet C8

```
static void Main(string[] args)
{
```

```
    ... // [SC] this part is same as before

    // [SC] Ask TwoA for a scenario recommendation for the dummy player (P = 0.75)
    Console.WriteLine("");
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Recommended scenario's ID: "
            + twoA.TargetScenarioID(DifficultyAdapter.Type, "dummy game", "dummy player"));
    }

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Output O5: for code snippet C8

```
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: medium scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario

Press any key to exit.
```

4.4 Using TwoA's assessment

TwoA's assessment is mainly about updating, aka recalculating, player's skill rating and scenario's difficulty rating. Such updates have two main purposes:

- more precise estimations of ratings to more accurately reflect player's true skill and scenario's true difficulty
- to account for and reflect changes in a player's learning state

Outcomes of the assessment also offer more practical pedagogical benefits to the teachers as means for learning analytics. These benefits will be discussed in later sections. For now, we concentrate on use of assessment within the game.

After a player completes a scenario, the game should have following performance metrics:

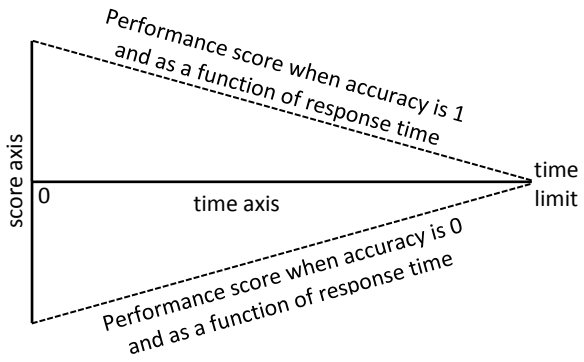
- Response time is duration of time the player spent playing the scenario. It should be measured in milliseconds. If the player was not able to complete the scenario within the time limit then the time limit should be considered as the response time.
- Accuracy is a boolean factor (either 1 or 0) indicating whether the player completed the scenario successfully (1) or unsuccessfully (0) irrespective of the time limit.

Given these two metrics, TwoA can calculate an internal performance score for the player. This performance score is further used to adjust the player's skill rating and scenario's difficulty rating. The graph below shows how accuracy and response time determine the performance score. The table on the left helps to interpret the graph. Succinctly, if accuracy is 1 then the performance score is positive. Low response time is rewarded with a higher score than high response time. If accuracy is 0 then the performance score is negative. Fast incorrect response times (e.g., random guesses) are punished more than slow incorrect response times. Reaching a time limit results in a score of 0.

In general, a higher positive performance score results in a higher increase in the player's skill rating and a mirror decrease in the scenario's difficulty rating. A higher negative performance score results in a higher decrease in the player's skill rating and a mirror increase in the scenario's difficulty rating.

Another important factor that determines how the ratings should change is the difference between the player's skill rating and scenario's difficulty rating at the time of recommendation. Succeeding in a more difficult scenario results in a higher increase in the player's skill rating than succeeding in a less challenging problem. Similarly, failing in a more difficult scenario results in a smaller decrease in the player's skill rating than failing in a less difficult scenario. Due to a combined effect of both performance score and initial differences in ratings, it is also possible that the player's rating may decrease due to high response time even if the player succeeded at the end.

	Low response time	High response time	Time limit reached
Accuracy = 1	High positive score	Low positive score	0
Accuracy = 0	High negative score	Low negative score	0



Now, let's explore concrete assessment code with our dummy example. Code snippet C9 show several examples of TwoA's assessment with different values of accuracy and response time. At first, we create two clones of the player and scenario. Since the assessment will recalculate the ratings, these clones will preserve the initial ratings for later uses.

The *TwoA.UpdateRatings* method is used for assessment. It requires six following arguments:

- An instance of the *PlayerNode* class; The skill rating (and some other parameters) in this instance will be updated.
- An instance of the *ScenarioNode* class; The difficulty rating in this (also some other parameters) in this instance will be updated.
- Player's response time measured in milliseconds.
- Player's accuracy that is either 0 or 1.
- A Boolean value; If it is false, only player's skill rating will be recalculated and updated, and scenario's rating will remain the same. The value should be true unless you are highly confident with difficulty ratings you already have and know the ratings should not change.
- A custom K factor that can control magnitude of changes in ratings. If this argument is 0 then TwoA uses a dynamically calculated K factor. It is highly recommended to pass 0 as an argument and avoid using a custom K factor.

In Example 1, we assume that the player was able to successfully complete the scenario within two minutes (the time limit is 15 minutes as we defined previously). In the output, you will observe a moderate increase in player's rating and corresponding decrease in scenario's rating.

In Example 2, we assume that the player also succeeded but only after 10 minutes of gameplay. While player's rating increases again, the increase is minute compared to one in Example 1 where the player spent only two minutes. Player's performance was not as good as expected, which is reflected in a smaller increase in the rating.

In Example 3, we assume that the player failed within 2 minutes. This time, the player's rating decreases and the scenario's rating increases.

Code snippet C9

```
static void Main(string[] args)
{
    ... // [SC] this part is same as before

    // [SC] for purpose of this demo, set calibration length to 0 effectively removing the calibration phase
    twoA.SetCalLength(DifficultyAdapter.Type, 0);

    // [SC] at first create some clones for a comparison purpose
    PlayerNode playerCloneOne = playerNode.ShallowClone();
    ScenarioNode scenarioCloneOne = scenarioNode.ShallowClone();
    PlayerNode playerCloneTwo = playerNode.ShallowClone();
```

```
    ScenarioNode scenarioCloneTwo = scenarioNode.ShallowClone();

    // [SC] print the initial ratings
    Console.WriteLine("\nInitial player rating: " + playerNode.Rating);
    Console.WriteLine("Initial scenario rating: " + scenarioNode.Rating);

    // [SC] Example 1: print ratings after a success in the hard scenario
    twoA.UpdateRatings(playerNode, scenarioNode, 120000, 1, true, 0);
    Console.WriteLine("\nPlayer rating after success: " + playerNode.Rating);
    Console.WriteLine("Scenario rating after success: " + scenarioNode.Rating);

    // [SC] Example 2: print ratings after a delayed success in the hard scenario
    twoA.UpdateRatings(playerCloneOne, scenarioCloneOne, 600000, 1, true, 0);
    Console.WriteLine("\nPlayer rating after delayed success: " + playerCloneOne.Rating);
    Console.WriteLine("Scenario rating after delayed success: " + scenarioCloneOne.Rating);

    // [SC] Example 3: print ratings after a failure in the hard scenario
    twoA.UpdateRatings(playerCloneTwo, scenarioCloneTwo, 120000, 0, true, 0);
    Console.WriteLine("\nPlayer rating after fail: " + playerCloneTwo.Rating);
    Console.WriteLine("Scenario rating after fail: " + scenarioCloneTwo.Rating);

    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

Output O6: for code snippet C9

```
Initial player rating: 6
Initial scenario rating: 5.5

Player rating after success: 6.02325541671409
Scenario rating after success: 5.47674458328591

Player rating after delayed success: 6.00560541671409
Scenario rating after delayed success: 5.49439458328591

Player rating after fail: 5.96589291671409
Scenario rating after fail: 5.53410708328591

Press any key to exit.
```

Here is the complete code that was written during this section:

A complete code for Section 4

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AssetPackage;
using TwoANS;

namespace TwoATutorialDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            TwoA twoA = new TwoA(); // [SC] creating a new TwoA instance

            Console.WriteLine("Adapter type: " + DifficultyAdapter.Type);
            Console.WriteLine("Adapter description: " + DifficultyAdapter.Description);
            Console.WriteLine("\nAdapter type: " + DifficultyAdapterElo.Type);
            Console.WriteLine("Adapter description: " + DifficultyAdapterElo.Description);


            // [SC] adding player data
            twoA.AddPlayer(
                new PlayerNode(DifficultyAdapter.Type, "dummy game", "dummy player") {
                    Rating = 6
                }
            );

            // [SC] Retrieving a player node by player ID
            PlayerNode playerNode = twoA.Player(DifficultyAdapter.Type, "dummy game", "dummy player");
```

```
// [SC] Printing player parameter
Console.WriteLine("\nPlayer ID: " + playerNode.PlayerID);
Console.WriteLine("Skill rating: " + playerNode.Rating);
Console.WriteLine("Play count: " + playerNode.PlayCount);
Console.WriteLine("K factor: " + playerNode.KFactor);
Console.WriteLine("Uncertainty: " + playerNode.Uncertainty);
Console.WriteLine("Last played: " + playerNode.LastPlayed.ToString(TwoA.DATE_FORMAT));


// [SC] Adding data for the 1st scenario
twoA.AddScenario(
    new ScenarioNode(DifficultyAdapter.Type, "dummy game", "easy scenario") {
        Rating = 1.2,
        TimeLimit = 900000
    }
);
// [SC] Adding data for the 2nd scenario
twoA.AddScenario(
    new ScenarioNode(DifficultyAdapter.Type, "dummy game", "medium scenario") {
        Rating = 1.6,
        TimeLimit = 900000
    }
);
// [SC] Adding data for the 3rd scenario
twoA.AddScenario(
    new ScenarioNode(DifficultyAdapter.Type, "dummy game", "hard scenario") {
        Rating = 5.5,
        TimeLimit = 900000
    }
);

// [SC] Retrieving a scenario node by scenario ID
ScenarioNode scenarioNode = twoA.Scenario(DifficultyAdapter.Type, "dummy game", "hard scenario");
// [SC] Printing player parameters
Console.WriteLine("\nScenarioID: " + scenarioNode.ScenarioID);
Console.WriteLine("Rating: " + scenarioNode.Rating);
Console.WriteLine("Play count: " + scenarioNode.PlayCount);
Console.WriteLine("K factor: " + scenarioNode.KFactor);
Console.WriteLine("Uncertainty: " + scenarioNode.Uncertainty);
Console.WriteLine("Last played: " + scenarioNode.LastPlayed.ToString(TwoA.DATE_FORMAT));
Console.WriteLine("Time limit: " + scenarioNode.TimeLimit);

// [SC] Ask TwoA for a scenario recommendation for the dummy player (P = 0.75)
Console.WriteLine("");
for (int i = 0; i < 10; i++) {
    Console.WriteLine("Recommended scenario's ID: "
        + twoA.TargetScenarioID(DifficultyAdapter.Type, "dummy game", "dummy player"));
}

// [SC] for purpose of this demo, set calibration length to 0 effectively removing the calibration phase
twoA.SetCalLength(DifficultyAdapter.Type, 0);

// [SC] at first create some clones for a comparison purpose
PlayerNode playerCloneOne = playerNode.ShallowClone();
ScenarioNode scenarioCloneOne = scenarioNode.ShallowClone();
PlayerNode playerCloneTwo = playerNode.ShallowClone();
ScenarioNode scenarioCloneTwo = scenarioNode.ShallowClone();

// [SC] print the initial ratings
Console.WriteLine("\nInitial player rating: " + playerNode.Rating);
Console.WriteLine("Initial scenario rating: " + scenarioNode.Rating);

// [SC] Example 1: print ratings after a success in the hard scenario
twoA.UpdateRatings(playerNode, scenarioNode, 120000, 1, true, 0);
Console.WriteLine("\nPlayer rating after success: " + playerNode.Rating);
Console.WriteLine("Scenario rating after success: " + scenarioNode.Rating);

// [SC] Example 2: print ratings after a delayed success in the hard scenario
twoA.UpdateRatings(playerCloneOne, scenarioCloneOne, 600000, 1, true, 0);
Console.WriteLine("\nPlayer rating after delayed success: " + playerCloneOne.Rating);
Console.WriteLine("Scenario rating after delayed success: " + scenarioCloneOne.Rating);

// [SC] Example 3: print ratings after a failure in the hard scenario
twoA.UpdateRatings(playerCloneTwo, scenarioCloneTwo, 120000, 0, true, 0);
Console.WriteLine("\nPlayer rating after fail: " + playerCloneTwo.Rating);
Console.WriteLine("Scenario rating after fail: " + scenarioCloneTwo.Rating);
```

```
            Console.WriteLine("\nPress any key to exit.");
            Console.ReadKey();
        }
    }
}
```

Finally, a complete output should look like this below:

Output O8: for code snippet C11

```
Adapter type: Game difficulty - Player skill
Adapter description: Adapts game difficulty to player skill. Skill ratings are evaluated for individual players.
Requires player accuracy (0 or 1) and response time. Uses a modified version of the CAP algorithm.

Adapter type: SkillDifficultyElo
Adapter description: Adapts game difficulty to player skill. Skill ratings are evaluated for individual players.
Requires player accuracy (value within [0, 1]) observations. Uses the Elo equation for expected score estimation.

Player ID: dummy player
Skill rating: 6
Play count: 0
K factor: 0.0375
Uncertainty: 1
Last played: 2018-01-09T11:57:30

ScenarioID: hard scenario
Rating: 5.5
Play count: 0
K factor: 0.0375
Uncertainty: 1
Last played: 2018-01-09T11:57:30
Time limit: 900000

Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario

Initial player rating: 6
Initial scenario rating: 5.5

Player rating after success: 6.02325541671409
Scenario rating after success: 5.47674458328591

Player rating after delayed success: 6.00560541671409
Scenario rating after delayed success: 5.49439458328591

Player rating after fail: 5.96589291671409
Scenario rating after fail: 5.53410708328591

Press any key to exit.
```

5 Using TwoA with continuous accuracy

This section demonstrates how TwoA can perform adaptation and assessment based on only accuracy as a performance metric. This time, accuracy is a continuous value in an interval [0, 1] instead of a Boolean value.

The code snippet C11 provide the complete Main method that uses adaptation and assessment based on accuracy only. The most of the code should be already familiar due to the previous section where it was described in details. We will zoom in only on different parts.

The code for adding player and scenario data is remains mainly the same. The first significant difference is that the *Type* variable is passed from the *DifficultyAdapterElo* class instead of *DifficultyAdapter* class. The second difference is that we do not need to initialize the *TimeLimit* property in the *ScenarioNode* class. Since, this mode uses only accuracy this property is irrelevant and ignored even if it is initialized with a custom value.

The code for adaptation also remains the same. Note that we are gain using the *DifficultyAdapterElo* class instead of *DifficultyAdapter* class.

In the code for assessment, we again call the *TwoA.UpdateRatings* method. Note that we are passing 0 as an argument for the response time parameter. In fact, any double value can be passed as response time. It does not matter since this argument is ignored during calculations. Also note that the *TwoA.UpdateRatings* method knows to use the accuracy-only adaptation because it looks up adaptation types from the instances of the *PlayerNode* and *ScenarioNode* classes.

How the player and scenario rating change is governed by the expected accuracy. Expected accuracy (a value between 0 and 1) is calculated from the difference between scenario's and player's ratings. Expected accuracy is a value between 0 and 1 and indicates how likely it is that the player will succeed in the scenario. If the two ratings are equal then the expected accuracy is 0.5 (50% chance of success). To further simplify the explanation, let's assume that the player's rating is always higher than the scenario's rating. It is a reasonable assumption due to $P = 0.75$. With this assumption, a bigger difference between the two ratings results in a higher expected accuracy that is above 0.5. If the player's observed accuracy is lower than the expected accuracy then the player's rating will decrease. If the player's observed accuracy is higher than the expected accuracy then the player's rating will increase. Changes in the player's ratings are directly proportional to the differences between two accuracies. The scenario's rating changes by the same amount but in an opposite direction than the player's rating. Finally, if the player performs as expected then the two ratings do not change.

In Example 1, we assume that the player had an accuracy of approximately 0.62. In other words, the player was 62% successful. In this example, the expected accuracy is also 0.62. Therefore, the player performed exactly as expected resulting in no changes in the two ratings.

In Example 2, we assume that the player had an accuracy of 1.0. This is considerably above the expected accuracy resulting in an increase in the player's rating and decrease in the scenario's rating.

In Example 3, we assume that the player failed showing 0 accuracy. The player's rating decreases and the scenario's rating increases by the same amount.

Code snippet C11

```
static void Main(string[] args)
{
    TwoA twoA = new TwoA(); // [SC] creating a new TwoA instance

    // [SC] adding player data
    PlayerNode playerNode = new PlayerNode(DifficultyAdapterElo.Type
        , "dummy game", "dummy player") {
        Rating = 6
    };
    twoA.AddPlayer(playerNode);

    // [SC] Adding data for the 1st scenario
    twoA.AddScenario(
        new ScenarioNode(DifficultyAdapterElo.Type, "dummy game", "easy scenario") {
            Rating = 1.2
        }
    );
    // [SC] Adding data for the 2nd scenario
    twoA.AddScenario(
        new ScenarioNode(DifficultyAdapterElo.Type, "dummy game", "medium scenario") {
            Rating = 1.6
        }
```

```
    );
    // [SC] Adding data for the 3rd scenario
    ScenarioNode scenarioNode = new ScenarioNode(DifficultyAdapterElo.Type, "dummy game"
        , "hard scenario") {
        Rating = 5.5
    };
    twoA.AddScenario(scenarioNode);


    // [SC] Ask TwoA for a scenario recommendation for the dummy player (P = 0.75)
    Console.WriteLine("");
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Recommended scenario's ID: "
            + twoA.TargetScenarioID(DifficultyAdapterElo.Type, "dummy game", "dummy player"));
    }

    // [SC] at first create some clones for a comparison purpose
    PlayerNode playerCloneOne = playerNode.ShallowClone();
    ScenarioNode scenarioCloneOne = scenarioNode.ShallowClone();
    PlayerNode playerCloneTwo = playerNode.ShallowClone();
    ScenarioNode scenarioCloneTwo = scenarioNode.ShallowClone();

    // [SC] print the initial ratings
    Console.WriteLine("\nInitial player rating: " + playerNode.Rating);
    Console.WriteLine("Initial scenario rating: " + scenarioNode.Rating);

    // [SC] Example 1: print ratings after 62% success in the hard scenario
    // [SC] the expected accuracy is also 0.6224599
    twoA.UpdateRatings(playerNode, scenarioNode, 0, 0.6224599, true, 0);
    Console.WriteLine("\nPlayer rating after 62% success: " + playerNode.Rating);
    Console.WriteLine("Scenario rating after 62% success: " + scenarioNode.Rating);

    // [SC] Example 2: print ratings after 100% success in the hard scenario
    twoA.UpdateRatings(playerCloneOne, scenarioCloneOne, 0, 1.0, true, 0);
    Console.WriteLine("\nPlayer rating after 100% success: " + playerCloneOne.Rating);
    Console.WriteLine("Scenario rating after 100% success: " + scenarioCloneOne.Rating);

    // [SC] Example 3: print ratings after 0% (failure) in the hard scenario
    twoA.UpdateRatings(playerCloneTwo, scenarioCloneTwo, 0, 0.0, true, 0);
    Console.WriteLine("\nPlayer rating after a fail: " + playerCloneTwo.Rating);
    Console.WriteLine("Scenario rating after a fail: " + scenarioCloneTwo.Rating);


    Console.WriteLine("\nPress any key to exit.");
    Console.ReadKey();
}
```

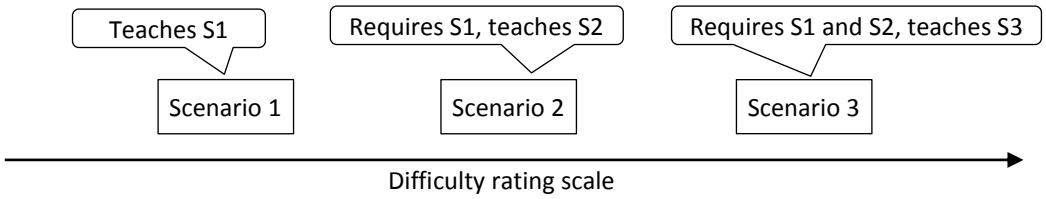
Output O8: for code snippet C11

```
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: hard scenario
Recommended scenario's ID: medium scenario
Recommended scenario's ID: hard scenario

Initial player rating: 6
Initial scenario rating: 5.5

Player rating after 62% success: 5.99999999840106
Scenario rating after 62% success: 5.50000000159894

Player rating after 100% success: 6.01249421608544
Scenario rating after 100% success: 5.48750578391456

Player rating after a fail: 5.97940046608544
Scenario rating after a fail: 5.52059953391456

Press any key to exit.
```

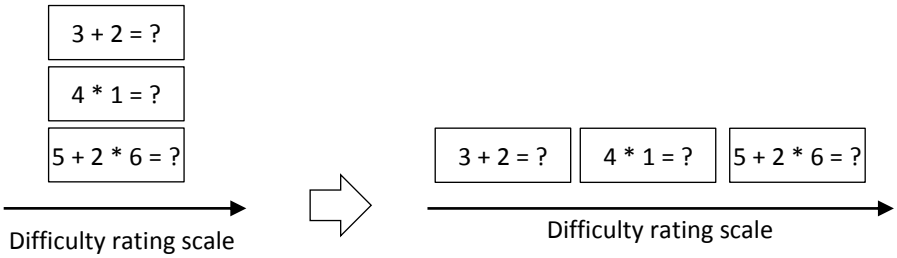
6 Learning with TwoA

6.1 Designing game scenarios

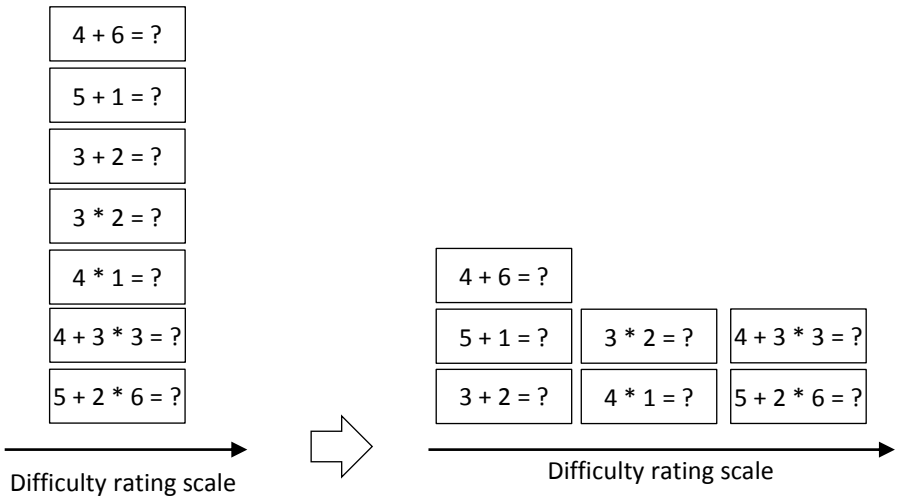
In this section, we will review how the TwoA asset affects the learning experience of the player. First, we explore the pedagogically sound approach for chunking a game into playable scenarios. Let's assume that a serious game is designed to teach a set of three skills S1, S2, and S3. We can further assume that the skill S1 is a prerequisite for the skill S2, and the skill S2 is a prerequisite for the skill S3. In other words, to learn the skill S2, the player needs to have acquired the skill S1. Similarly, to learn the skill S3, the player needs to have acquired the skills S1 and S2. This dependency among the skills taught by the game can be used to design game scenarios of increasing complexity as shown in figure below.



Let's explore a more concrete example of game scenarios based on a hypothetical serious game for teaching basic arithmetic skills such as addition and multiplication. Learning addition is a prerequisite for learning multiplication. Knowledge of both operators is required for learning operator precedence in more complex equations. We can create three different scenarios where each scenario requires solving an arithmetic problem. These scenarios are shown in figure below. Next, we can assign some starting difficulty ratings to the scenarios (e.g., 0.01 for all three scenarios) and let players play the scenarios. Based on players performance metrics, TwoA will be able to calculate relative difficulties of scenarios and as well as expertise levels of individual players.

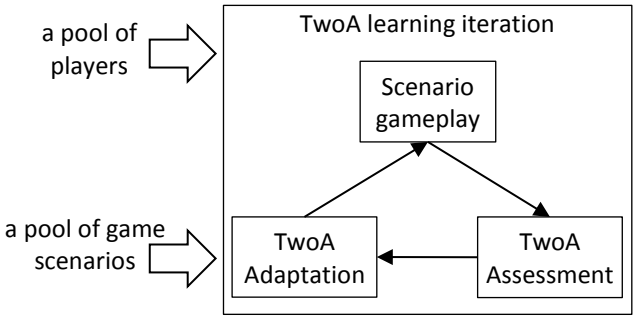


To provide some variety in the gameplay and learning, it is also recommended to include several scenarios with the same learning content but with slight variations. In our example arithmetic game, we can include multiple scenarios with different addition problems. All addition scenarios will have similar difficulty ratings but provide some variety in the gameplay. During adaptation, TwoA recommends a scenario that was played the least among the scenarios of the similar difficulty ratings.



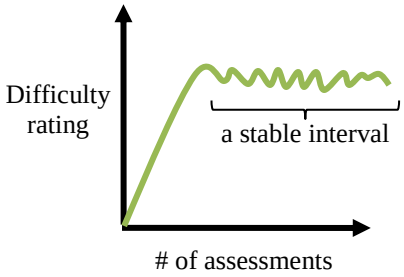
6.2 Learning iteration.

We can see a gameplay of a single scenario as a learning iteration. On the one hand, the player may play a scenario that was not administered before to the player. In this case, the learning iteration involves acquisition of new skills. On the other hand, the player may also replay one of the previously played scenarios. Such replay facilitates practice of previously acquired skills. Practice is an important factor for learning since it facilitates deeper consolidation of skills. A sufficient skill mastery requires both knowledge acquisition and consolidation. Outcomes of the learning iteration need to be assessed. This assessment has two purposes. First, iterative assessments increase accuracies of estimated skill and difficulty ratings. More assessments increase the probability that the skill rating reflects player's actual skills. Similarly, more assessments increase the probability that the difficulty rating reflects scenario's true difficulty. The exact number of assessments required to maximize accuracy of ratings is hard to predict since there are many factors (e.g., number of scenarios, number of players, player's skill level at start, etc.) that can influence this number.

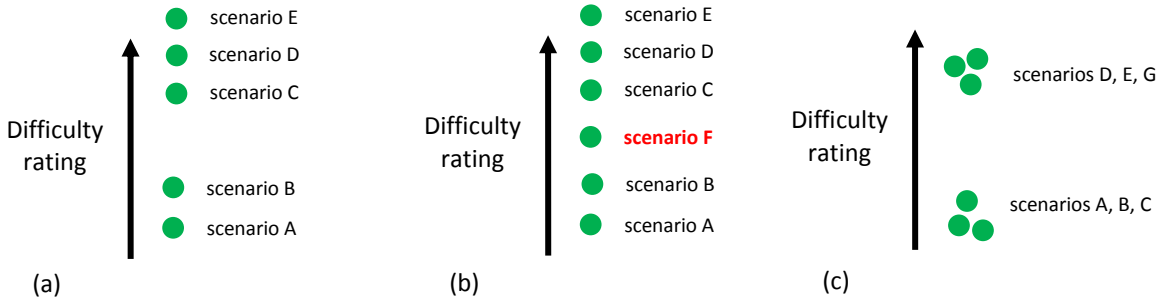


6.3 Interpreting difficulty ratings

One indicator that a difficulty rating was estimated accurately enough is when the rating becomes stable by fluctuating up and down around some mean.



Analysis of stable difficulty ratings can be beneficial in further refining the game design to improve its pedagogical effectiveness and efficiency. First, we can identify gaps in learning content. As an example, let's take the figure below. In (a), the ratings for scenarios B and C are spaced from each more than the average spacing between other pairs of scenarios. This indicates there is a higher than expected jump in difficulty from B to C that might hinder a learning experience. In (b), introduction of scenario F makes spacing between modules the same contributing to a more gradual increase in difficulty.

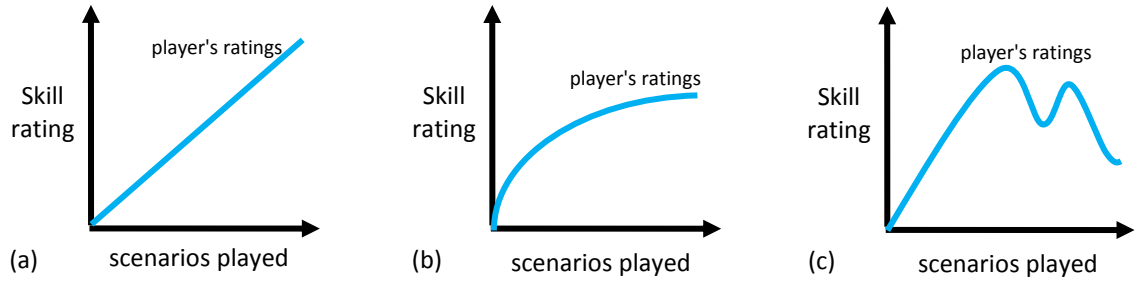


Additionally, we can analyze how much scenarios are different or similar in terms of difficulty ratings. Such analysis helps us verify whether the assumptions put during the game design phase actually hold. Scenarios that are similar in learning content should form a coherent cluster. In (c) of the figure above, scenarios form two clusters based on similarity of their difficulty ratings. It is likely that scenarios in the same cluster require same skills and offer similar learning content. Alternatively, if there is a scenario that has learning content different from the rest of the scenarios in the same cluster then redesign may be necessary. For example, the scenario may involve learning content irrelevant to the learning domain of the game. On the other hand, the scenario may be outside of the cluster of scenarios with which it was originally assumed to have similar learning content. It may be an indication that factors not considered during a design phase may be influencing a learning process. For example, TwoA will calculate different difficulty ratings for the two arithmetic problems involving the multiplication operation: "10 * 10 = 100" and "7 * 8 = ?". In this particular case, difficulty is defined no only by the type of arithmetic operation but also by the strategy a player can use. Multiplying two tens requires fast and simple strategy of combining two zeros. On the other hand, the second problem requires a robust knowledge of the multiplication table from which the answer should be retrieved. This is why the second problem is more difficult than the first one. TwoA can help to identify such obscure differences and help to optimize game's learning content.

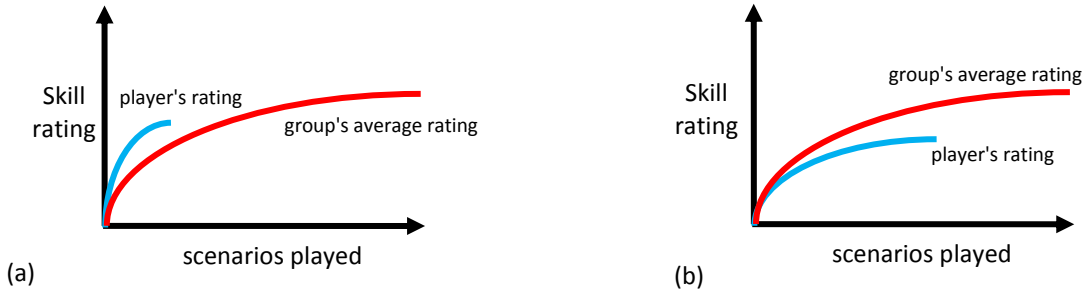
6.4 Interpreting a skill rating

While the stability criterion also holds true for a skill rating, it is also possible that the skill rating was evaluated accurately without reaching a stable point. In the latter case, considerable changes in the skill rating may reflect player's learning progress. This is the second purpose of the iterative assessment. It allows tracking of the player's learning progress. We can plot how player's skill rating changed over several assessment iterations. The resulting curve can be seen as player's learning curve. The learning curve can be very useful as a feedback to support a teacher

in conducting a formative assessment and identifying points of intervention. As shown in the figure below, three examples of patterns that can be identified in player's skill ratings. (a) Player's learning at a steady rate. (b) Player's learning rate is slowing down due to a ceiling effect or barriers to learning. Some attention should be paid by the instructor. (c) Player's rating dropped after playing some scenarios. It is a clear indication that the player is underperforming in these scenarios. A teacher is advised to provide a feedback.



Additionally, player's skill rating can be compared to the group average as shown in the figure below. (a) Player's learning rate is above group average indicating that the player is a fast learner and performing quite well. (b) Player's learning rate is slow compared to group's average indicating that the player may have some learning difficulties and require feedback from a teacher.



Finally, player's skill rating can always be compared to the scenarios' difficulty ratings to identify the player's learning progress in terms of skills acquired and mastered. It is likely that the player acquired all skills necessary for solving scenarios with difficulty ratings lower than the player's skill rating. Furthermore, lower the difficulty rating of the scenario relative to the skill rating higher the player's mastery of skills necessary to solve that scenario. It is because the player had more opportunity to practice and consolidate those skills.