Bank Overhaul Mod Documentation

Voicea Andrei

January 6, 2025

Contents

1 Introduction

- **Project Purpose:** The purpose of the project is to improve the functionality of banking systems while introducing new features.
- Overview of main functionalities: I have added the possibility to get interest for the deposit that is made in the bank, balanced the way the loan system works, and the possibility to configure the whole way, from how the interest can behave, to the choice of using my version of the loan system to the player attributes that modify the interest.

2 Project Structure

- **File and folder organization:** The file structure is simple, the .dfmod file extension is located in a document named "Mods", to be directly extracted when downloading by vortex in the appropriate folder without having to be moved manually.
- Project dependencies: It has no necessary dependencies, but it is necessary to be loaded after the Roleplay and Realism mod to be able to modify the bank loan limit in this mod and avoid conflicts.

3 Technical Requirements

3.1 Software Requirements

- Unity Hub
- Unity: if you use the android version download: 2022.3.30f1 and for Windows, Linux and OSX versions: 2019.4.40f1. **It is preferable to constantly check when projects are modified as this version is subject to change.**
- Daggerfall Unity Files
- Daggerfall Unity Android Files
- Bank Overhaul Mod
- Visual Studio
- Sourcetree (optional)
- TexMaker (optional)

4 Installation Guide

1. To install the mod files go on my GitHub link and click on the Code button -> Download ZIP

5 Usage

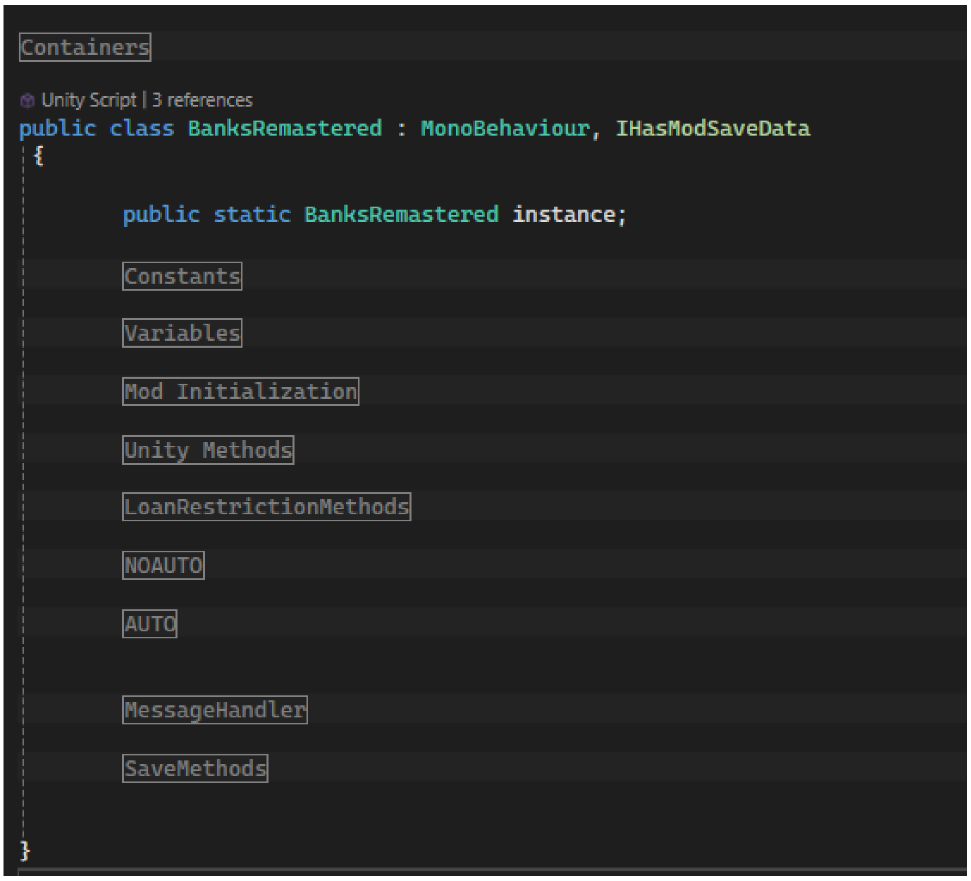
5.1 Main Functionalities

- In the project files you have all the scripts necessary to change this mod, I also added the ZipModScript.bat which is used to compress all the mod files for every operating system, you can use it but modify the paths in order to work.

6 Source Code

6.1 Code Organization

- Description of modules and files: Currently there is only one script that handles all the functionality of the mod. The script contains the initialization logic of the mode, the configurations that can be changed and their loading in the game, the new added functionality that is divided into the one for deposit and the one for bank loan, respectively a message handler and the logic for saving, loading data.



```
Containers

Unity Script | 3 references
public class BanksRemastered : MonoBehaviour, IHasModSaveData
{

        public static BanksRemastered instance;

        Constants

        Variables

        Mod Initialization

        Unity Methods

        LoanRestrictionMethods

        NOAUTO

        AUTO

        MessageHandler

        SaveMethods

}
```

Figure 1: Image showing the project's modules.

6.2 Code Examples

Listing 1: Containers

```
public struct BankStruct
{
    public long BankDepositDate;
    public long RemainedDays;
    public bool BonusRewarded;
}

[FullSerializer.fsObject("v1")]
public class BanksRemasteredSaveData
{
    public BankStruct[] bankstruct = new BankStruct[
       BanksRemastered.BankStructSize];
    public bool HasLoan;
}
```

BankStruct is used to store the new variables of the banks, works as an extender to the already existent bank structures. The second interior class is used for saving.

Listing 2: Mod Initialization

```
        static Mod mod;

        [Invoke(StateManager.StateTypes.Start, 0)]
        public static void Init(InitParams initParams)
        {
            mod = initParams.Mod;
            var go = new GameObject(mod.Title);
            go.AddComponent<BanksRemastered>();
            mod.LoadSettingsCallback = LoadSettings;
            mod.SaveDataInterface = instance;
            FormulaHelper.RegisterOverride(mod, "
               CalculateMaxBankLoan", (Func<int>)
               CalculateMaxBankLoan);
            mod.IsReady = true;
        }

    private static void LoadSettings(ModSettings modSettings,
       ModSettingsChange change)
    {
        try
        {
            AutomaticDeposit = mod.GetSettings().GetValue<bool>("
               GeneralSettings", "AllowAutomaticDepositing");
            RestrictLoanOption = mod.GetSettings().GetValue<bool>(
               "GeneralSettings", "AllowLoanRestriction");
            LoanAmount = loanVals[mod.GetSettings().GetInt("
               GeneralSettings", "LoanMaxPerLevel")];
            BonusRate = mod.GetSettings().GetValue<float>("
               GeneralSettings", "BonusRate");
            DepositDaysNumber = mod.GetSettings().GetValue<int>("
               GeneralSettings", "DepositDays");
            BonusRateWithStats = mod.GetSettings().GetValue<bool>(
               "MiscSettings", "AllowStatsToCalculateBonusRate");
            BonusRateOffset = BonusRateOffsetVals[mod.GetSettings
               ().GetInt("MiscSettings", "BonusRateOffset")];

        }
        catch (Exception ex)
        {
            Debug.LogError($"Failed␣to␣load␣settings:␣{ex.Message}
               ");
        }

    }
```

The Main class is structured as a singleton, one purpose of this class is to initialize the mod which is mostly done in the Init function, note that you need the Daggerfall-Workshop.Game.Utility.ModSupport library in order to use it, here you can also find the override to the MaxBankLoan function and the LoadSettings method which will make sure that all the configurations are loaded.

Listing 3: Loan Methods

```
    private void RestrictLoaning(TransactionType type,
       TransactionResult result, int amount)
    {

         if (result == TransactionResult.NONE)
             HasLoan = true;

    }

    private void EnableLoaning(TransactionType type,
       TransactionResult result, int amount)
    {
        int index = GameManager.Instance.PlayerGPS.
           CurrentRegionIndex;
        if ((result == TransactionResult.NONE &&
           DaggerfallBankManager.HasLoan(index) == false) ||
           result == TransactionResult.OVERPAID_LOAN)
            HasLoan = false;


    }

    public static int CalculateMaxBankLoan()
    {
        return GameManager.Instance.PlayerEntity.Level *
           LoanAmount;
    }
```

Here we can see the loan methods, the system works as follows: if the player make a loan anywhere the flag HasLoan is raised, if this boolean is true the LoanAmount will be set to 0, the EnableLoaning Method will be executed every time the player pays back the right amount of money and will reset the flag, after the flag is reseted the LoanAmount will be the initialLoanAmount. And the last method is just the same formula used in game with the LoanAmount as a new variable.

Listing 4: Depositing Manually

```
    private void SetDepositTimer(TransactionType type,
       TransactionResult result, int amount)
        {

            if (result == TransactionResult.NONE)
            {
                int index = GameManager.Instance.PlayerGPS.
                   CurrentRegionIndex;
                long date = DaggerfallUnity.Instance.WorldTime.
                   DaggerfallDateTime.ToClassicDaggerfallTime() +
                   ConversionTime;

                bankstruct[index].BankDepositDate = date;
                bankstruct[index].BonusRewarded = false;


               DaggerfallUI.AddHUDText(MessageHandler(
                  DepositDaysNumber == 1 ? MessageState.
                  DEPOSIT_ONE_DAY : MessageState.DEPOSIT   ,
                  DepositDaysNumber), MessageDelay);


            }
            else
            {
                Debug.Log(MessageHandler(MessageState.
                   FAILED_DEPOSIT));
            }

        }


        private void RewardBonusDeposit()
        {

            int index = GameManager.Instance.PlayerGPS.
               CurrentRegionIndex;
            long CurrentDate = DaggerfallUnity.Instance.WorldTime.
               DaggerfallDateTime.ToClassicDaggerfallTime() +
               ConversionTime;
            if (DaggerfallBankManager.BankAccounts[index].
               accountGold != 0 && CurrentDate >= bankstruct[index
               ].BankDepositDate + DepositDaysDue && bankstruct[
                index].BonusRewarded == false && bankstruct[index].
                BankDepositDate != 0)
            {
                int BonusGold = CalculateBonusRate(index);
                DaggerfallBankManager.BankAccounts[index].
                   accountGold += BonusGold;
                DaggerfallUI.AddHUDText(MessageHandler(
                   MessageState.REWARD, BonusGold), MessageDelay);
                bankstruct[index].BonusRewarded = true;

            }

        }
```

Here is the option for depositing without resetting the deposit date automatic. The SetDepositTimer method is executed every time the player makes a deposit. The reward method is checked every frame in the Update method. The code works as follow: The player makes a deposit and his position is saved, as well as the date, converted in the daggerfall time system. **The index is used to get the bank in the current region that the player is, this is the linking key between the BankStruct and the bank structure in the original game.** After saving, a message will be shown to the player, here we have two options, if the deposit is for one day it will show other message than the one for multiple days. The reward method checks if the player can get his interest and if the conditions are met, the bonus gold will be calculated using a special function. At the end the BonusRewarded is reset in order for the player to make another deposit manually.

Listing 5: Depositing Automatic

```
        private void AUTOSetDepositTimer(TransactionType type,
           TransactionResult result, int amount)
        {

            if (result == TransactionResult.NONE)
            {
                int index = GameManager.Instance.PlayerGPS.
                   CurrentRegionIndex;

                if (bankstruct[index].BankDepositDate == 0)
                {
                    long date = DaggerfallUnity.Instance.WorldTime
                       .DaggerfallDateTime.ToClassicDaggerfallTime
                       () + ConversionTime;
                    bankstruct[index].BankDepositDate = date;
                    DaggerfallUI.AddHUDText(MessageHandler(
                       MessageState.DEPOSIT, DepositDaysNumber),
                       MessageDelay);


                }

            }
            else
            {
                Debug.Log(MessageHandler(MessageState.
                   FAILED_DEPOSIT));
            }

        }

        private void AUTORewardBonusDeposit()
        {

            int index = GameManager.Instance.PlayerGPS.
               CurrentRegionIndex;
            long CurrentDate = DaggerfallUnity.Instance.WorldTime.
               DaggerfallDateTime.ToClassicDaggerfallTime() +
               ConversionTime;
            long DateRewardDeposit = bankstruct[index].
               BankDepositDate + DepositDaysDue - bankstruct[index
               ].RemainedDays;
        if (CurrentDate >= DateRewardDeposit && bankstruct[index].
           BankDepositDate != 0)
            {

            long DaysPassedFromLastPayout = ((CurrentDate -
               bankstruct[index].BankDepositDate) + bankstruct[
               index].RemainedDays);
            int initialGold = DaggerfallBankManager.BankAccounts[
               index].accountGold;

            for (int i = 1; i <= DaysPassedFromLastPayout /
               DepositDaysDue; i++)
                    DaggerfallBankManager.BankAccounts[index].
                       accountGold += CalculateBonusRate(index);

            if(DaggerfallBankManager.BankAccounts[index].
               accountGold == 0)
                DaggerfallUI.AddHUDText(MessageHandler(
                   MessageState.MISSED_DEPOSIT), MessageDelay);
            else
                DaggerfallUI.AddHUDText(MessageHandler(
                   MessageState.REWARD, DaggerfallBankManager.
                   BankAccounts[index].accountGold - initialGold),
                    MessageDelay);

            bankstruct[index].BankDepositDate = CurrentDate;
            if (DaysPassedFromLastPayout >= DepositDaysDue)
                bankstruct[index].RemainedDays = (
                   DaysPassedFromLastPayout % DepositDaysDue);
            else bankstruct[index].RemainedDays = 0;
            }
        }
```

The differences between the first method and the one form manual depositing are: now we don't reset the date if the player makes a deposit, and we no longer need the BonusRewarded flag. The second method is the one which changed more. In the DateRewardDeposit we save the date that the interest will be rewarded and subtract the days that remained from the last payout. **Let's just say that we set the interest time to 10 days, if we travel in other regions for 34 days, when we return to the region with the deposit we expect to have 3 payouts and 6 days for the next deposit, this is why we subtract the remained days.** The for loop will ensure that all the payouts are given. If the player takes all the money from the bank the time will continue to flow and he will be warned that he missed some payouts. At the end we check to see the remained days.

Listing 6: Bonus Rate equations

```
    private int CalculateBonusRate(int index)
    {

        if (BonusRateWithStats == false)
            return (int)((BonusRate / 100) * DaggerfallBankManager
               .BankAccounts[index].accountGold);
        else
        {
            PlayerEntity playerEntity = GameManager.Instance.
               PlayerEntity;
            float BonusRateWithStats = BonusRate * (
               BonusRateOffset + ((float)(playerEntity.Stats.
               PermanentPersonality * playerEntity.Stats.
               PermanentLuck * playerEntity.Skills.
               GetPermanentSkillValue(DaggerfallConnect.DFCareer.
               Skills.Mercantile)) / 1000000));
            Debug.Log(BonusRateWithStats);
            return (int)((BonusRateWithStats / 100) *
               DaggerfallBankManager.BankAccounts[index].
               accountGold);
        }
    }
```

This is the method which will handle all the payouts for every configuration that the player made at the beginning.

Listing 7: Message Handler

```
   public enum MessageState
   {
       DEPOSIT_ONE_DAY = 0,
       DEPOSIT = 1,
       REWARD = 2,
       FAILED_DEPOSIT = 3,
       FAILED_LOAD_SETTINGS = 4,
       MISSED_DEPOSIT = 5,
       GET_SAVE_ERROR = 6,
       LOAD_SAVE_ERROR = 7
   }
```

```
    private string MessageHandler(MessageState MessageCode, int MessageNumber = 0)
    {
        switch (MessageCode)
        {
            case MessageState.DEPOSIT_ONE_DAY:
                return "Bonus deposit gold in " + MessageNumber + " day";
            break;

            case MessageState.DEPOSIT:
                return "Bonus deposit gold in " + MessageNumber + " days";
            break;

            case MessageState.REWARD:
                return "Your Deposit Generated " + MessageNumber.ToString() + " Gold";
            break;

            case MessageState.FAILED_DEPOSIT:
                return "Failed Deposit";
            break;

            case MessageState.FAILED_LOAD_SETTINGS:
                return "Failed To Load Settings";
            break;

            case MessageState.MISSED_DEPOSIT:
                return "You Missed The Bonus Gold";
            break;

            case MessageState.GET_SAVE_ERROR:
                return "Failed To Get Saved Data";
            break;

            case MessageState.LOAD_SAVE_ERROR:
                return "Failed To Load Saved Data";
            break;

        }
        return "Wrong Message Code";
    }
```

Here we can find a simple message handler used to avoid code duplication, we have every message used with his code and the selection is made with a switch statement.

Listing 8: Main Functionality

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

             mod.LoadSettings();
             DepositDaysDue = DaggerfallDateTime.MinutesPerDay *
                DepositDaysNumber;
             initialLoanAmount = LoanAmount;

             DaggerfallWorkshop.Game.Serialization.SaveLoadManager
                .OnStartLoad += (SaveData_v1 saveData) =>
             {
                 HasLoadedData = false;
             };

             if (RestrictLoanOption == true)
             {

                  DaggerfallBankManager.OnBorrowLoan +=
                     RestrictLoaning;
                  DaggerfallBankManager.OnRepayLoan +=
                     EnableLoaning;

             }

            if (AutomaticDeposit == true)
            {
                DaggerfallBankManager.OnDepositGold +=
                   AUTOSetDepositTimer;
                DaggerfallBankManager.OnDepositLOC +=
                   AUTOSetDepositTimer;

            }
            else
            {
                DaggerfallBankManager.OnDepositGold +=
                   SetDepositTimer;
                DaggerfallBankManager.OnDepositLOC +=
                   SetDepositTimer;

            }
        }

        private void Update()
        {

             if (HasLoadedData == true)
             {

                 if (AutomaticDeposit == true)
```

```
                    {

                        AUTORewardBonusDeposit();

                    }
                    else
                    {

                        RewardBonusDeposit();

                    }

                     if (RestrictLoanOption == true)
                     {
                         if (HasLoan == true)
                             LoanAmount = 0;
                         else LoanAmount = initialLoanAmount;

                     }

                }

        }
```

In the Start method we load the settings the initialize the remaining variables. At the end we load the methods in the correct events. In Update we check the reward and loan methods only after the saving is done.

Listing 9: Saving Methods

```
        public Type SaveDataType
        {
            get { return typeof(BanksRemasteredSaveData); }
        }

        public object NewSaveData()
        {
            return new BanksRemasteredSaveData
            {

                bankstruct = new BankStruct[BankStructSize],
                HasLoan = false
            };
        }

    public object GetSaveData()
    {
        try
        {
            BanksRemasteredSaveData saveData = new
               BanksRemasteredSaveData
            {
                bankstruct = new BankStruct[BankStructSize]
            };

            for (int i = 0; i < bankstruct.Length; i++)
            {
                saveData.bankstruct[i] = bankstruct[i];
            }

            saveData.HasLoan = HasLoan;

            return saveData;
        }
        catch (Exception ex)
        {
            Debug.LogError(MessageState.GET_SAVE_ERROR);
            return null;
        }
    }


    public void RestoreSaveData(object saveData)
    {
        try
        {
            BanksRemasteredSaveData bankSaveData = (BanksRemasteredSaveData)saveData;
            bankstruct = new BankStruct[BankStructSize];

            for (int i = 0; i < bankSaveData.bankstruct.Length; i++)
            {
                bankstruct[i] = bankSaveData.bankstruct[i];
            }

            HasLoan = bankSaveData.HasLoan;

            HasLoadedData = true;
        }
        catch (Exception ex)
        {
            Debug.LogError(MessageHandler(MessageState.LOAD_SAVE_ERROR));
        }


    }
```

The saving logic is done pretty basic, don't forget that we need the IHasModSaveData interface in order to use the methods. Here we save all the BankStruct instances and the HasLoan flag. After restoring the data we set the HasLoadedData flag to true to ensure that the mod can start to work as expected.

And finally, the mod settings are found in the modsettings.json near the main script, I made a simple structure that can be customized without any problems.

7 Conclusions and Future Improvements

- Future features: An improvement could be to organize the code in several classes, the initial problem is that they are very interdependent and are complicate to separate properly, another problem is that I work with long variables that take up quite a lot of space, but unfortunately a conversion of the date from daggerfall to a small format would mean to perform mathematical splits and I would face data loss, decreasing the accuracy. As new items that can be added I will list them in the git page.

8 Contact Information

- Contact details: Email -> andreivoicea7@gmail.com
- Useful links: Nexus Mods