|  |
| --- |
| Design Document |
| Family Jewels |
| Peter Brill Jr.  Paul Fahey  **Revision 0.8**  **Tuesday, March 27, 2012**  **Revision 0.9 Friday, March 30, 2012**  **Revision 0.9.1 Wednesday, April 11, 2012**  **Revision 0.9.2 Thursday, April 12, 2012**  **Revision 0.9.3 Friday, April 13, 2012** |

Table of Contents

[Introduction 2](#_Toc322018632)

[Introduction to this Document 2](#_Toc322018633)

[Visual of User Interface 3](#_Toc322018634)

[User Commands 5](#_Toc322018635)

[Design of Features 5](#_Toc322018636)

[Feature One – Natural Economy 5](#_Toc322018637)

[Data Structures Used: 5](#_Toc322018638)

[Classes Used: 6](#_Toc322018639)

[Variables Used: 7](#_Toc322018640)

[Functions Used: 8](#_Toc322018641)

[Feature Two – Daily Fluctuation 9](#_Toc322018642)

[Data Structures Used: 9](#_Toc322018643)

[Classes Used: 9](#_Toc322018644)

[Variables Used: 9](#_Toc322018645)

[Functions Used: 9](#_Toc322018646)

[Feature Three – Win Condition 10](#_Toc322018647)

[Data Structures Used: 10](#_Toc322018648)

[Classes Used: 10](#_Toc322018649)

[Variables Used: 10](#_Toc322018650)

[Functions Used: 10](#_Toc322018651)

[Manifest Constants and Macros 11](#_Toc322018652)

[Global Data Map 11](#_Toc322018653)

[Economy Class Design 11](#_Toc322018654)

[Economy class Properties 12](#_Toc322018655)

[Economy class Methods 12](#_Toc322018656)

[Economy class Handlers 15](#_Toc322018657)

[Player Class Design 15](#_Toc322018658)

[Player class Properties 15](#_Toc322018659)

[Player class Methods 16](#_Toc322018660)

[Player class Handlers 19](#_Toc322018661)

[Application Design 19](#_Toc322018662)

[Organization of Code 22](#_Toc322018663)

[Other Design Considerations 23](#_Toc322018664)

# Introduction

This project is primarily focused on the development and release of the application ‘Family Jewels’. Family Jewels (Known herein as FJ), will be a simple economy and buy/trade simulator, borrowing directly from such works as ‘Drug Wars’ or the simple economy systems found in many RPG and Puzzle games. Simply put, the only real strategy for FJ is to buy low and sell high until a win condition is met. Above all else our focus with FJ is to make it fun and rewarding to the player; attempting to be at times both challenging and fun.

# Introduction to this Document

This document is to serve as a focus and planning tool for the design and implementation of the FJ application. The document will go through the interface design and feature design, as well as a list of user appropriate information. This document is to be considered the blueprint for all the things we want to include in the FJ project.

# Visual of User Interface

We’ve decided that a interface based solely on the command line would be more than enough to allow the user to interact with the application. Everything will be displayed primarily on one main screen as follows, with small decision trees branching out for different inputs before repainting the screen with each new day, purchase, or sale.

Family Jewels v1 – Day 1 – Your Cash: 80

You Have:

Garnet – 1, Sapphire – 0, Emerald – 0, Ruby – 0, Diamond - 1

===Jewel – Market Price===

Garnet – 10

Sapphire – 25

Emerald – 50

Ruby – 80

Diamond – 100

======================  
1. New Day  
2. Sell

3. Buy

4. Quit

|  |
| --- |
| ====================== 1. New Day 2. Sell  3. Buy  4. Quit  -----------------------------------------  2  What would you like to Sell? ===Instructions===  1. Garnet Enter 1 for Garnet  2. Sapphire Enter 2 for Sapphire  3. Emerald Enter 3 for Emerald  4. Ruby Enter 4 for Ruby  5. Diamond Enter 5 for Diamond  ----------------------------------------  5  How many Diamonds would you like to Sell?  1 |

Family Jewels v1 – Day 1 – Your Cash: 180

You Have:

Garnet – 1, Sapphire – 0, Emerald – 0, Ruby – 0, Diamond - 0

===Jewel – Market Price=== ===Instructions===

Garnet – 10 Enter **1** for New Day

Sapphire – 25 Enter **2** to Sell

Emerald – 50 Enter **3** to Buy

Ruby – 80 Enter **4** to Quit

Diamond – 100

======================  
1. New Day  
2. Sell

3. Buy

4. Quit

--------------------

# User Commands

For interaction with the main part of the interface, the user is given four simple commands that lead in most cases to another prompt requiring more information. The simple function of each command is broken down in the following table:

|  |  |
| --- | --- |
| 1: ‘New Day’ | Calls the ‘new day‘ function for the player, rolling all five of the jewels on the market to new values and incrementing the players day count. Also checks to see if the win condition is met. |
| 2: ‘Sell’ | Calls the ‘sell’ function after asking the player for an amount and the number of the jewel they wish to sell to the market. |
| 3: ‘Buy’ | Calls the ‘buy’ function after asking the player for an amount and the number of the jewel they wish to buy from the market. |
| 4: ‘Quit’ | Calls the ‘quit’ function and asks the user if they are sure before exiting the program |

# Design of Features

Concerning the planned features we have detailed the design following each features listing as it was written in the Product Requirements.

## Feature One – Natural Economy

1. The game endeavors to **simulate a natural economy**.
2. The games primary strategy will be one of supply and demand, with the primary strategy for the game being to simply ‘buy low’ and ‘sell high’
3. The use will start with 40 Dollars, enough to buy potentially anything in the game given planned daily fluctuations, spare of course Diamond

### Data Structures Used:

A **single queue** will be used with array implementation. See [Feature Two](#_Feature_Two_–) for details.

### Classes Used:

There will be an **Economy** class and a **Player** class to deal with the earnings of the player and all interaction with the economy.

**Economy Class:**

* One instance of this class is required
* The class must have a public interface to interact with the main application

**Player Class:**

* One instance of this class is required
* This class must have a public interface to store and return variables to the main application as well as the **Economy** class

### Variables Used:

**Economy Class:**

The economy class is tasked with keeping track of and manipulating the market as a whole. The current prices of each individual item must be tracked as well as the base price of each item. The ‘Init’ variables will be local only and unchanging, but the daily values that fluctuate must be accessible by the **New Day** function, so the information can be updated as the screen is redrawn.

|  |  |  |  |
| --- | --- | --- | --- |
| **Local Variables** | | | |
| **Type** | **Name** | **Constant Value** | **Usage** |
| Integer | InitGarnet | 10 | This group of variables is used to track the base values of jewels before being manipulated |
| Integer | InitSapphire | 25 |
| Integer | InitEmerald | 50 |
| Integer | InitRuby | 80 |
| Integer | InitDiamond | 100 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Global Variables** | | | |
| **Type** | **Name** | **Value Range** | **Usage** |
| Integer | Garnet | (InitGarnet) +/- 50% | This group of variables is used to track the current market price of jewels |
| Integer | Sapphire | (InitSapphire) +/- 50% |
| Integer | Emerald | (InitEmerald) +/- 50% |
| Integer | Ruby | (InitRuby) +/- 50% |
| Integer | Diamond | (InitDiamond) +/- 50% |

**Player Class:**

The player class is tasked with keeping track of and manipulating player data as a whole. Its jobs are simple and its interfaces few.

|  |  |  |  |
| --- | --- | --- | --- |
| **Global Variables** | | | |
| **Type** | **Name** | **Value Range** | **Usage** |
| Integer | Garnets | Between 0 and 2 Billion | This group of variables is used to track the players inventory |
| Integer | Sapphires |
| Integer | Emeralds |
| Integer | Rubys |
| Integer | Diamonds |
| Integer | Cash | Between 0 and 2 Billion | Tracks the player’s liquid assets |
| Integer | Days | Between 1 and 2 Billion | Tracks the days elapsed |

### Functions Used:

#### Economy Class:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Argument** | **Usage** |
| Constructor | Economy | None | Initializes values + populates queue |
| Void | NewDay | None | Rolls new values for jewels and increments *Days* |
| Integer | Sell | Integer, Integer | Allows the player to sell to the economy – Returns profit |
| Integer | Buy | Integer, Integer | Allows the player to buy from the economy – Returns cost |
| Integer | Value | Integer | Returns the value of a gem indicated by argument |

#### Player Class:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Name** | **Argument** | **Usage** |
| Boolean | PlayerWinning | None | Checks to see if *Cash* is above 2000 |
| Boolean | HasCash | Integer | Checks to see if the player has sufficient funds |
| Boolean | HasJewels | Integer, Integer | Checks to see if the player has sufficient jewels |
| Void | RemoveJewels | Integer, Integer | Removes jewels from players inventory |
| Void | AddJewels | Integer, Integer | Add jewels to the players inventory |

## Feature Two – Daily Fluctuation

1. The economy will **fluctuate randomly** with each new day.
2. The game uses a system that changes the values of all gems to plus (or minus) fifty percent of their base value. These bounds are in place to keep the prices from sliding into the ‘entirely unrealistic’ realm, rather than ‘mostly inaccurate’.

### Data Structures Used:

Queue

* The implementation for this application only requires one queue
* The queue is specifically and only for the **New Day** function of the **Economy** class
* The queue will only be worked with directly within the **Economy** class and only needs to be accessed locally

### Classes Used:

The **Economy** class will be used for the daily fluctuation, detailed in the last [section.](#_Classes_Used:)

### Variables Used:

The **Economy** class will contain and manipulate the variables required for this future, detailed in the [previous section.](#_Variables_Used:)

### Functions Used:

The **NewDay** function of the **Economy** class is detailed in the [previous section](#_Functions_Used:).

## Feature Three – Win Condition

1. The game will feature a simple **win condition**.
2. The games win condition is checked during each new day, to see if it exceeds 2000 Dollars. If the win condition is met, the player will be greeted with competition text, and the game is over.

### Data Structures Used:

This section is not applicable.

### Classes Used:

The **Player** class is detailed in the [previous section](#_Classes_Used:) and will be primarily used to implement the win condition.

### Variables Used:

The **Player** class uses the *cash* variable to check to see if the player has reached the win condition. This is detailed in the [previous section](#_Classes_Used:).

### Functions Used:

The **PlayerWinning** function of the **Player** class will be used to check to see if the player has one. It will simply check the *cash* variable and compares it to 2000. If the player ever exceeds 2000 dollars then they will win the game. This function is detailed in a [previous section](#_Functions_Used:).

# Manifest Constants and Macros

The following variables will be constants, defined in the **Economy** class header and editable for ‘fine tuning’ for future versions of the program.

|  |  |  |  |
| --- | --- | --- | --- |
| **Economy** Class Constant Variables | | | |
| **Type** | **Name** | **Constant Value** | **Usage** |
| Integer | InitGarnet | 10 | This group of variables is used to track the base values of jewels before being manipulated |
| Integer | InitSapphire | 25 |
| Integer | InitEmerald | 50 |
| Integer | InitRuby | 80 |
| Integer | InitDiamond | 100 |
| **Queue** Class Constant Variables | | | |
| Integer | MAXSIZE | 5 | Sets the maximum size of the queue used in the **NewDay** function |

# Global Data Map

No critical or even important data is stored or available globally. Our application implements our one queue locally within the **Economy** class.

# Economy Class Design

The **Economy** class does not have any expressed relationships with any other class.

## Economy class Properties

The following section details all the data members for the **Economy** class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Economy** Class Local Variables | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | InitGarnet | 10 | This group of variables is used to track the base values of jewels before being manipulated |
| Integer | InitSapphire | 25 |
| Integer | InitEmerald | 50 |
| Integer | InitRuby | 80 |
| Integer | InitDiamond | 100 |
| **Economy** Class Global Variables | | | |
| Integer | Garnet | (InitGarnet) +/- 50% | This group of variables is used to track the current market price of jewels |
| Integer | Sapphire | (InitSapphire) +/- 50% |
| Integer | Emerald | (InitEmerald) +/- 50% |
| Integer | Ruby | (InitRuby) +/- 50% |
| Integer | Diamond | (InitDiamond) +/- 50% |

## Economy class Methods

Unless rearrangement is required due to the detail of the data, all class methods/member functions will be detailed in this way:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** | **Variables** |
| None | None | None | None | None |
| **Method Algorithm** | | | | |
| CALL Example  DECLARE *Num* AS Integer 10  Output “This is an example text box.”  Output “This is our example number ” & *Num* & “.”  Return | | | | |

**Economy** class member functions are listed in a [previous section](#_Functions_Used:).

|  |  |  |  |
| --- | --- | --- | --- |
| **Economy (Constructor) PUBLIC** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | n/a | n/a |
| **Economy (Constructor) Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | InitGarnet | 10 | This group of variables is used to track the base values of jewels before being manipulated |
| Integer | InitSapphire | 25 |
| Integer | InitEmerald | 50 |
| Integer | InitRuby | 80 |
| Integer | InitDiamond | 100 |
| Integer | Garnet | (InitGarnet) +/- 50% | This group of variables is used to track the current market price of jewels |
| Integer | Sapphire | (InitSapphire) +/- 50% |
| Integer | Emerald | (InitEmerald) +/- 50% |
| Integer | Ruby | (InitRuby) +/- 50% |
| Integer | Diamond | (InitDiamond) +/- 50% |
| **Economy (Constructor) Algorithm** | | | |
| **Economy** OF **Economy**  DECLARE Integer  *InitGarnet, Garnet* AS 10  *InitSapphire, Sapphire* AS 25  *InitEmerald, Emerald* AS 50  *InitRuby, Ruby* AS 80  *InitDiamond, Diamond* AS 100  DECLARE *Queue* AS **Queue**  CALL **Enqueue** OF *Queue* USING *initGarnet, initSapphire, initEmerald, initRuby, initDiamond*  END **Economy** OF **Economy** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Economy) NewDay** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | n/a | Void |
| **(Economy) NewDay Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | n/a |
| **(Economy) NewDay Algorithm** | | | |
| **Newday** OF **Economy**  SET *Garnet* TO *Queue.***Dequeue**  SET *Garnet* TO *Garnet* MULTIPLIED BY RANDOM 1.5 to 0.5  REPEAT FOR EACH GEM  CALL **Enqueue** OF *Queue* USING *initGarnet, initSapphire, initEmerald, initRuby, initDiamond*  INCREMENT **Player.***Days*  END **Newday** OF **Economy** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Economy) Sell** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer Type, Integer Amount | Void |
| **(Economy) Sell Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | To check for required gems |
| **(Economy) Sell Algorithm** | | | |
| **Sell** OF **Economy** USING *Type, Amount*AS INTEGER  DECLARE Integer *profit*  SET *profit* TO *Amount* MULTIPLIED BY **Value** OF **Economy** USING *Type*  IF **Player.HasJewels** USING *Type, Amount*IS TRUE THEN  SET **Player.***Cash* TO**Player.***Cash + profit*  **Player.**RemoveJewels USING *Type, Amount*  IF **IsWinning** OF **Player** IS TRUE THEN  OUTPUT ENDING SCREEN  ELSE Return  ELSE  OUTPUT “ERROR: You don’t have the required gems!”  RETURN  END **Sell** OF **Economy** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Economy) Buy** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer Type, Integer Amount | Void |
| **(Economy) Buy Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | *Cost* | n/a | To check the cost of the transaction |
| **(Economy) Buy Algorithm** | | | |
| **Buy** OF **Economy** USING *Type, Amount*AS INTEGER  DECLARE Integer *cost*  SET *cost* TO *Amount* MULTIPLIED BY **Value** OF **Economy** USING *Type*  IF **Player.HasCash** USING *cost* IS TRUE THEN  SET **Player.***Cash* TO**Player.***Cash – cost*  **Player.**AddJewels USING *Type, Amount*  ELSE  OUTPUT “ERROR: You don’t have the required cash!”  RETURN  END **Buy** OF **Economy** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Economy) Value** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer Type | Integer |
| **(Economy) Value Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | *Val* | n/a | To check the current value of the gem |
| **(Economy) Value Algorithm** | | | |
| **Buy** OF **Economy** USING *Type*AS INTEGER  SWITCH *Type*  IS 1 THEN RETURN *Garnet*  IS 2 THEN RETURN *Sapphire*  IS 3 THEN RETURN *Emerald*  IS 4 THEN RETURN *Ruby*  IS 5 THEN RETURN *Diamond*  ELSE THEN  OUTPUT “ERROR: Incorrect gem value requested!”  RETURN *2000*  END **Buy** OF **Economy** | | | |

## Economy class Handlers

Class handlers and class methods are described in the same way with no differentiation in this case. The **Economy** class methods are detailed in the [previous section](#_Economy_class_Methods).

# Player Class Design

The **Player** class does not have any expressed relationships with other classes.

## Player class Properties

The following details all the data within the **Player** class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Player** Class Global Variables | | | |
| **Type** | **Name** | **First Value** | **Usage** |
| Integer | Garnet | 0 | This group of variables is used to track the player inventory of jewels |
| Integer | Sapphire | 0 |
| Integer | Emerald | 0 |
| Integer | Ruby | 0 |
| Integer | Diamond | 0 |
| Integer | Cash | 0 | Cash tracking |
| Integer | Days | 1 | Day Tracking |

## Player class Methods

**Player** class member functions are listed in a [previous section](#_Functions_Used:).

|  |  |  |  |
| --- | --- | --- | --- |
| **(Player) PlayerWinning** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | n/a | Boolean |
| **(Player) PlayerWinning Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | *n/a* | n/a | n/a |
| **(Player) PlayerWinning Algorithm** | | | |
| **PlayerWinning** OF **Player**  IF *Cash* OF **Player** IS GREATER THAN 2000  OUTPUT “You have made over 2000 dollars trading jewels!”  RETURN TRUE  ELSE  RETURN FALSE  END **PlayerWinning** OF **Player** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Player) HasCash** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer *required* | Boolean |
| **(Player) HasCash Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | *Required* | n/a | The required value to check against |
| **(Economy) HasCash Algorithm** | | | |
| **HasCash** OF **Player**  IF *Cash* OF **Player** IS EQUAL TO OR GREATER THAN *required*  RETURN TRUE  ELSE  RETURN FALSE  END **HasCash** OF **Player** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Player) HasJewels** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer *type ,*Integer *amount* | Boolean |
| **(Player) HasJewels Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | *n/a* | n/a | n/a |
| **(Economy) HasJewels Algorithm** | | | |
| **HasJewels** OF **Player**  SWITCH *type*  IS 1 THEN  IF **Player.***Garnets* IS EQUAL TO OR GREATER THAN *amount*  RETURN TRUE  ELSE RETURN FALSE  IS 2 THEN  IF **Player.***Sapphires* IS EQUAL TO OR GREATER THAN *amount*  RETURN TRUE  ELSE RETURN FALSE  IS 3 THEN  IF **Player.***Emeralds* IS EQUAL TO OR GREATER THAN *amount*  RETURN TRUE  ELSE RETURN FALSE  IF 4 THEN  IF **Player.***Rubys* IS EQUAL TO OR GREATER THAN *amount*  RETURN TRUE  ELSE RETURN FALSE  IF 5 THEN  IF **Player.***Diamond* IS EQUAL TO OR GREATER THAN *amount*  RETURN TRUE  ELSE RETURN FALSE  ELSE  OUTPUT “You do not have enough jewels!”  RETURN FALSE  END **HasJewels** OF **Player** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Player) RemoveJewels** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer *type ,*Integer *amount* | Void |
| **(Player) RemoveJewels Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | n/a |
| **(Player) RemoveJewels Algorithm** | | | |
| **RemoveJewels** OF **Player**  SWITCH *type*  IS 1 THEN RETURN *Garnet*  SET **Player.***Garnets TO* **Player.***Garnets - amount*  IS 2 THEN RETURN *Sapphire*  SET **Player.***Sapphires TO* **Player.***Sapphires - amount*  IS 3 THEN RETURN *Emerald*  SET **Player.***Emeralds TO* **Player.***Emeralds- amount*  IS 4 THEN  SET **Player.***Rubys TO* **Player.***Rubys - amount*  IS 5 THEN  SET **Player.***Diamonds TO* **Player.***Diamonds – amount*  ELSE  OUTPUT “Incorrect Jewel Removal Input!”  RETURN  END **RemoveJewels** OF **Player** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **(Player) AddJewels** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Integer *type ,*Integer *amount* | Void |
| **(Player) AddJewels Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | n/a |
| **(Economy) AddJewels Algorithm** | | | |
| **AddJewels** OF **Player**  SWITCH *type*  IS 1 THEN RETURN *Garnet*  SET **Player.***Garnets TO* **Player.***Garnets + amount*  IS 2 THEN RETURN *Sapphire*  SET **Player.***Sapphires TO* **Player.***Sapphires + amount*  IS 3 THEN RETURN *Emerald*  SET **Player.***Emeralds TO* **Player.***Emeralds + amount*  IS 4 THEN  SET **Player.***Rubys TO* **Player.***Rubys + amount*  IS 5 THEN  SET **Player.***Diamonds TO* **Player.***Diamonds + amount*  ELSE  OUTPUT “Incorrect Jewel Addition Input!”  RETURN  END **AddJewels** OF **Player** | | | |

## Player class Handlers

Class handlers and class methods are described in the same way with no differentiation in this case. The **Player** class methods are detailed in the [previous section](#_Player_class_Methods).

# Application Design

The application will create the user interface of the game and assumes all functions and variables are working as intended in the economy and player classes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Draw** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Void | Void |
| **Draw Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | n/a |
| **Draw Algorithm** | | | |
| **Draw**  Clear Screen  Display header information (Game Title, Version, Day, Player Cash)  Display Gems with updated values  Display Gems with updated prices  Display instructions lined up with gem prices  Display player options  END **Draw** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Quit** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Void | Void |
| **Quit Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| n/a | n/a | n/a | n/a |
| **Quit Algorithm** | | | |
| **Quit**  Clear Screen  Display Quit message  END **Quit** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Main** | | | |
| **Pre** | **Post** | **Inputs** | **Outputs** |
| n/a | n/a | Void | Void |
| **Main Variables** | | | |
| **Type** | **Name** | **Value** | **Usage** |
| Integer | gemType | user | To represent the value of a gem for buying and selling |
| Integer | Amount | user | To represent the amount of a gem a player wants to buy or sell |
| Integer | Selection | User | To represent the users menu choice |

**Main Algorithm**

**Main**

* Create *gemType* as type integer
* Create *amount* as type integer
* Create *selection* as type integer
* Create *economy* as **Economy**
* Create *player* as **Player**
* Draw Screen
* Take user option input (Selection)

While (True)

Draw Screen

SWITCH *selection*

IS 1 THEN

* Have *economy* call **New Day**
* CONTINUE

IS 2 THEN

While (True)

Prompt user input for gem to sell

Take user gem input (*gemType*)

If (*gemType*! = 1,2,3,4, or 5)

* Display error message
* CONTINUE

Prompt user input for amount of gem to sell

Take user amount input (*amount*)

Have *economy* call **Sell**

Pass in *selection* & *amount*

BREAK

END WHILE

IS 3 THEN

While (True)

Prompt user input for gem to buy

Take user gem input (*gemType*)

If (*gemType* ! = 1,2,3,4, or 5)

* Display error message
* CONTINUE

Prompt user input for amount of gem to sell

Take user amount input (*amount*)

Have *economy* call **Buy**

Pass in *selection* & *amount*

BREAK

END WHILE

IS 4 THEN

Have application call **Quit**

ELSE

OUTPUT “Incorrect Selection Input!”

CONTINUE

END WHILE

END **Main**

# Organization of Code

This section will list the files needed to compile this program properly and keep the code organized.

|  |  |
| --- | --- |
| **File** | **Description** |
| Economy Class Header (.h) | This file will contain prototypes for the data members and member functions of the economy class. |
| Economy Class Body (.cpp) | This file will contain the data member and member function definitions of the economy class. |
| Player Class Header (.h) | This file will contain prototypes for the data members and member functions of the player class. |
| Player Class Body (.cpp) | This file will contain the data member and member function definition of the player class. |
| Miscellaneous Header (.h) | This file will contain prototypes for miscellaneous functions that don't pertain to the economy and player class. |
| Miscellaneous Body (.cpp) | This file will contain definitions for miscellaneous functions that don't pertain to the economy and player class. |
| Application (.cpp) | This file will contain the main function of the program that will make the necessary calls to the player and economy classes along with calls to miscellaneous functions. This file will require an include of the economy and miscellaneous header files to function properly. |

# Other Design Considerations

This section will briefly go over other design considerations that were thought of in the development of this version of Family Jewels. These may end up being ideas we might use for future versions of the game.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Stack | In a new version of the game we would like to experiment with a stack data structure instead of the current queue structure to feed the gems into the economy. |
| Java Version | This design was representing a C++ version of Family Jewels in the command line. We would like to create a future version of the game using the java programming language with a GUI for the user and make the necessary tweaks in the design to complement java features. |