Product Requirements

***Family Jewels***

**Peter ‘The Brick’ Brill Jr.**

**Paul ‘Bird-With-Arms’ Fahey**

**Revision 1**

**Friday, December 16, 2011**

Table of Contents

[Introduction 3](#_Toc311787862)

[Product Overview 3](#_Toc311787863)

[Product Functional Requirements 4](#_Toc311787864)

[Product Non-Functional Requirements 4](#_Toc311787865)

[Usability Requirements 5](#_Toc311787866)

[Reliability Requirements 5](#_Toc311787867)

# Introduction

The *Family Jewels* application is a simple game program that aims to be fun, expandable, and slightly sarcastic. This document will describe briefly some of the intended specifications for *Family Jewels* as well as serve as a ‘measuring stick’ to compare against any iterations of the application.

# Product Overview

The *Family Jewels* game is a simple way to play with an extremely simplified economy. The basic premise starts from the fact that you have entered the precious stone trade with a finite amount of money. The price of each jewel in the market tends to be very different, a garnet costing exponentially less than a diamond. Indeed, the prices in the market also fluctuate randomly like they would in a realistic economic system.

The game starts with that premise, and relies on another finite number for its end condition: You are attempting to trade back and forth until you have earned enough money to purchase heirloom jewels. Once these heirloom jewels are purchased, you can pass them down to your poor, jewel-less family and help take care of them for generations to come.

# Product Functional Requirements

These are the functional requirements we’ve had in mind for the project since our initial brainstorming session, and are considered non-negotiable.

1. The *Family Jewels* player will start with a sum enough to purchase low-priced gemstones only.
2. The player will eventually be capable of purchasing everything in the global market, if they play like a smart businessman.
3. Buying low and selling high is the primary strategy for the game.
4. Prices will fluctuate randomly, but within bounds that keeps silly things like ‘One dollar diamonds’ ending up in the market. The boundary therein will be 50% more or less in price day to day from the base price.

# Product Non-Functional Requirements

These are some of the requirements that outline pitfalls we wish to avoid concerning the end-user experience.

1. Situations where the player starts and cannot get any foothold no matter how well they play will be avoided: The starting cash must be carefully balanced to be capable of buying low level gemstones even at their highest fluctuation: Although this is not economically sound, bad luck could keep the player in this situation for some time.
2. ‘One dollar diamonds’ are to be avoided at all costs, and even though the value of each gemstone is not realistic to the real world, the values of gemstones to one another should be consistent.
   1. A diamond will never be worth less than a garnet.
3. The game should not be considered by anyone with casual gaming experience to be ‘difficult to play’.
4. The game should generally not be considered by anyone with casual gaming experience to be ‘without some significant challenges’.
5. The game will not crash, damage, corrupt or otherwise maliciously impact the user’s computer, and any crashing/damage/corruption occurring during the use of this application should be entirely coincidental.

# Usability Requirements

Concerning usability, the program fits very nicely into a few points.

1. It is for anyone capable of reading and responding to generated prompts accordingly; most likely between the ages of 5 to that of moderate senility.
2. The program should be easy to install, run, and to remove on any Windows XP, Windows Vista, or Windows 7 capable computer.
3. The program should be usable on any size monitor capable of correctly rendering a command prompt window.

# Reliability Requirements

We generally would expect that the program will nearly never fail, and if it does indeed fail it will be no fault of the design or code in question. Upper limits will be placed on many variables to ensure that if somehow the user does manage to push the application to the breaking point, it will recognize this and gracefully either correct itself, or close in a neat and orderly fashion: Ideally explaining what exception occurred to the user.