A Particle Interaction Simulator Developed Using Functional Programing Techniques

Hugo Robinson

Contents

Analysis 1

Background 1

Smaller Than Indivisible? 1

You Must Be Joking Mr Feynman 1

The Standard Model 2

Particles and Anti-particles 3

When Small Things go Bang 3

A Small Side Note on the Creation of the Universe 3

Description of project 3

Inside a Simulation 3

Particles from Top to Bottom 3

Why Functional? 4

You Must be Joking Mr McCarthy 4

A Cup of Coffee? 4

Challenges - Imagine Programing Without, Variables, Loops, Selection or Sequence 4

What is the Point? 5

Approach 5

External stakeholder 5

Notes From Discussion 5

Specific objectives 6

Must have 6

Should have 6

Could have 6

Won’t have 6

Design 7

Overall design approach 7

Plan for final development structure 7

The Class System 7

Functional Model (Done collisions functions, not vector functions) 8

Like an Onion 8

Why C# and not Haskell 9

Specific problems and their solutions 9

Heads and tails 9

How to Randomise a Determined Value 11

One of these is Not Like the Others 14

Areas involving technical complexity 14

Selection 14

Nested Selection 15

Recursion 16

Higher Order Functions 16

Tuples 19

How to do Functional in C# 19

Inheritance 19

Generics 21

Technical implementation 23

Testing 32

Test Plan and Inbuilt Security 32

High Level Test Plan (Outside in) 33

Low Level Test Plan (Inside out) 41

Collision Functions 41

Vector Functions 62

Summery Of Testing 65

Evaluation 67

Appendix 1 1

# Analysis

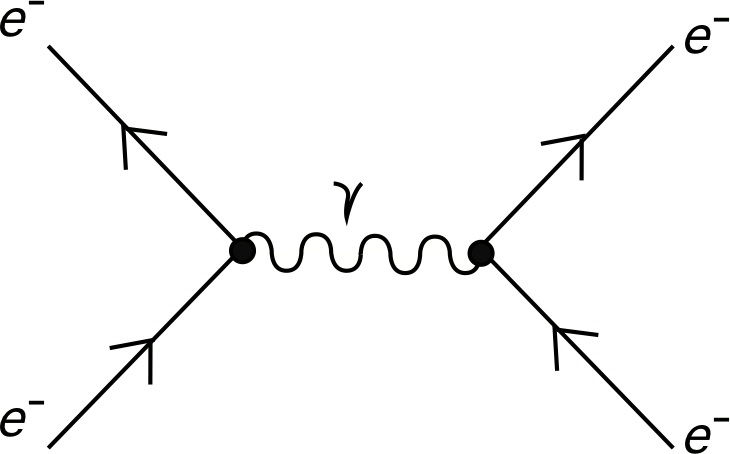
## Background

### Smaller Than Indivisible?

The world of physics can seem big and complex, and quite frankly it is. However everything form the smallest amoeba to the largest Galaxies all work on the same principles, the interactions between particles. Particle physics, as we know it today was first discovered between 1879 and 1897 when the electron was discovered. This meant that the atom was not indivisible but was comprised of many smaller particles. The discovery led Joseph Thompson to release his Plum Pudding representation of an atom where a dense cloud of positive charge was interlaced with electrons. However as we know today, this model is not correct, the new model with a nucleus of positive charge was proposed Ernest Rutherford in 1919 after his experiment which involved firing alpha particles at a very thin sheet of gold foil. Most of the particles passed through the foil as expected however some diffracted and even fewer bounced back. This was so astonishing at the time that Ernest Rutherford said, “It was quite the most incredible event that has ever happened to me in my life. It was almost as incredible as if you fired a 15-inch shell at a piece of tissue paper and it came back and hit you” <https://www.goodreads.com/quotes/105980-it-was-quite-the-most-incredible-event-that-has-ever>. Then in 1931 James Chadwick discovers the neutron, which gives the final piece of the puzzle for the creation of the atom and how the standard model of the atom is represented today. As a result of these studies the four fundamental forces of nature were discovered, The Strong Force, The Weak Force, Electromagnetism and Gravity. The Strong force mediates how the nucleons stay together and acts over a range of femptometers. The weak force mediates the decay of particles. The electromagnetic force mediates how different charges and magnetisms interact with each other, and gravity is the reason matter can clump together to form galaxies, stars and planets.

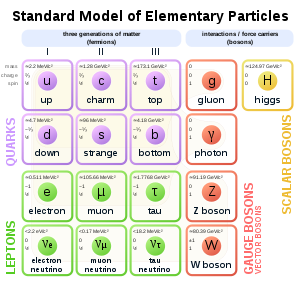
### You Must Be Joking Mr Feynman

Along with this, throughout the last century, different interactions between particles have been observed, both natural and created in colliders. These interactions govern everything from the levitation of superconductors <http://www.superconductors.org/> to the entire field of Chemistry. Many of these interactions can be represented through Feynman diagrams.



This Feynman Diagram shows a basic electrostatic repulsion where two electrons that poses the same charge come too close together. Then an interaction particle, in this case a photon, causes the two electrons to move away from each other. This diagram is also represented on a graph where the x-axis is space and the y-axis is time.

### The Standard Model



The Standard Model shows all known Elementary particles that we know of. The two most recognisable are the electron and the photon. One of the more interesting particles is the Higgs Boson that gives everything mass. This particle is not model in my project however as very little is known about it, by anyone.

### Particles and Anti-particles

Everything we see around us is matter, from the tables I am working on, to the chair I am sitting on, to the snow that is drifting gently outside. All of it is created from particles; however not all matter in the universe is created from the same particles, anti-matter also exists and it is made from anti-particles. These anti-particles are in every way that same as their counterparts however they have an opposite charge. So the standard particle an electron has a charge of -1, its anti-particle, the positron has a charge of +1 but they both have the same rest mass (9.10956 x 10 -31 Kg).

### When Small Things go Bang

One of the simplest interactions that is found in nature is the annihilation. This is when a particle and its antiparticle collide to create two photons of equal energy and opposite directions. Dependent on the velocity of the two particles the energy of the photons will be different, but the two photons will always have the same energy as each other. This is calculated by using Einstein famous equation E=mc2 (E = Energy, m = Rest Mass, c = The speed of light) and the equation Ke = 0.5mv2 (Ke = Kinetic Energy, m = Rest Mass, v = Velocity). The addition of E and Ke gives the total energy for that particle. This energy is then halved and as photos have a set speed (3x108 m/s) the energy difference takes the form of a higher frequency for more energy and a lower frequency for less. This allows photons to be created in any part of the electro-magnetic spectrum, from radio to gamma.

### A Small Side Note on the Creation of the Universe

In the beginning, or so we think, the Universe was a very small, very hot, very dense particle, filled with all of the energy ever to exist. Then, all of a sudden, it expanded, rapidly. Within the first few femtoseconds the universe called enough for Quarks and Leptons to form. These particles are the building blocks of all matter, however at this time it was still too hot to form atoms. At round 0.1 milliseconds after the Big Bang the universe has cooled to about 1012K where atoms start to form out of the sea of quarks. Also the discrepancy between matter and anti-matter happens now as the two start to annihilate each other. This discrepancy means that everything we see today is matter but however it is only 1 billionth of the matter created in the Big bang. At approximately one hundred seconds the Universe is 109K, cool enough for fusion to take place, creating the first few light nuclei such as helium. This creation process shows how the particles we see today were made and how they interact. When the Universe was too hot the leptons and quarks had too much energy and could not form larger particles. Once the energy dissipated larger particles could be formed and the different forces came into being that then allowed not only the creation of larger atoms, but of whole galaxies.

## Description of project

### Inside a Simulation

This project is designed to simulate particle interactions in multiple scenarios where up to two particles are entered by the user, along with some requirements such as velocity. The program will then use this information to run real calculations to determine the velocity and position of new particles created. This information will then be presented to the user.

### Particles from Top to Bottom

Particle classification is a great example of a hierarchical structure where each individual particle is represented by a group classification. All particles re split into three groups; Hadrons, Leptons and Quarks. Leptons and Quarks are also the fundamental particles as they are not made of anything. Hadrons are create out of Quarks, however the Hadron class splits into two parts, the Baryons and the Mesons. The difference between the two are that Baryons are made up of three Quarks and Mesons are created from a Quark and an Anti-Quark. For how this structure is broken down see Appendix 1

## Why Functional?

### You Must be Joking Mr McCarthy

In 1958, John McCarthy was leading a team at MIT to ‘create a system for programming computations over symbolic data’ <https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf> - Chapter 2 Lisp - Page 5. Through his work the language Lisp was create, which took many of the ideas from Lambda Calculus, however still used some programing conventions such as variables. Programing in Haskell Second Edition – By Graham Hutton – 1.4 - Page 8 . The first true functional language was created in the 1960’s with the development of ISWIM by Peter Landin, this showed that languages do not need variable assignment to work. In 1987 the idea of Haskell was born as a modern, revolutionary language. Throughout the 1990’s type classes and monads were developed by Philip Wadler which were two areas that set Haskell apart from other langue’s. In 1999 the language was ready and the first versions, Haskell 98, was released<https://en.wikipedia.org/wiki/Haskell_(programming_language)>.

### A Cup of Coffee?

The analogy used here was used by Luca Bolognese in a talk about Microsoft’s F# language in 2009. He compares the execution of statements and expressions to making a cup of coffee. In a procedural language the program is written as a sequence of commands, each command dictates what to do next to reach the end result. This can involve calculations or the modification of objects. To bring it back to the coffee analogy, to obtain a coffee with two sugars in the program would first create the coffee object and then add two sugars to it to reach the final result. However Functional languages are written using the evaluation of expressions, these expressions dictate the properties of the object instead of creating it first with standard properties, this means the program will not create a coffee and then add the sugar but will instead create a coffee with sugar already in. This means we cannot use the incorrect form of the object as there is only one form, or alternatively wee cannot drink the coffee without sugar as it is already created with sugar in. This subtle difference in how the coffee is created has some large effects on the approach on how to write the code to do so. Going functional can pose some serious challenges, but also bestows some excellent rewards. Real World Functional programming- By Tomas Petricek and Jon Skeet- Page 5

### Challenges - Imagine Programing Without, Variables, Loops, Selection or Sequence

There are some very serious challenges that arise when trying to create a program using functional techniques. One is the lack of variables used. Right from the beginning of being taught to program variables have been used, however they are not allowed in functional as variables can change. Whereas in functional all types have to be immutable. Therefore if any data structure needs to be changed, a new data structure has to be created with exactly the same entities as the previous one and whatever change has taken place also applied. The lack of sequence also makes functional difficult as everything has to be done using only one line within a function, this reduces some of the clarity when coding and requires simple sequential statements to either be moved into one line, or extracted into separate functions. There is also no selection in functional, therefore ternary operators must be used on the return statement instead. Any sort of loop is also not allowed in functional programing; therefore recursion must be used throughout.

### What is the Point?

The Functional Paradigm is not new, but until recently it has not be used outside academia however, with todays requirements of big data it has become much more prevalent. It is used heavily in science as functional techniques due to its reliability when headlining large volumes of data; it also can be very easily tested to make sure the results that are output are correct and there are no logic errors. Having learnt some basics of functional programing through the course I decided I wanted to peruse it further, extending myself past the trivial problems introduced to us. Therefore with my interest in the topic and its evident use in science it was perfect for me to develop me project around.

There are multiple known benefits to going functional, especially when using a true functional language. As stated above they are very reliable due to the lack of side effects, simply put if a function is calling two others, and it is known that the two functions work how they are meant to then the first function will also work how it is meant to and the same, correct answer will always come out. Functional code is very terse as all of the function implementations are only one line long, therefore the code base is very small, yet it has a high amount of complexity to it so a lot can be done with not much code. As Mark Twain once said “I didn’t have time to write you a short letter, so I write you a long one” Another major benefit to functional is parallelisation, due to all of the functions being able to work on their own the code can be split up and run of different processors with the results being brought together at the end. Therefore the code can be run much faster as it can run multiple paths at the same time, making it very useful for working with large data sets such as in data analytics Real World Functional Programing – By Tomas Petricek and Jon Skeet – 1.4.3 Page20. Pure functional languages such as Haskell can use referential transparency, this is when the program can cache the result of a function call and replace that function with its answer as long as it has the same parameters. Therefore when calling a function multiple times on the same line to use the same value in different ways, the program does not need to follow the function call each time, instead just producing the answer making the program much more efficient than it appears <https://wiki.haskell.org/Referential_transparency>.

## Approach

### External stakeholder

My approach is to present the idea of what I am trying to do to a physics teacher who will become the stakeholder in my project. I will then meet with them to gain ideas for added interactions to my project. This approach will allow me to develop my program in line with the requirements of the stakeholder. The requirements of the stakeholder will be reasonably restricted by the functional nature of the program.

### Notes From Discussion

These notes are not a full transcript from my discussion with the stakeholder but are condensed notes of the major points brought up.

* Need to have the basic interactions taught in A-level Physics such as Pair Production, Beta+ and Beta- decays and Electrostatic Repulsion
* Would like to see apparatus simulated such as a cyclotron and velocity selector using the required formula to work out velocities
* If a GUI was to be implemented a visual representation of the interactions would be required (I responded with the fact that GUI was not possible as the program I functional)

## Specific objectives

### Must have

My main objective for this project was to implement the core if the program using functional programing techniques.

The program should allow a user to input what interactions they would like to simulate and the program will the return the list of particles created and some necessary properties of those particles (eg velocity)

A range of particles implemented as objects with their own set properties that the rest of the program can call upon.

The stakeholder has provided some requirements for simulations he would like to see and these need to be met

* Beta+ and Beta- Decays
* Electrostatic repulsion
* Pair Production
* Cyclotron
* Velocity Selector
* Vectors for particles ejected

### Should have

Full list of elements that can be called upon when an element is created

Full hierarchy of different sub atomic particles.

### Could have

To create long decay chains using the decay graph for heavy, unstable elements such as uranium until they become stable.

Decides if an element created is stable or unstable and then gives the option for the element to undergo its decay process.

### Won’t have

Bubble chamber GUI due to functional

Large Hadron Collider GUI due to functional style and processing demands being too high and complex.

# Design

## Overall design approach

The original plan – GUI + Functional mideset

Function Model

### Plan for final development structure

Layer 1: Particle

* Where the particle diagram is used and that relation is coded into the system. All particles are also provided with a full array propitiates and methods they may require (eg Mass to Energy)
* This is the first layer to be implemented as it will allow the particles to be used throughout the development of the rest of the program with ease
* Uses separate program to the collision, vectors and UI programs
* Start with the Quarks and then move onto the bigger particles

Layer 2: Collision

* Where all the calculations for the collisions takes place and outputs what particles are created
* Will need different collisions for different types (eg Electron capture, annihilation, Proton-proton)
* Should not require any of the vectors layer in coding

Layer 3: Vector

* Where the vector calculations and positioning takes place
* Calls upon both the particles and collisions layer to determine weather any new collisions take place after the primary collision

Layer 4: UI

* The terminal screen UI should be developed alongside the rest of the program but should not be integrated into it to allow a possible development of a graphical UI at a later date, however this step will not be included in this project.
* This section will not be following the functional paradigm as it is very challenging to develop a UI as “A functional program defines a pure function, with no side effects, however the whole point of running a program is to have some side effect” – https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/ - Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell – Slide 8

This plan keeps allows for a higher layer to call on a lower layer but not the other way, this is to make sure that further down the development of the program, if a layer needs to be radically changed it should be easier as it is not being referenced by every other part of the program, only the layers above it.

### The Class System

The classification of particles allows for me to easily implement a class hierarchy system, which will form the base for every interaction in the project, as these classes will represent all of the particles with their attributes being stored as properties. These classes can then be created at individual instances multiple times and still retain their basic properties such as charge and rest mass. However their position and velocity can still be changed. A basic version of the particle hierarchy can be seen below.

Where a represents an abstract class and a represents a concrete class.

Need to redo particle relationship

### Functional Model (Done collisions functions, not vector functions)

* Annihilation
  + CreateAnnihilationPhoton
    - EnergyToWaveLength
    - MassToEnergy
    - TotalRestMass
    - GetRestMass
    - VelocityToEnergy
    - TotalVelocity
    - GetVelocity
* Pair production
  + WavelengthToEnergy
  + MassToEnergy
  + CreatePairproductionOutputGreaterThanProtonRestMass
    - EnergyToVelocity
    - WavelengthToEnergy
    - MassToEnergy
    - GetRestMass
* Electron Capture
  + AtomCreator
    - CreateProtonList
    - CreateNeutronList
* Cyclotron
* Electrostatic Repulsion
* Atom Creator
  + CreateprotonList
  + CreateNeutronList

### Like an Onion

Alongside keeping the layers separate for management it also integrates very well into the functional mind-set. With functions in each layer calling functions a layer below it, meaning that a pure function in one layer will always stay pure as it cannot call a dirty function, which will only reside in the highest layer (UI). A pseudo-code example of how this function call would look would be;

*UI(Vector(Collision(Particle(Arguments))))*

Where the arguments are only the values required in the Particle’s constructor, these values would be obtained through the UI layer form user inputs.

*Variable Arguments => User Input*

*Variable CollisionOutputs = > UI(Vector(Collision(Particle(Arguments))))*

*Terminal Output => CollisionOutputs*

Note that Arguments is stored in a variable, this is not allowed in functional and this phrase involves sequence which is also not allowed in functional, however both are necessary for a UI.

Other layers may also require other arguments such as a random number.

*UI(Vector(Collision(Particle(Arguments),RandomNum)))*

### Why C# and not Haskell

There are a couple of reasons as to why c# is being used, one is that it is the language I am most familiar with as it is the one I have used throughout my A-level, Visual Studios is an excellent IDE that allows me to develop more accurate code without having to spend a long time looking for syntax errors. C# also can support a functional style which means my goals can be accomplished using it. C# is also much more supportive of a dirty UI which call the pure functions inside the program as opposed to Haskell UI which is much harder to set up and would take my focus away from the main core of the program.

## Specific problems and their solutions

### Heads and tails

Normal list in C# is simply a dynamic data structure that does not require a range to be set beforehand.

However a list in functional programing is quite different, for a start a list that cannot be changed is a lot less useful than one that can. Therefore how is it implemented?

An Functional list can be created in a couple of states much like a normal list, with out any elements, with only one element, or with multiple elements inside. There for the class requires multiple constructors, to create an empty list and a populated list. For the program I have decided to use a NuGet package to implement the functionality that would be provided by true functional langue’s such as Haskell. This package is the Quadrivia Functional Library <https://github.com/QuadriviaOrg/FunctionalLibrary>.

|  |
| --- |
| namespace Quadrivia.FunctionalLibrary |
|  | { |
|  | public class FList<T> |
|  | { |
|  | internal FList() |
|  | { |
|  | Empty = true; |
|  | } |
|  | internal FList(T head, FList<T> tail) |
|  | { |
|  | Empty = false; |
|  | Head = head; |
|  | Tail = tail; |
|  | } |
|  | internal bool Empty { get; private set; } |
|  | internal T Head { get; private set; } |
|  | internal FList<T> Tail { get; private set; } |
|  |  |
|  | public override string ToString() |
|  | { |
|  | return Empty ? |
|  | "" |
|  | :Tail.Empty ? |
|  | Head.ToString() |
|  | :Head + ", " + Tail; |
|  | } |
|  |  |
|  | public override bool Equals(object obj) |
|  | { |
|  | return !(obj is FList<T>) ? |
|  | false : |
|  | (Empty && (obj as FList<T>).Empty) || |
|  | (Head.Equals((obj as FList<T>).Head) && Tail.Equals((obj as FList<T>).Tail)); |
|  | } |
|  | public override int GetHashCode() |
|  | { |
|  | return Head.GetHashCode() + Tail.GetHashCode(); |
|  | } |
|  | } |
|  | } |

The implementation here contains the properties Head, Tail and Empty and they are immutable due to the private set. There are also the two constructors that are used to create the list. One creates an empty list and for that implementation the property Empty is set to true. Afterwards there is a constructor that takes in a generic value for Head and then a tail of type Flist. Therefore a list of one element will consist of a head and an empty tail. However this works very well for functional as each time a list needs to be changed a new list must be created with the new values. Therefore the old list becomes the tail that is passed in and the new value is the head that is prepended to the new list.

### How to Randomise a Determined Value

The notion of creating something random is inherently against the point of functional programing where everything is deterministic. So how can randomness come about when the program itself is designed around complete predictability? The answer is to keep the “Random” function very deterministic, therefore if one number is entered one hundred times, the same number would always be output one hundred times. Therefore the randomness must come form a variation in the seed number, this is where adding two seed number will drastically change the range of numbers that can be calculated but also a much less deterministic feel to the function. However still, if the same two numbers are input then the same number will always be output. Much like the Flist for the program uses the NuGet package Quadrivia for the implementation of a random function.

|  |
| --- |
| using System; |
|  |  |
|  | namespace Quadrivia.FunctionalLibrary |
|  | { |
|  | //Acknowledgement: Algorithm copied from https://www.codeproject.com/KB/recipes/SimpleRNG.aspx?display=Print |
|  | //which is based on original work by George Marsaglia. |
|  |  |
|  | public class FRandom |
|  | { |
|  | public readonly int Number; |
|  | private readonly uint U; |
|  | private readonly uint V; |
|  | internal FRandom(int result, uint u, uint v) |
|  | { |
|  | Number = result; |
|  | U = u; |
|  | V = v; |
|  | } |
|  |  |
|  | internal FRandom(uint u, uint v) |
|  | { |
|  | U = u; |
|  | V = v; |
|  | } |
|  |  |
|  | /// <summary> |
|  | /// Produces pseudo-random number from two specified non-zero seed values. |
|  | /// If either value is zero, a default will be used instead. |
|  | /// </summary> |
|  | /// <param name="u"></param> |
|  | /// <param name="v"></param> |
|  | /// <returns></returns> |
|  | public static FRandom Seed(uint u, uint v) |
|  | { |
|  | return new FRandom(u != 0 ? u: 521288629, v != 0 ? v : 362436069); |
|  | } |
|  |  |
|  | /// <summary> |
|  | /// Uses the default values for the seed, so returned FRandom will always be the same |
|  | /// </summary> |
|  | /// <returns></returns> |
|  | public static FRandom SeedDefault() |
|  | { |
|  | return new FRandom(521288629, 362436069); |
|  | } |
|  |  |
|  | /// <summary> |
|  | /// Seed the random generator using the system clock |
|  | /// </summary> |
|  | /// <param name="clockNow">typically DateTime.Now is passed in</param> |
|  | /// <returns></returns> |
|  | public static FRandom SeedFromClock(DateTime clockNow) |
|  | { |
|  | return Seed((uint)(clockNow.ToFileTime() >> 16), (uint)(clockNow.ToFileTime() % 4294967296)); |
|  | } |
|  |  |
|  | /// <summary> |
|  | /// Used for generating a sequence from a single known seed. |
|  | /// Skip(0,...) is the same as Next, but Skip(1,...) is the same |
|  | /// as calling Next twice but having to pass the result of the first |
|  | /// one into the second. |
|  | /// </summary> |
|  | /// <param name="skip">Number of random numbers in sequence to skip</param> |
|  | /// <param name="previous"></param> |
|  | /// <param name="minValue"></param> |
|  | /// <param name="maxValue"></param> |
|  | /// <returns></returns> |
|  | public static FRandom Skip(int skip, FRandom previous, int minValue, int maxValue) |
|  | { |
|  | return skip <= 0 ? |
|  | Next(previous, minValue, maxValue) |
|  | : Skip(skip - 1, Next(previous, minValue, maxValue), minValue, maxValue); |
|  | } |
|  |  |
|  | public static FRandom Next(FRandom previous, int minValue, int maxValue) |
|  | { |
|  | return new FRandom(NextRanged(previous, minValue, maxValue), NewU(previous.U), NewV(previous.V)); |
|  | } |
|  |  |
|  | private static uint NewU(uint u) |
|  | { |
|  | return 36969 \* (u & 65535) + (u >> 16); |
|  | } |
|  |  |
|  | private static uint NewV(uint v) |
|  | { |
|  | return 18000 \* (v & 65535) + (v >> 16); |
|  | } |
|  |  |
|  | private static double Next(uint oldU, uint oldV) |
|  | { |
|  | return ((NewU(oldU) << 16) + NewV(oldV) + 1.0) \* 2.328306435454494e-10; |
|  | } |
|  |  |
|  | private static int NextRanged(FRandom previous, int minValue, int maxValue) |
|  | { |
|  | return (int)(minValue + Next(previous.U, previous.V) \* (maxValue - minValue)); |
|  | } |
|  |  |
|  | } |
|  | } |

As shown in this code from the package I am using, it starts by defining the properties of the class Number, U and V. Where Number is the result of past calculations, U and V are the two seed numbers. Notice that there are two constructors on this class, one that takes three perimeters and one that take two. This is because for the random function to work it must use its previous iteration but cannot change it. Therefore the previous result must be used to find the next one in a new instance of FRandom. The constructor with only two perimeters is used in the very first iteration of the FRandom function with the seed method. The benefits of using this method are the testing of randomised values. As it is deterministic test can be carried out using set values to make sure other parts of the program are working.

### One of these is Not Like the Others

The problem of the photon has troubled physicist for centuries so it will come as no surprise that it will also cause problems for this project. Due to the photon having no mass, unlike everything else, a constant velocity, unlike everything else and is both a wave and a particle at the same time will make it difficult to incorporate their calculations into functions that’s implementations are designed for all other particles.

There are two solution to this problem, changing all other particles to work using their De Broglie wavelength to match the wave like nature of light or to use overloaded functions. Overloaded functions are functions with the same implementation; however take different perimeters. This means upon runtime if a function that is overloaded is passed a photon it will use the function designed for it and return the appropriate answer. A Pesudo-Code example of an overloaded function to return the combined energy of two of the same particles is as shown.

*CombinedEnergy (Generic Particle1, Generic Particle2)*

*{return(Particle1.RestMass\*c2) + (Particle2.RestMass\*c2)}*

*CombinedEnergy (Photon Particle1, Photon Particle2)*

*{return((h\*c)/Particle1.Wavelength) + (h\*c)/Particle2.Wavelength)}*

The first function uses the equation E = mc2 (E = Energy, m = Rest Mass, c = Speed of Light in a vacuum) to find the energy of a particle however this does not work for photons due to them having no mass and therefore by this equation zero energy.

The second function uses the equation E = h\*c/Wavelength (E = Energy, h = Plank’s Constant, c = Speed of Light in a vacuum) to find the energy of the photon, which will not work for the other particles.

## Areas involving technical complexity

### Selection

Selection in procedural programing is fairly simple in high level languages with the IF statement being found in most languages. However in functional programing does not allow selection in the form of an If statement:

*IF (Boolean Condition) Then*

*Return a*

*Else*

*Return b*

As this is not allowed and selection is a vital part of my program, I must turn to ternary operators. A ternary operator is a selection statement that executes on one line, therefore it does not break any of the rules of functional program, a ternary operator would be implemented like this:

*Return Boolean Condition ? a : b*

Where everything left of the question mark is the selection itself and the colon acts as the ELSE.

Selection in True Functional

The functional language Haskell also has selection, in the form of guards. Guards are represented in Haskell using the vertical line symbol, | , or “pipes” and follow a similar format to the switch case selection statement:

*Switch (Value)*

*Case(a):*

*Execute code*

*Break;*

*Case (b):*

*Execute code;*

*Break;*

The same thing can be stated in Haskell:

*Function Definition*

*Func (f)*

*| f == a = Execute code*

*| f == b = Execute code*

Notice the pipes act in a very similar way to the case statements shown above however they do not execute in sequence unlike the case statements.

### Nested Selection

With the same line of thought from the section above, nested selection also uses ternary operators in functional, therefore instead of being implemented like they would in sequence:

*IF (Boolean Condition 1) Then*

*IF (Boolean Condition 2) Then*

*Return a*

*Else*

*Return b*

*Else*

*IF (Boolean Condition 3) Then*

*Return c*

*Else*

*Return d*

As a ternary operator it is implemented like so:

*Return Boolean Condition 1 ? Boolean Condition 2 ? a : b : Boolean Condition 3 ? c : d*

*Return Boolean Condition 1 ? Boolean Condition 2 ? a : b : Boolean Condition 3 ? c : d*

Where each colour represents one condition: This functionality allows for multiple selection statement to execute on the same line

### Recursion

One of the major difficulties with functional programing is the inability to use loops, this means no iteration over data, no checking or creation of lists and everything can only be used once before the program has to be restarted, however this can be overcome by using recursion.

Recursion is the process of calling a function from within itself to create iteration in the program. A recursive function will store the result of each pass on the call stack before calling itself with a slightly smaller version of the same problem. Once an end condition is reached where the function will no longer call itself, the call stack is then popped, returning all of the previous results.

### Higher Order Functions

A higher order function is simply defined as a function that either takes a function as an argument or returns a function<https://en.wikipedia.org/wiki/Higher-order_function>. Therefore Higher order functions are everywhere in functional programing. All other functions are know as first order functions where normal types such as int and string are the perimeters and they are also retuned.

#### Inbuilt

##### Map

The map function is an inbuilt capability of functional programing where a list is copied into another list with all of the elements passing through a function.

With the function

*MultiplyByTwo (int a)*

*Return a \* 2*

And a list of type int

*List {1,2,3,4,5}*

The map function

*Map MultiplyByTwo List*

This will take each element in List, apply the function MultiplyByTwo to them and then create a new list of the same type and of the same size and place each new element into them. Therefore the new list produced would be.

*Listb {2,4,6,8,10}*

Where List and Listb are two different lists with the same data type and of the same size.

##### Filter

The filter function is much like map where it produces a new list of the same type however filter conducts a Boolean check over each element in the list and will only place it into the new list if it passes the check.

Doing this in procedural would involve looping over an if statement much like:

*For length of List*

*If Item In list = condition*

*Add Item to New list*

However in functional using the filter function

*Filter condition List*

A more concrete example of this would be a list of type int being filtered

*List {1,2,3,4,5}*

*Filter (>3) List*

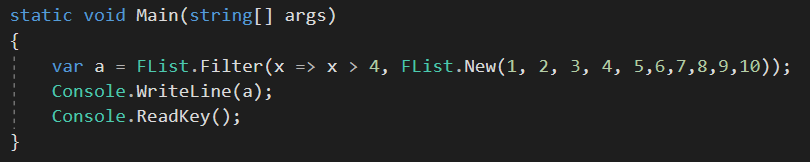
Would produce the list

*Listb {4,5}*

Where List and Listb are of the same type but different lengths

##### Lambda

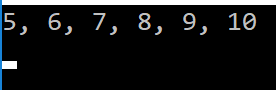
The condition function can be passed as a lambda, which is a locally defined function instead of a func, which is defined elsewhere, not as a parameter of another function. Below is an example of a lambda being used in the filter function.



The filter function requires (Func, Flist) parameters, the Flist argument is a new list of numbers 1-10. However before that, the first argument is;

*x => x > 4*

This is the lambda part where the locally defined function of a variable is passed in called x, the function checks if x is greater then 4, if so true is returned. If true is returned filter then puts that item into a new list, and we see the new list output.



##### Fold

The fold function is different to both Map and Filter as it does not produce a new list at the end but a single value or a string. There are also two types of fold, FoldR and FoldL. Some of the time it does not matter which one is used, however with certain operates such as – and / it does.

The fold function will apply an operator to two items in a list to produce a new item. It will then apply the operator to this item and the next one in the list until the list is of length 1 and a single answer is produced.

*List {1,2,3,4,5}*

*FoldR + List*

Would produce the answer

*15*

The difference in FoldR and FoldL is the order they iterate through the list, FoldR preforms its calculation like,

(1+(2+(3+(4+(5+0)))))

(1+(2+(3+(4+5))))

(1+(2+(3+9)))

(1+(2+12))

(1+14)

(15)

Whereas FoldL will preform a calculation like,

(((((0+1)+2)+3)+4)+5)

((((1+2)+3)+4)+5)

(((3+3)+4)+5)

((6+4)+5)

(10+5)

(15)

However This only work with certain operators as with others the answer produced will be different

*FoldR – List*

*(1-(2-(3-(4-(5-0)))))*

*(1-(2-(3-(4-5))))*

*(1-(2-(3--1)*

*(1-(2-4)*

*(1--2)*

*(3)*

Whereas

*FoldL – List*

*(((((0-1)-2)-3)-4)-5)*

*((((-1-2)-3)-4)-5)*

*(((-3-3)-4)-5)*

*((-6-4)-5)*

*(-10-5)*

*(-15)*

### Tuples

Tuples are a data structure much like an array as it is an indexed way of storing data that has a limit of space. However the difference with tuples is that they can only hold seven items, in c#, and those items do not have to be of the same type, and its indexing starts from one for some reason. This allows functions to generate multiple answers of different types and return them. Furthermore tuples are immutable which only reinforces the functional style.

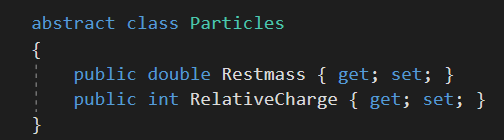
### How to do Functional in C#

As C# is not a pure functional language such as Haskell or F#, however it is a multi-paradigm language therefore it can support functional but only with some of the benefits of a true functional language, such as the inability in C# for referential transparency. Equally as C# is not a functional language it does not have the same inbuilt restrictions as Haskell so non-functional code will still work, this means I will have to be very disciplined in my development to avoid making functions “dirty”

### Inheritance

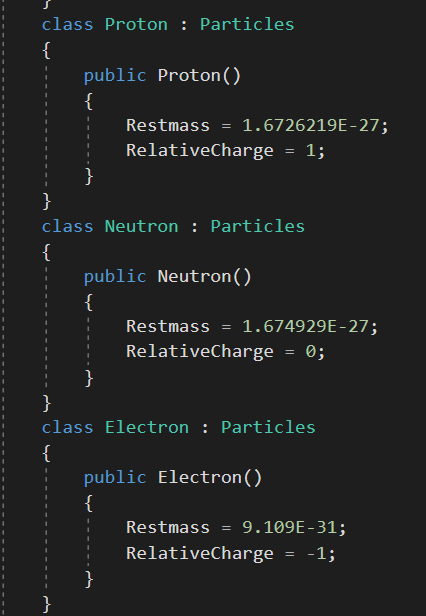
Inheritance is used to group many different user defined classes under one set where they usually share properties and methods. To set up a basic inheritance structure including some particles is as follows.

To start we create the abstract class that will sit at the top of the tree: this class will be Particles



This is a relatively simple class for this demonstration, but it sets the properties that classes below it will have and as it is abstract it s not able to be reached directly and there will never be an instance of Particles generated by the program.

Next is to add some particles that will inherit from the abstract class



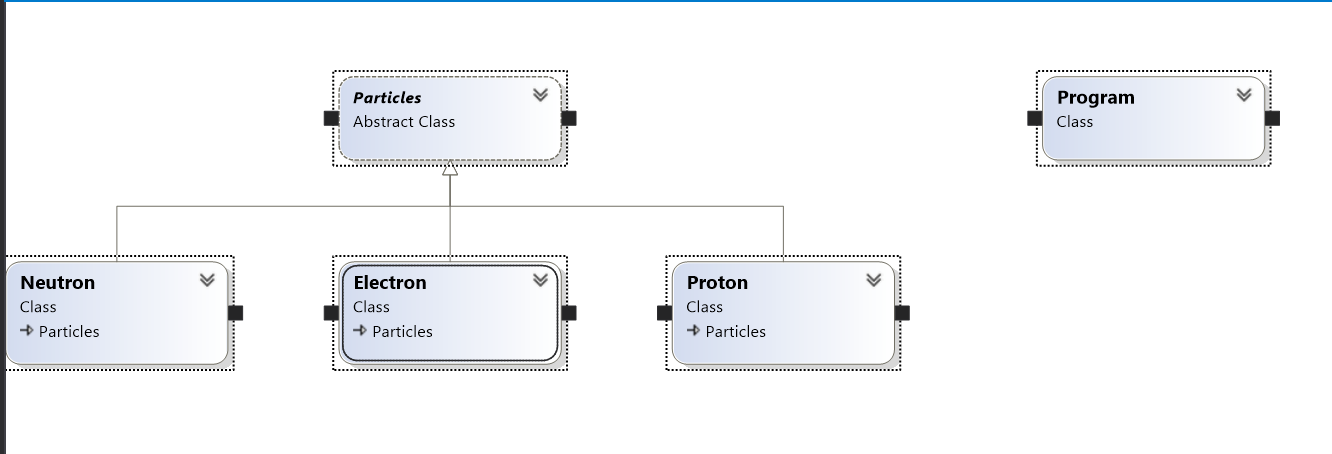
These are three simple particles that are only using the properties by the abstract class in their constructors. Notice that they don’t have properties of their own. The inheritance comes from the extension of Particles from the name of the class.

*Class Proton : Particles*

. Is the name of the class.

. Is the extension.

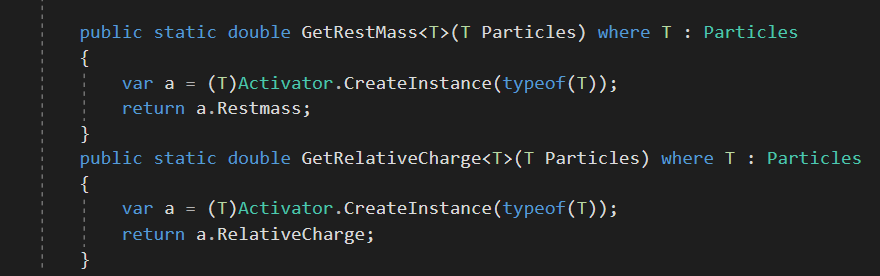
. is the name of the abstract class.

This shows the class diagram that is generated by the program. The Particles class is the superclass to the three underneth whereas the Program class is not a superclass as nothing inherits from it. 

### Generics

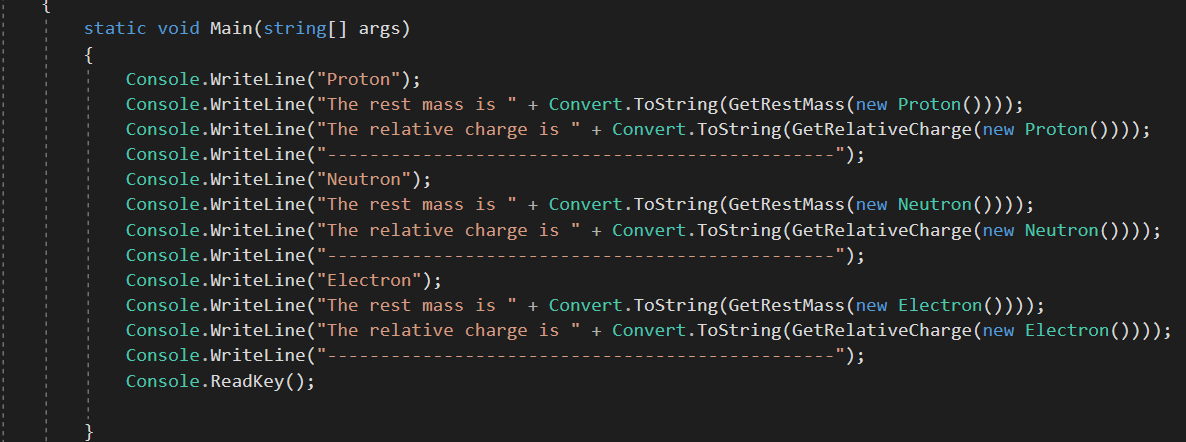
One major issue that can occur is how to passes functions different permeates at run time and not hardcode them in beforehand. The answer is generics, generics are a very useful too provided by modern programing languages where the boundaries of dynamic and static languages become blurred. Generics allow a function to specify a range of types in allowed in its argument, instead of just one. This works very well with user defined types that follow a hierarchical structure where a group of classes all derive from one abstract class. As all of the classes derive for one would contain the same structure and most of the same properties, different methods and functions designed around the abstract class would work on all of the others as well.

Using the example from inheritance we can create two functions that obtain the properties for all of these classes without having to specify directly which class they are working with.



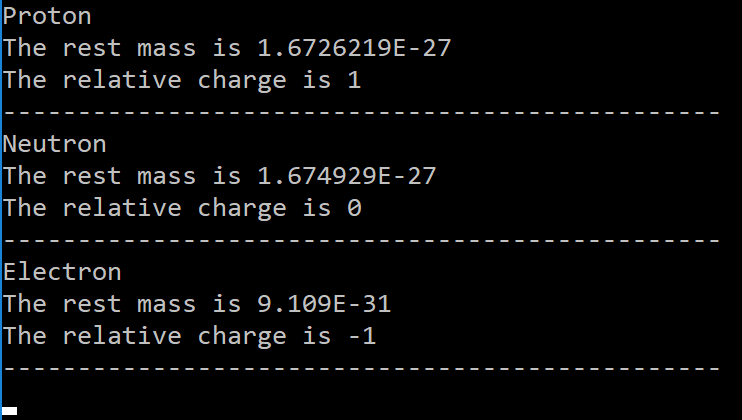
This is where the generics appear. The function shows it is using generics by having <T> after the name, after are the perimeters where the type of the variable passed in is of type T or type generic, then to identify the range of types allowed to be entered the function states that T extends Particles or that T extends the abstract class that the other classes inherit from. The second line of the function is the creation of the instance of T where if the class required parameters to be added they would be added after the *typeof(T).* The program is now using a method called reflection, reflection is where the code is able to obtain the type of an object without having the type hardcoded into the program, as can be seen this functionality is invaluable when using generics. *.*Once the variable a has been defined as the type of T the properties of it can now be accessed much like a normal class. This is not completely functional as there are the use of variables however for simplicity in explaining generics I have returned to procedural.

The final step is the UI,



Where each function is called more than once with different class types.

And the final output is



# Technical implementation

[This is your complete code, with some annotations added to highlight complexity and/or reference sections in Design]

Paste in in colour

Use a class diagram

Need to add subheadings for all classes used

Add GitHub link + screenshot of checkins

Include a line count for each file and then a total line count

## Particle Collision Project

## Colision

# 

# Testing

## Test Plan and Inbuilt Security

#### This testing will consist of two elements to the testing. The high level testing is testing the user interaction with the system and if it guards against the user inputting values that don’t work. This will involve using the program and entering accepted values and incorrect values, providing screenshots of what is produced. The second form of the testing is the low level testing where unit-tests are used to test each function in the program. This works very well with functional programing as each function can be called and tests individually to allow me to get full coverage throughout the program.

#### Functions may throw exceptions, as some inputs would be invalid, however the UI will stop these inputs from being input in the first place because if the user is unable to enter a different data type then the functions in the core of the program are already protected form this eventuality. There for the functions will only be guarding against problems that may arise within the functions themselves such as “List is empty” exceptions, this can be done in a functional way using ternary operators, whereas the UI will catch errors that cannot be caught functionally.

## High Level Test Plan (Outside in)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| What Testing? | What Case? | How to Reproduce | Expected | Actual? |
| 1. Annihilation | Happy | Enter 1 | The option to chose either a proton anti-proton pair or electron-positron pair to annihilate |  |
|  | Fail | Enter “A”  Enter 8 | "Invalid input, please try again"  "Number not offered as an option, please enter a number shown" |  |
| Proton – Anti proton | Happy | Enter 1 |  |  |
|  | Fail | Enter “A” |  |  |
| Electron - Positron | Happy | Enter 2 |  |  |
| Enter the modulus of the velocity | Happy | Enter 100 | To go onto the next line |  |
|  | Edge | Enter 0 | To go onto the next line |  |
|  | Fail | Enter “A” | Invalid input and will give option to enter again |  |
| 2. Electron Capture | Happy | Enter 2 | Give option to enter mass number |  |
|  | Fail | Enter “A” | Invalid input and will give option to enter again |  |
| Enter mass number | Happy | Enter 56 | Give option for atomic number |  |
|  | Edge | Enter 0 | Invalid input and will give option to enter again |  |
|  | Fail | Enter 296 | Invalid input and will give option to enter again |  |
| Enter proton number | Happy | Enter 12 | Will produce final result |  |
|  | Edge | Enter 1 | Produce null error |  |
|  | Fail | Enter 119 | Invalid input and will give option to enter again |  |
| 3. Pair Production | Happy | Enter 3 | Allow to enter the wavelength |  |
|  | Fail | Enter “A” | Invalid input and will give option to enter again |  |
| Enter Wavelength | Happy | Enter 100 | Will move onto frequency |  |
|  | Edge | Enter 1E-10 | Will move onto frequency |  |
|  | Fail | Enter -10 | Invalid input and will give option to enter again |  |
| Enter Frequency | Happy | Enter 100 | Will produce final result |  |
|  | Edge | Enter 1E-10 | Will produce final result |  |
|  | Fail | Enter -10 | Invalid input and will give option to enter again |  |
| Velocity Selector | Happy | Enter 4 | Allow user to choose between protons and electrons |  |
| Atom interactions | Happy | Enter 5 | Begin the creation of an atom |  |
| Enter atomic number | Happy | Enter 12 | Move onto mass number |  |
|  | Edge | Enter 1 | Move onto mass number |  |
|  | Fail | Enter 119 | Invalid input and will give option to enter again |  |
| Enter Mass number | Happy | Enter 56 | Give options for the interactions |  |
|  | Fail | Enter 0 | Invalid input and will give option to enter again |  |
| Beta-Plus | Happy | Enter 1 |  |  |
|  | Fail | Enter 5 | Invalid input and will give option to enter again |  |
| Beta-Minus | Happy | Enter 2 | Produce final output |  |
| Alpha decay | Happy | Enter 4 | Produce final output |  |
| Exit to main menu | Happy | Enter “10” | The program reverts to the first menu |  |
| Electrostatic Repulsion | Happy | Enter 6 | Move onto choice between protons and electrons |  |
| Two Protons | Happy | Enter 1 | Move onto first velocity |  |
| Enter Velocity | Happy | Enter 100 | Move onto starting positions |  |
|  | Error | Enter -10 | Invalid input and will give option to enter again |  |
| Enter coordinates | Happy | 2  3  4 | Will move onto second velocity |  |
| Two Electrons | Happy | Enter 2 | Move onto first velocity |  |
| Cyclotron | Happy | Enter 7 | Will allow choice of particles to accelerate |  |
| Protons | Happy | 1 | Option to fill in electric field strength |  |
|  | Error | 5 | “Invalid answer” and the menu will be shown again |  |
| Electrons | Happy | 2 | Option to fill in electric field strength |  |
| Positrons | Happy | 3 | Option to fill in electric field strength |  |
| AntiProtons | Happy | 4 | Option to fill in electric field strength |  |
| Electric field Strength | Happy | 10 | Move onto magnetic field strength |  |
|  | Error | -10 | Invalid answer and will give option to repeat |  |
| Magnetic field strength | Happy | 10 | Will move onto target velocity |  |
|  | Error | -10 | Invalid answer and will give option to repeat |  |
| Target velocity | Happy | 100 | Will output final answer |  |
|  | Error | -10 | Invalid answer and will give option to repeat |  |

## Low Level Test Plan (Inside out)

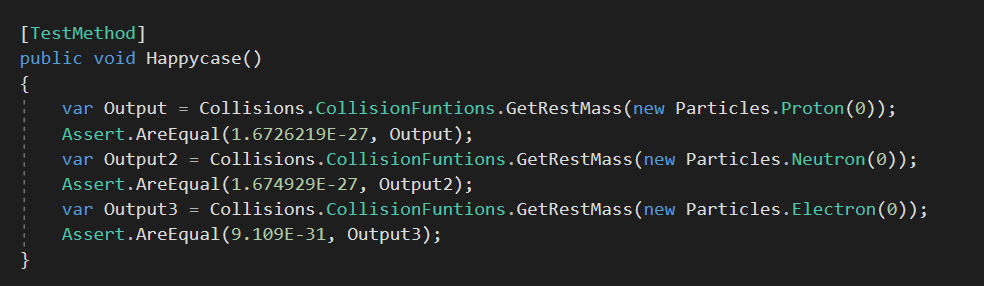
### Collision Functions

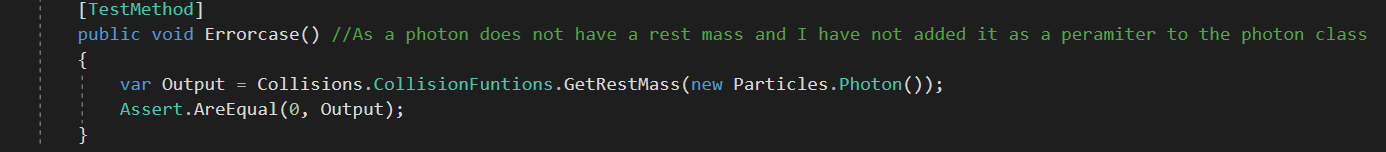
#### Lowest Functions

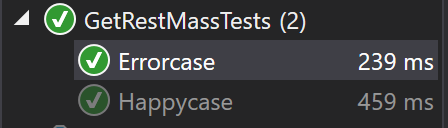
##### GetRestMass Func

static Func<Particles.Particle, double> GetRestMass = x => x.RestMass;

* Normal –



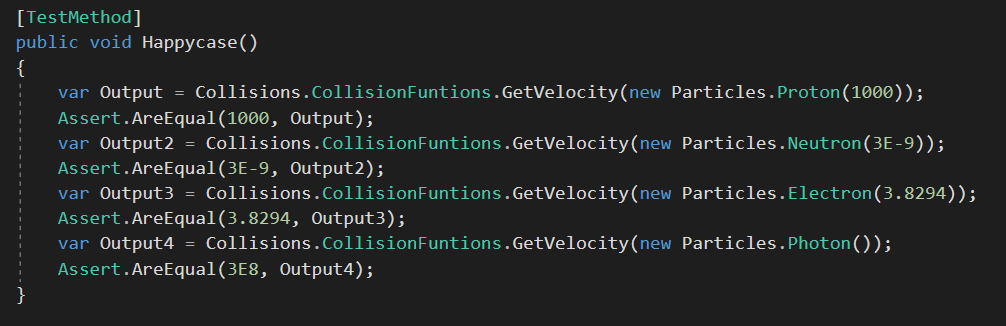
* Error – Pass in an object without a rest mass
* Tests Passing



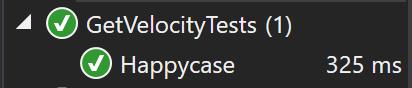
##### GetVelocity Func

static Func<Particles.Particle, double> GetVelocity = x => x.Velocity;

* Normal - Pass in a particle which has a velocity



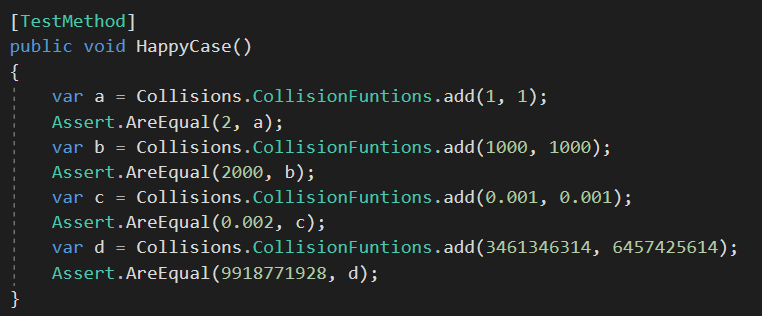
* Edge – N/A
* Error - Pass in an object without a velocity
* Test Passing



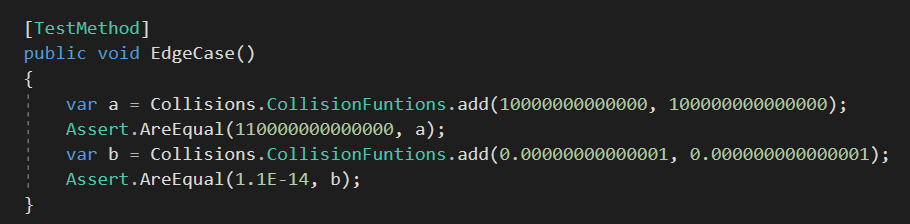
##### Add Func

static Func<double, double, double> add = (x, y) => x + y;

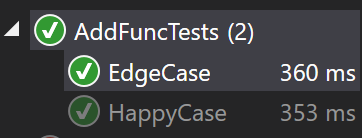
* Normal –



* Edge –



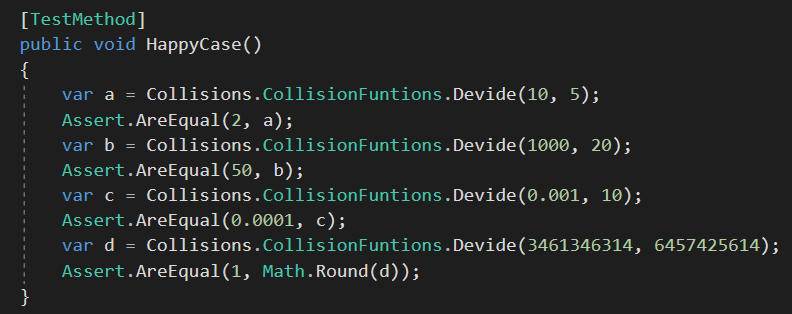
* Error – Unable to pass anything but a double
* Tests passing



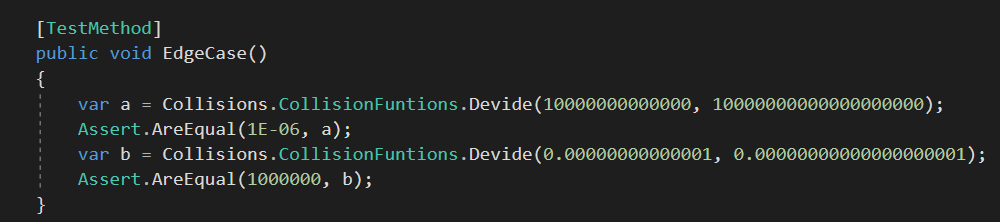
Devide Func

public static Func<double, double, double> Devide = (x, y) => x / y;

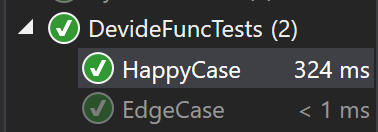
* Normal –



* Edge –

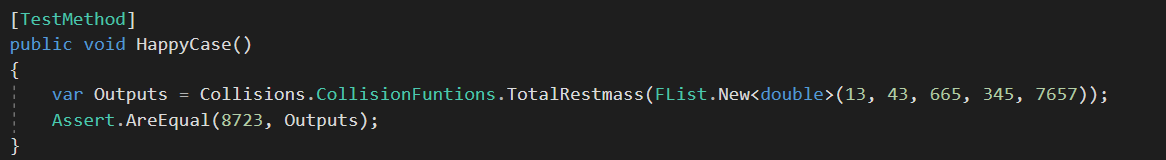


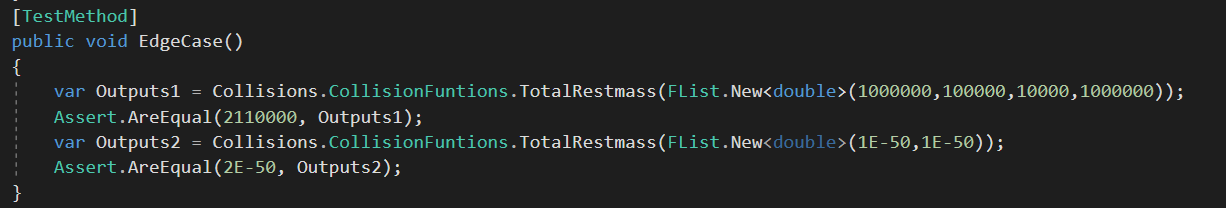
* Error – Unable to pass anything but a double
* Tests Passing –



##### TotalRestMass Function

public static double TotalRestmass(FList<Double> ListOfMasses)

* Normal – Enter list of Masses
* Edge – Entering very large masses and very small masses



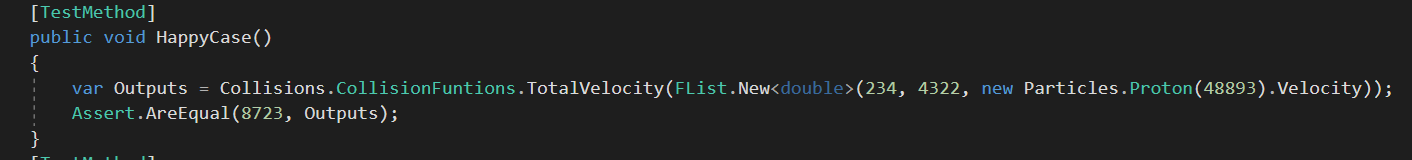
* Error – Enter list of objects
* Tests passing



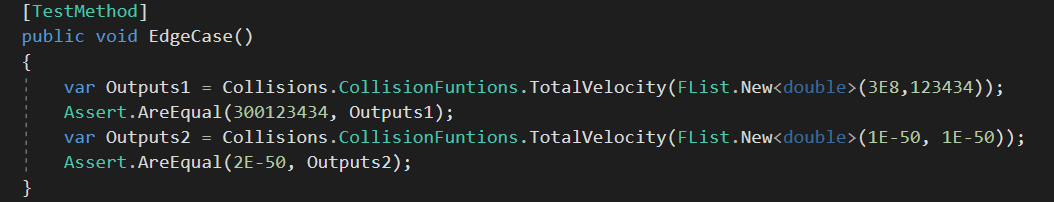
##### TotalVelocity Function

public static double TotalVelocity(FList<double> ListOfVelocity)

* Normal - Enter list of Velocitys



* Edge – Entering very large masses and very small velocities



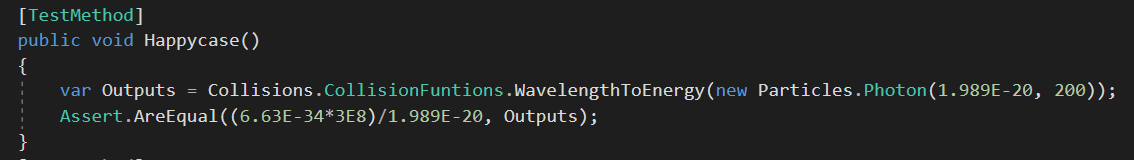
* Error - Enter list of objects
* Tests passing



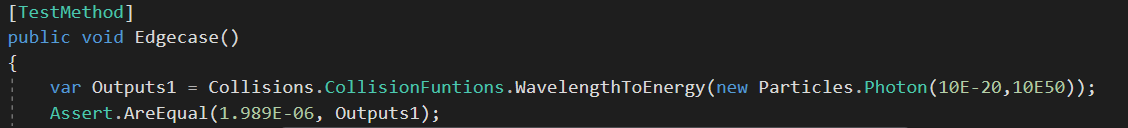
##### WavelengthToEnergy Function

public static double WavelengthToEnergy (Photon Photon)

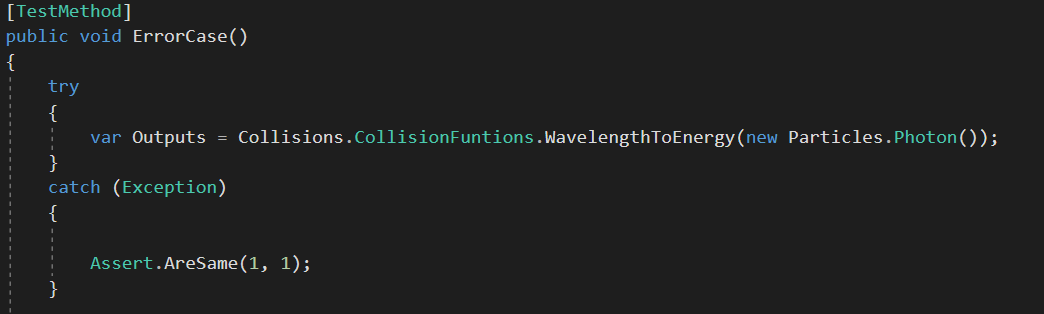
* Normal – Enter normal wavelength Photon



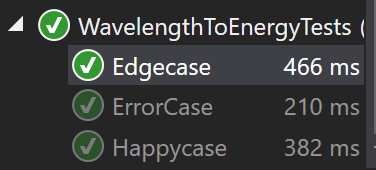
* Edge – Enter one very low wavelength and one very high



* Error –



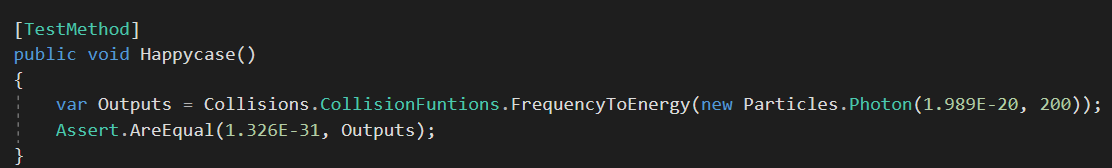
* Tests Passing



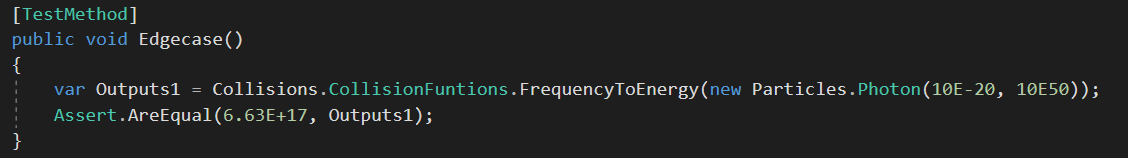
##### FrequencyToEnergy

public static double FrequencyToEnergy (Photon Photon)

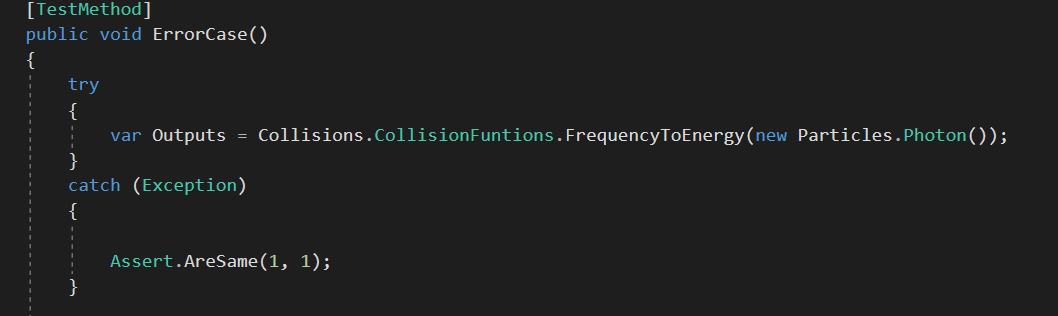
* Normal - Enter normal frequency Photon



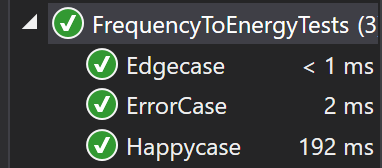
* Edge - Enter one very low wavelength and one very high



* Error –



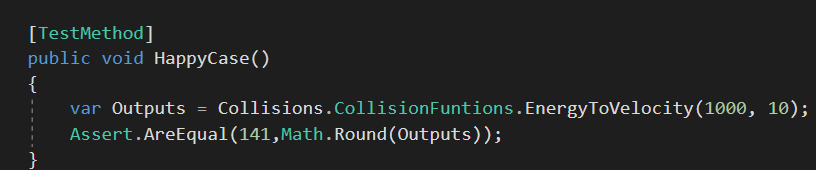
* Tests Passing



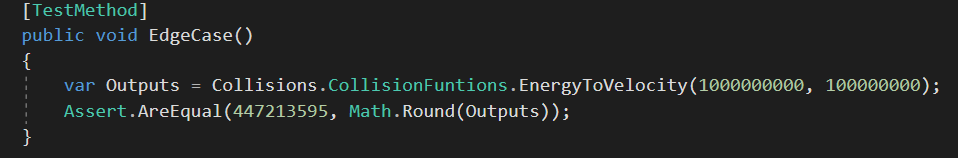
##### EnergyToVelocity

public static double EnergyToVelocity(double energy, double mass)

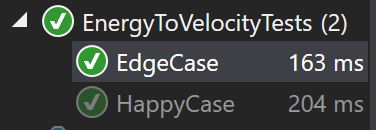
* Normal –



* Edge –



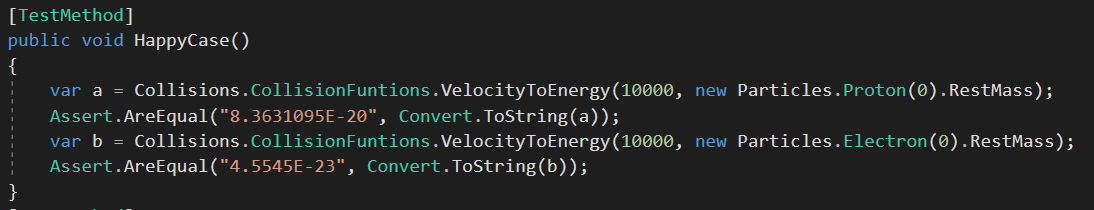
* Error –
* Tests Passing –



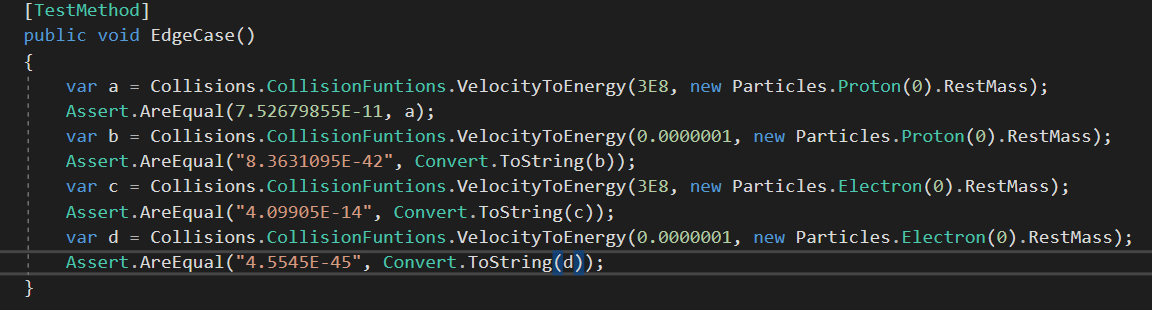
##### VelocityToEnergy

private static double VelocityToEnergy(double totalParticleVelocity, double totalRestMass)

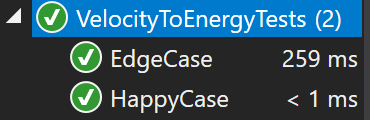
* Normal –



* Edge –



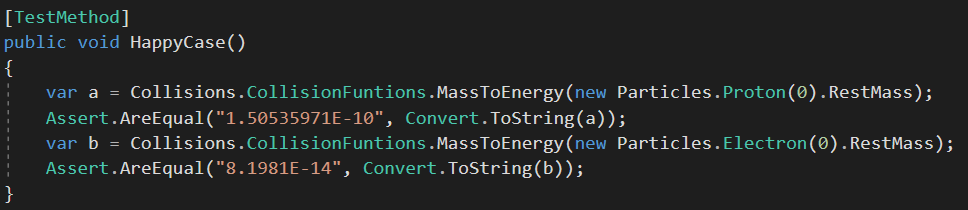
* Error –
* Tests Passing



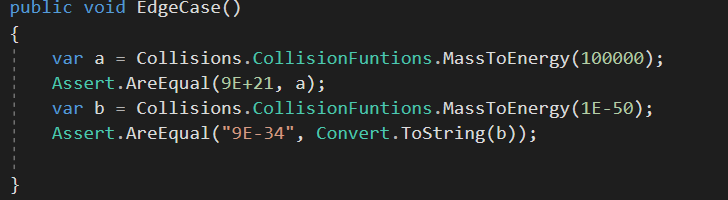
##### MassToEnergy

private static double MassToEnergy(double totalRestMass)

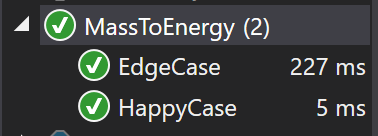
* Normal –



* Edge –



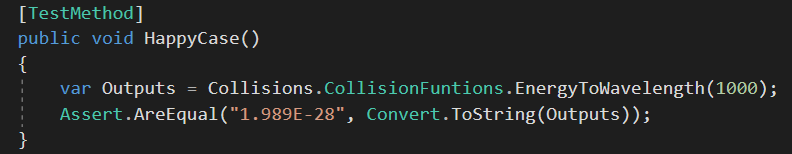
* Error -
* Tests Passing –



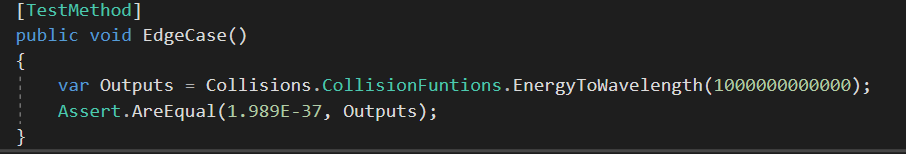
##### EnergyToWaveLength

public static double EnergyToWavelength(double energy)

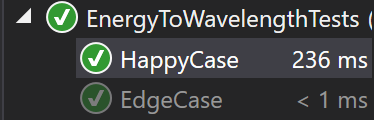
* Normal –



* Edge –



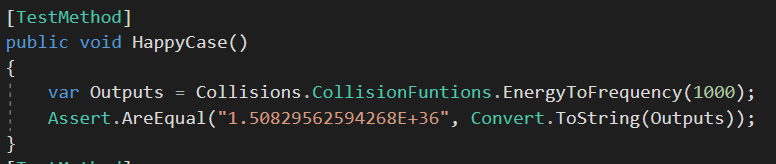
* Error –
* Test Passing



##### EnergyToFrequency

public static double EnergyToFrequency(double energy)

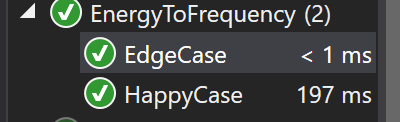
* Normal –



* Edge –



* Error –
* Tests Passing



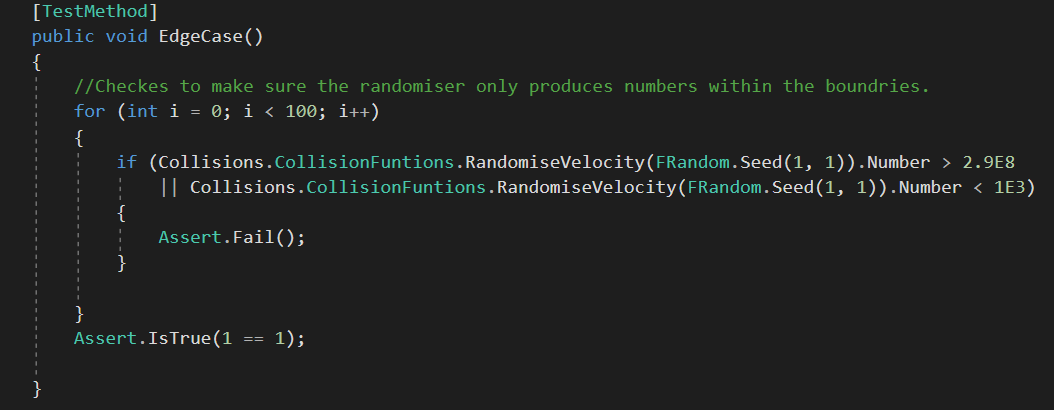
##### RandomiseVelocity

public static FRandom RandomiseVelocity(FRandom Rand)

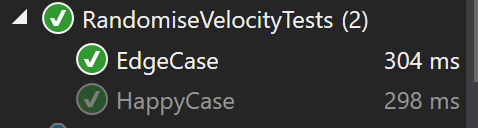
* Normal –



* Edge –



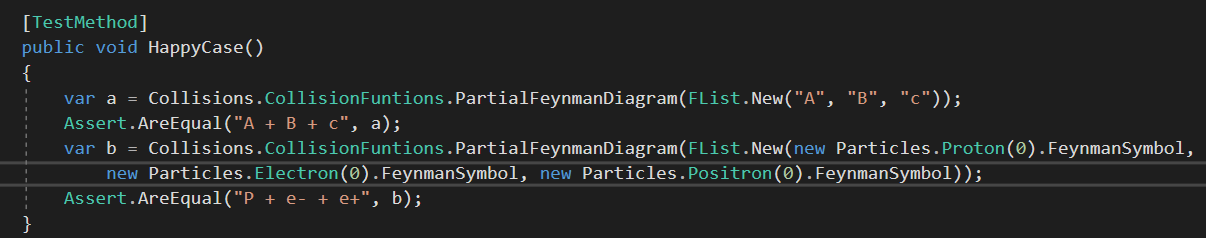
* Error –
* Test Passing



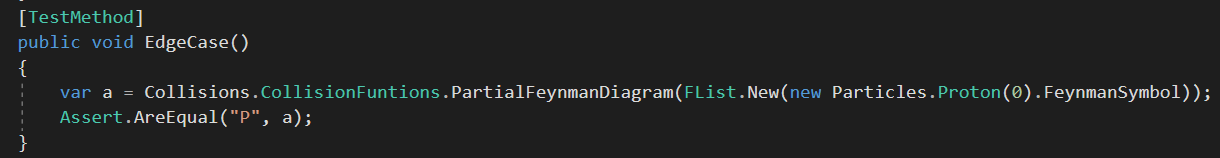
##### PartialFeynmanDiagram

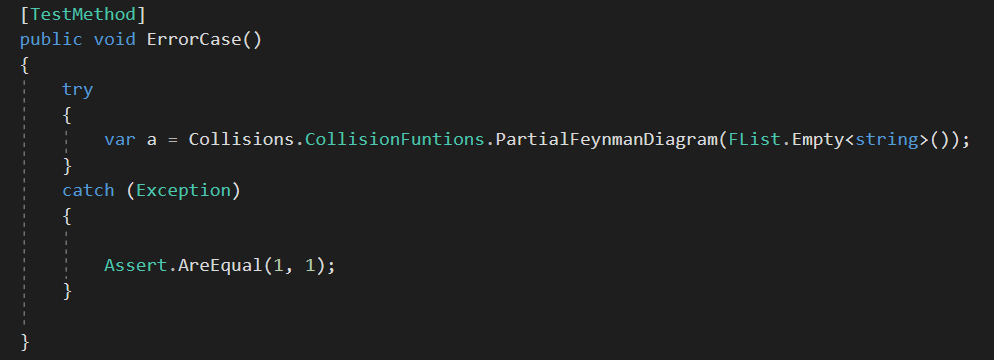
public static string PartialFeynmanDiagram(FList<string> List)

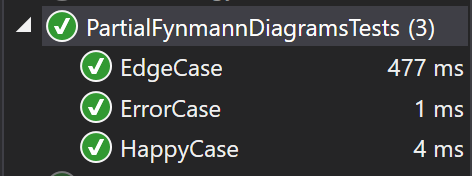
* Normal –



* Edge –



* Error –
* 
* Tests Passing –



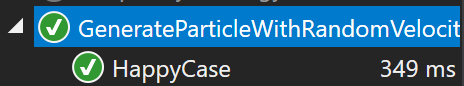
##### GenerateparticleWithRandomVelocity

public static Particle GenerateParticleWithRandomVelocity<T>(T particle, FRandom rand) where T : Particle

* Normal –



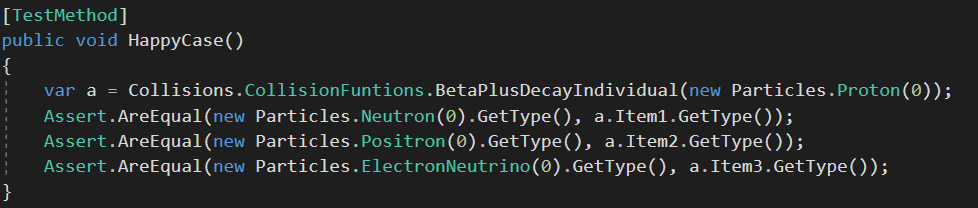
* Edge -
* Error –
* Tests Passing



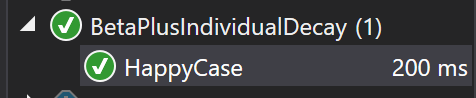
##### BetaPlusDecayIndividual

public static Tuple<Neutron, Positron, ElectronNeutrino> BetaPlusDecayIndividual(Proton P)

* Normal –



* Edge -
* Error –
* Tests Passing

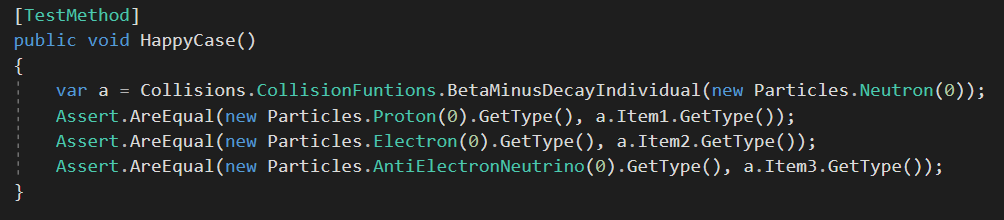


##### BetaMinusDecayIndividual

public static Tuple<Proton, Electron, AntiElectronNeutrino>

BetaMinusDecayIndividual(Neutron N)

* Normal –



* Edge -
* Error –
* Tests Passing

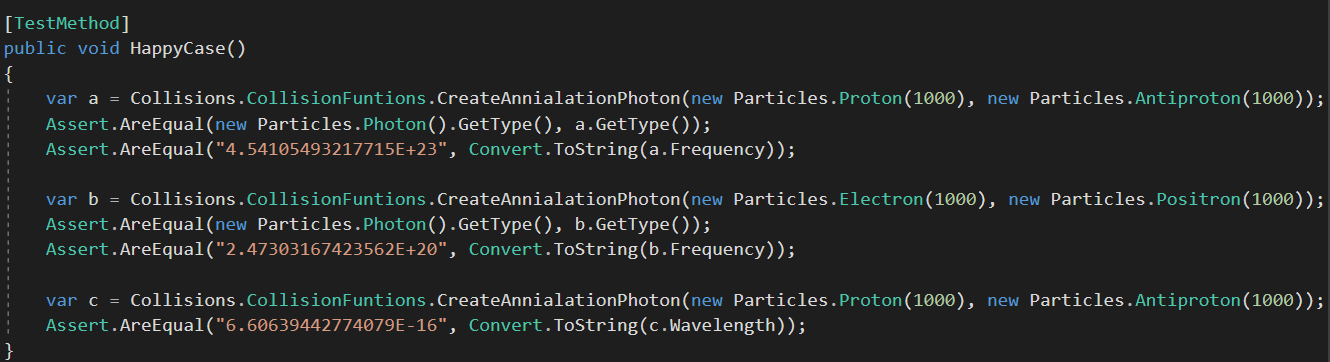


#### Middle Functions

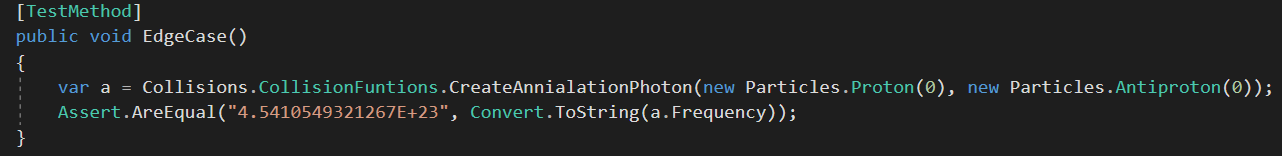
##### CreateAnnialationPhoton

public static Photon CreateAnnialationPhoton(Particle Particle, Particle AntiParticle)

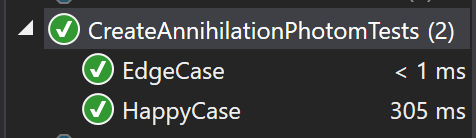
* Normal –



* Edge –



* Error –
* Tests Passing



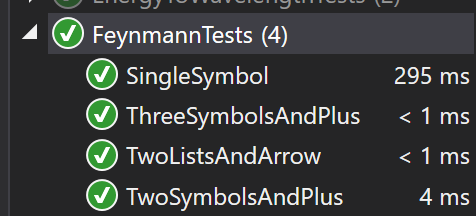
##### FeynmanDiagram

public static string FeynmanDiagram(FList<String> Inputs, FList<String> Outputs)

* Normal –



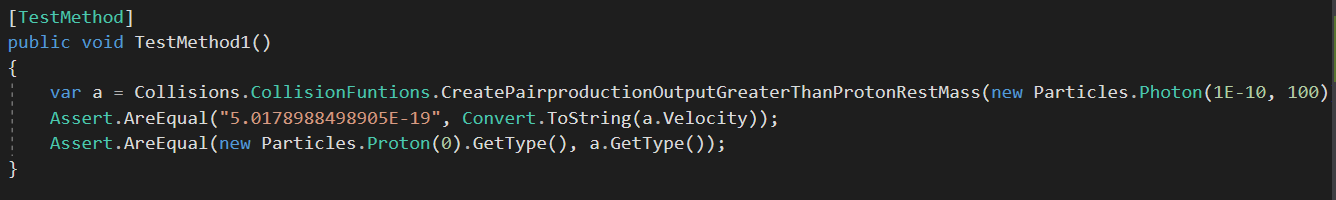
* Edge –
* Error –
* Tests Passing



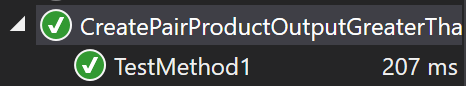
##### CreatePairproductionOutputGreaterThanProtonRestMass

public static Proton CreatePairproductionOutputGreaterThanProtonRestMass(Photon Photon)

* Normal –



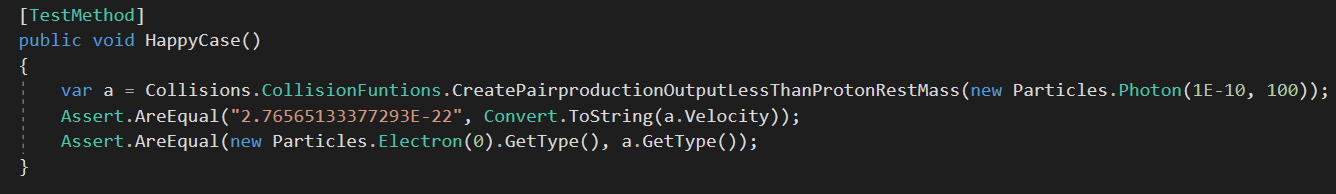
* Edge –
* Error –
* Tests Passing



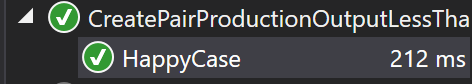
##### CreatePairproductionOutputLessThanProtonRestMass

public static Electron CreatePairproductionOutputLessThanProtonRestMass(Photon Photon)

* Normal –



* Edge –
* Error –
* Tests Passing



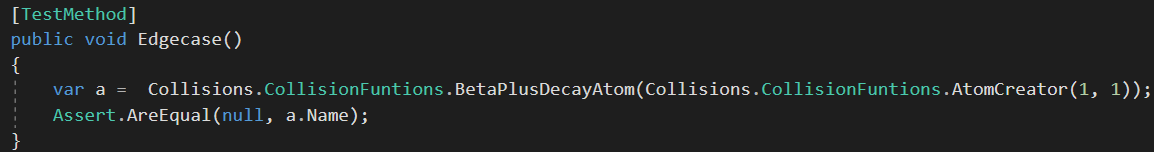
##### BetaPlusDecayAtom

public static Atom BetaPlusDecayAtom(Atom A)

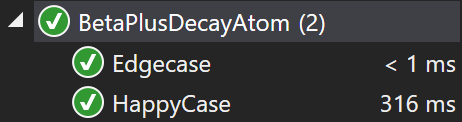
* Normal –



* Edge –



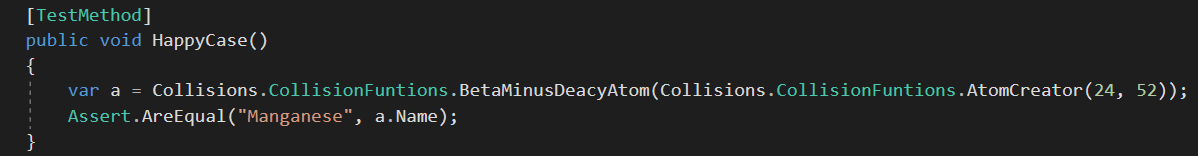
* Error –
* Tests Passing



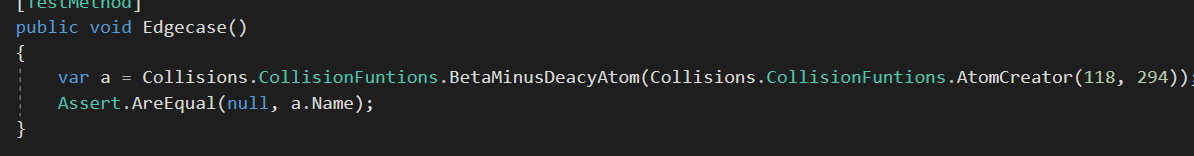
##### BetaMinusDeacyAtom

public static Atom BetaMinusDeacyAtom(Atom A)

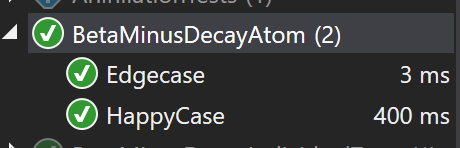
* Normal –



* Edge –



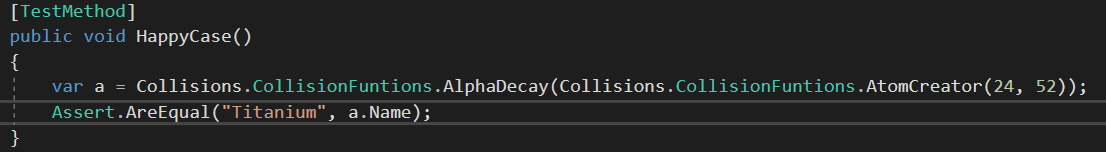
* Error –
* Tests Passing



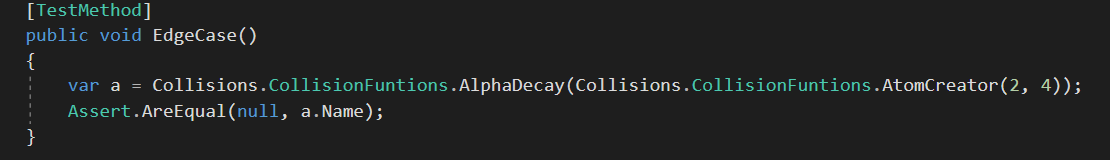
##### AlphaDecay

public static Atom AlphaDecay (Atom A)

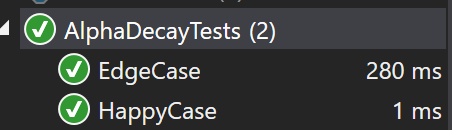
* Normal –



* Edge –



* Error –
* Tests Passing

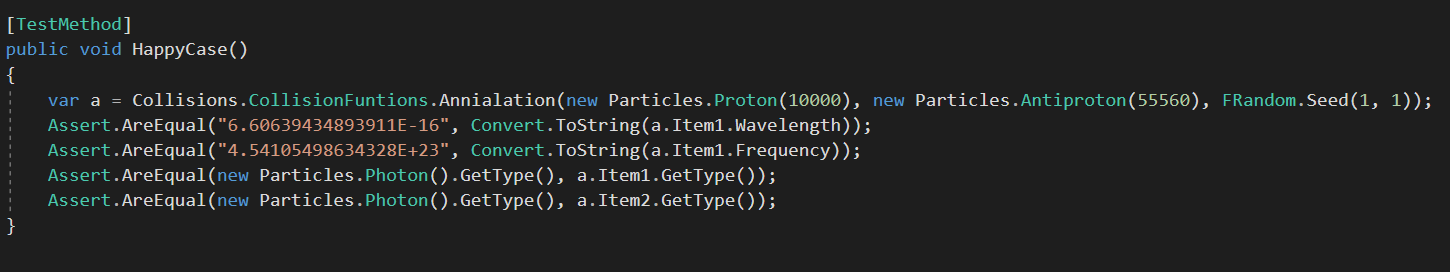


#### Heights Functions

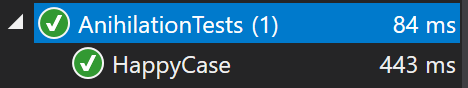
##### Annialation

public static Tuple<Photon, Photon> Annialation(Particles.Particle Particle, Particles.Particle AntiParticle, FRandom Rand)

* Normal –



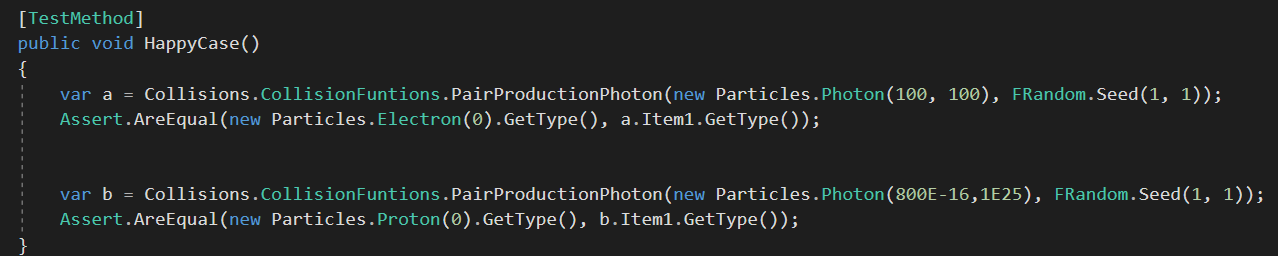
* Edge –
* Error –
* Tests Passing



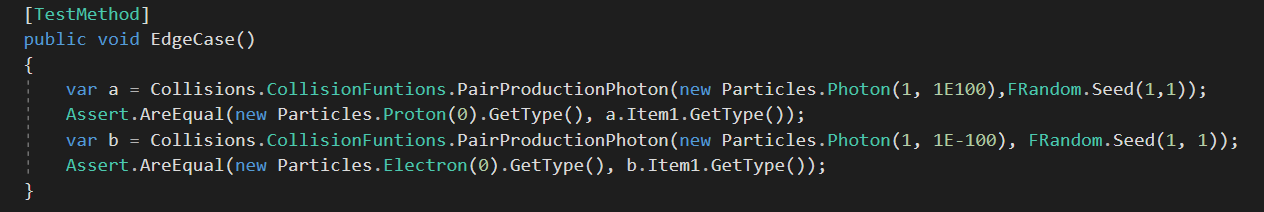
##### PairProductionPhoton

public static Tuple<Particle, Particle> PairProductionPhoton(Photon Photon, FRandom Rand)

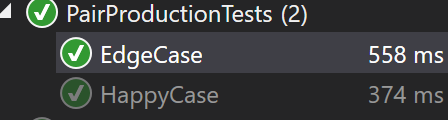
* Normal –



* Edge –



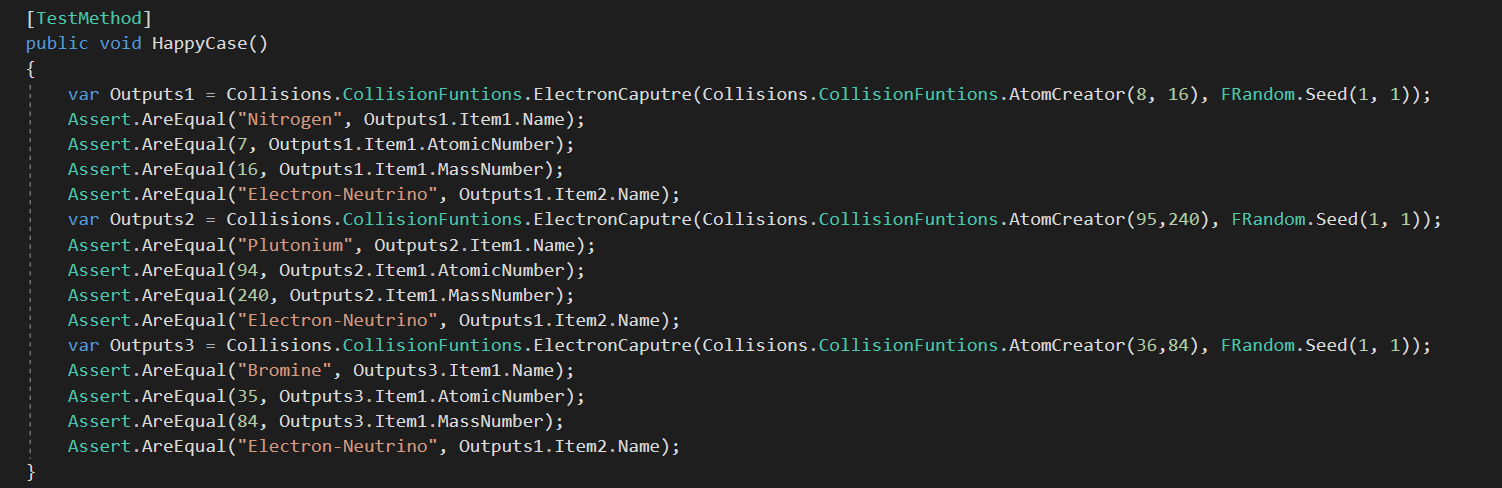
* Error –
* Tests Passing -



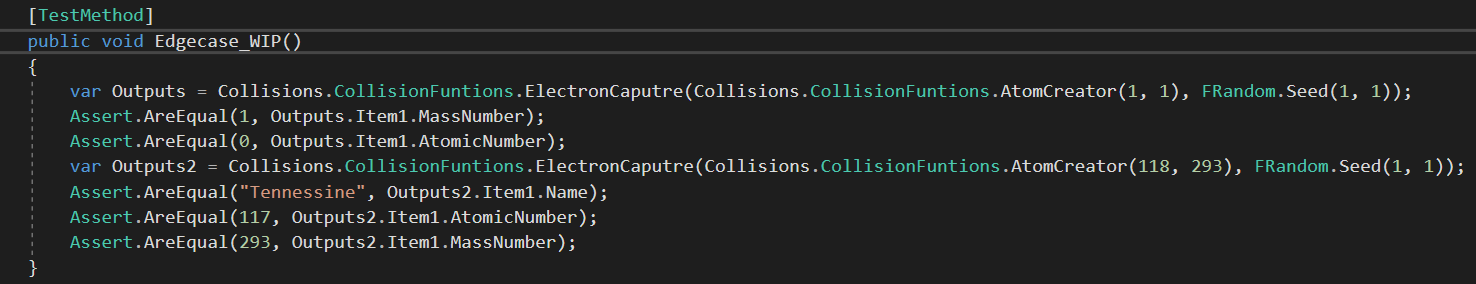
##### ElectronCaputre

public static Tuple<Atom, Particle> ElectronCaputre(Atom Atom, FRandom Rand)

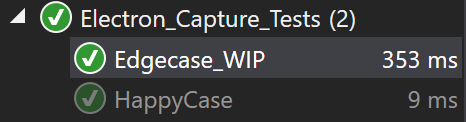
* Normal –



* Edge –



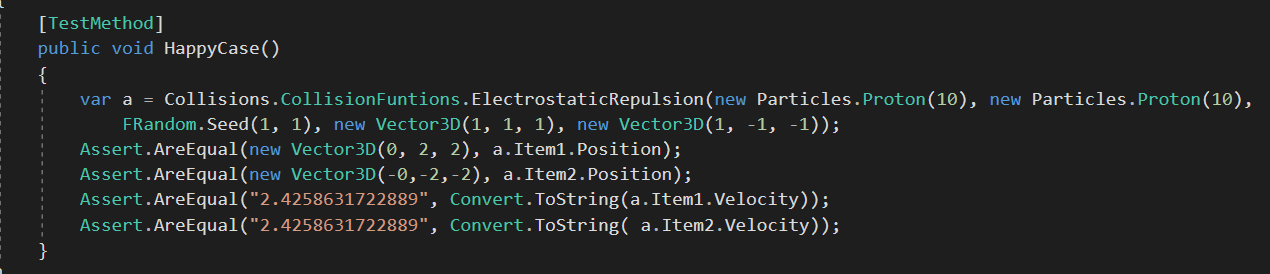
* Error –
* Tests Passing –



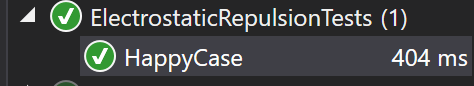
##### ElectrostaticRepulsion

public static Tuple<Particle,Particle> ElectrostaticRepulsion <T>(T P1, T P2, FRandom Rand, Vector3D StartingPositionParticle1, Vector3D StartingPositionParticle2) where T : Particle

* Normal –



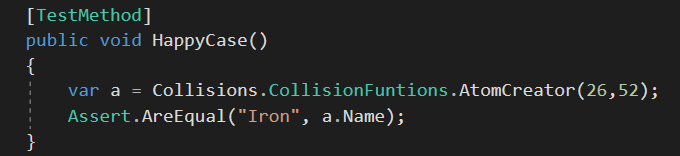
* Edge –
* Error –
* Tests Passing -



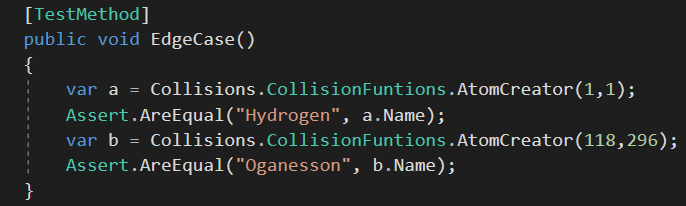
##### AtomCreator

public static Atom AtomCreator(int AtomicNumber, int MassNumber)

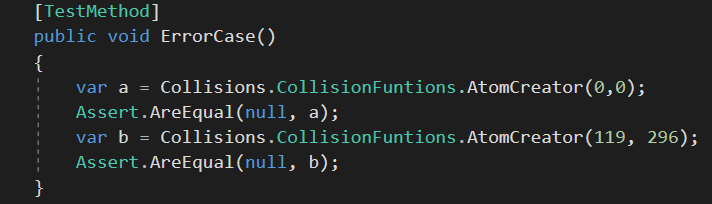
* Normal –



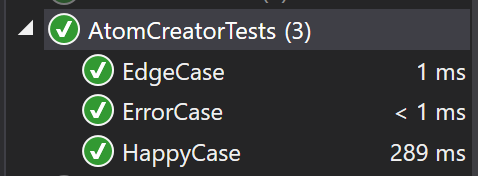
* Edge –



* Error –



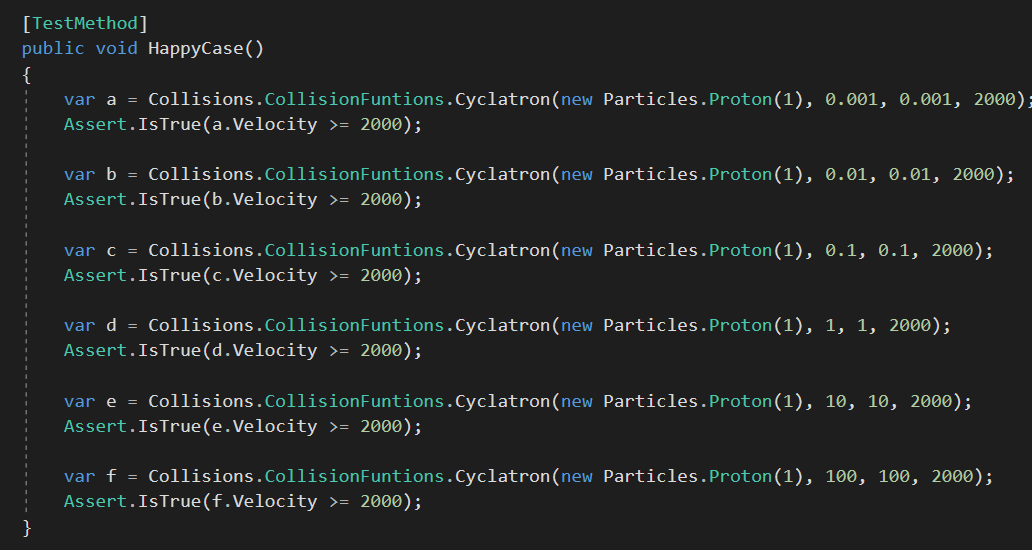
* Tests Passing –



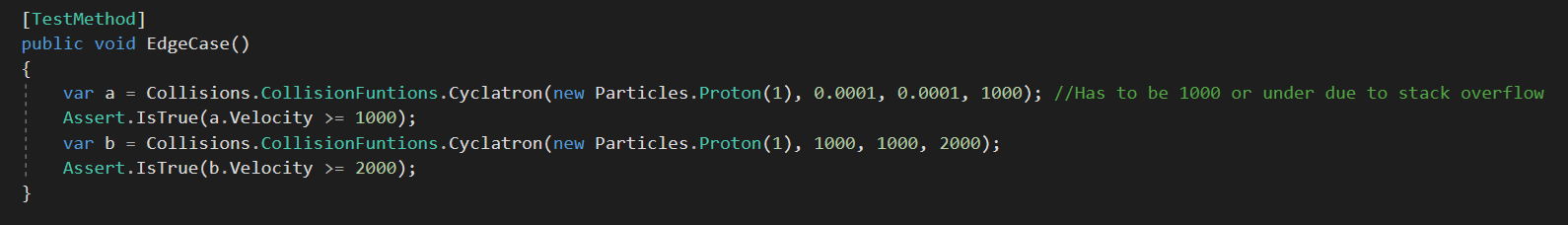
Cyclotron

public static Particle Cyclatron<T>(T P, double E, double B, double RequiredV) where T: Particle

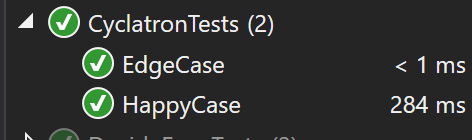
* Happy –



* Edge –



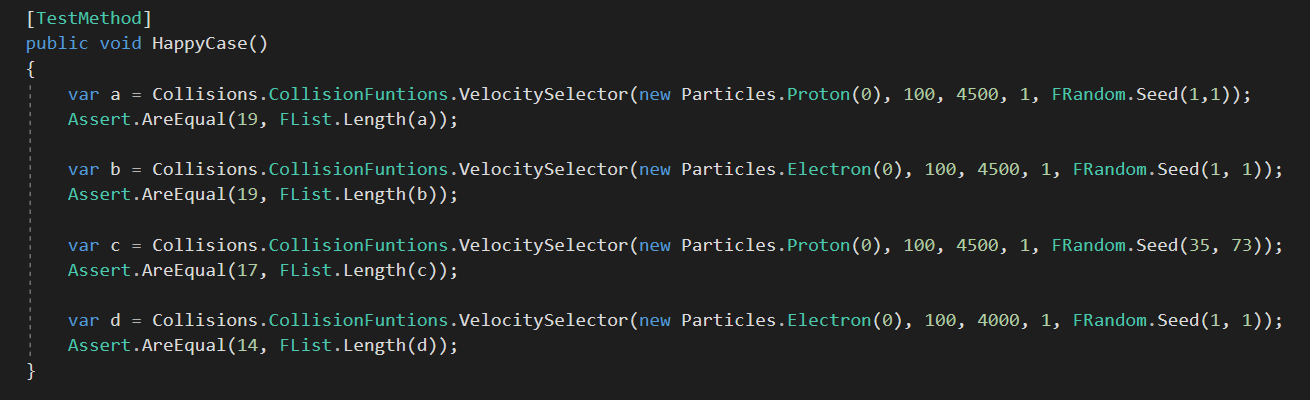
* Error –
* Tests passing –



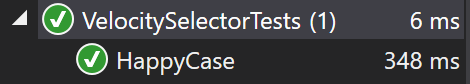
Velocity Selector

public static FList<Particles.Particle> VelocitySelector<T>(T Particle, int NumberOfparticles, double E, double B, FRandom Rand) where T : Particle

* Happy –



* Edge –
* Error –
* Tests passing –



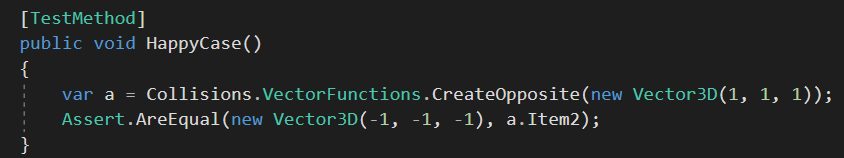
### Vector Functions

#### Lowest Functions

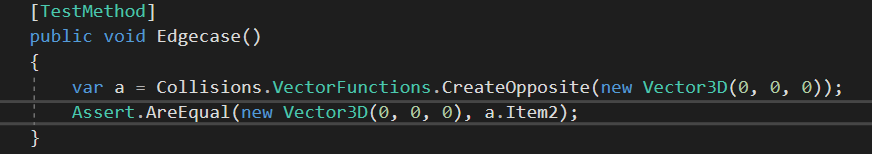
##### CreateOpposite

public static Tuple<Vector3D,Vector3D> CreateOpposite (Vector3D V)

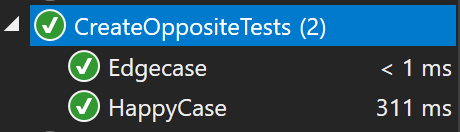
* Normal –



* Edge –



* Error –
* Tests Passing–

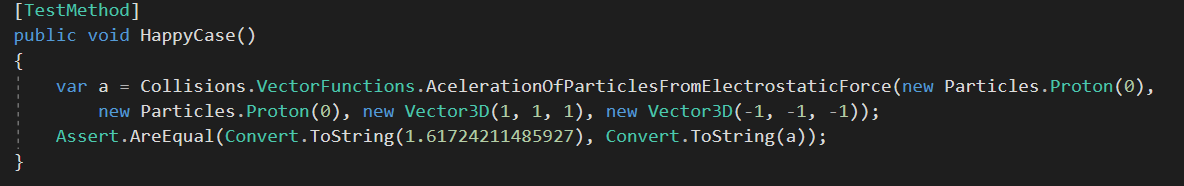


**Middle Functions**

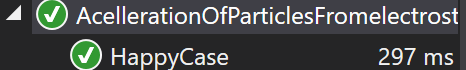
AccelerationOfParticlesFromElectrostaticForces

public static double AcelerationOfParticlesFromElectrostaticForce<T>(T Particle1, T Particle2, Vector3D StartingPositionParticle1, Vector3D StartingPositionParticle2) where T : Particle

* Happy –



* Edge -
* Error -
* Tests Passing –

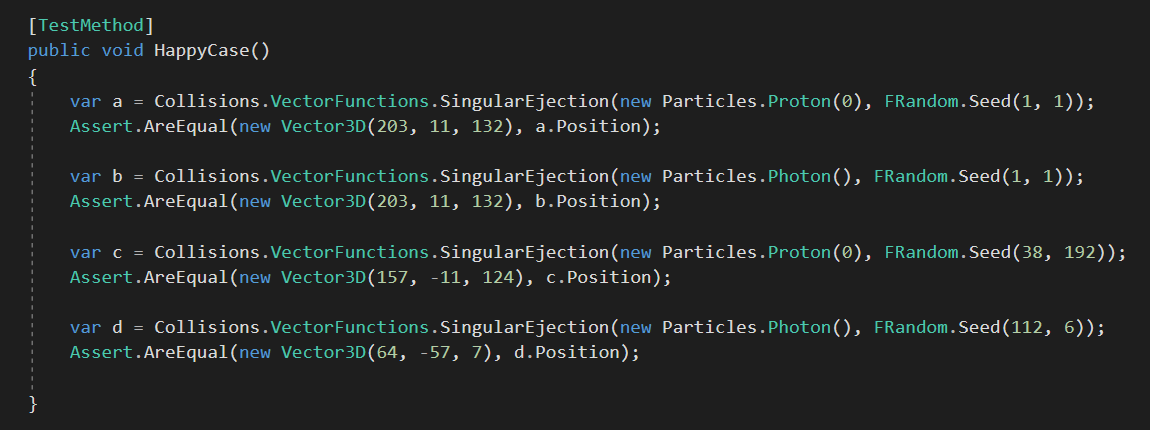


#### Heights Functions

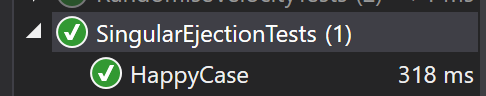
##### SingularEjection + SingularEjectionPhoton

public static Particle SingularEjection<T>(T p, FRandom Rand) where T : Particle

* Normal –



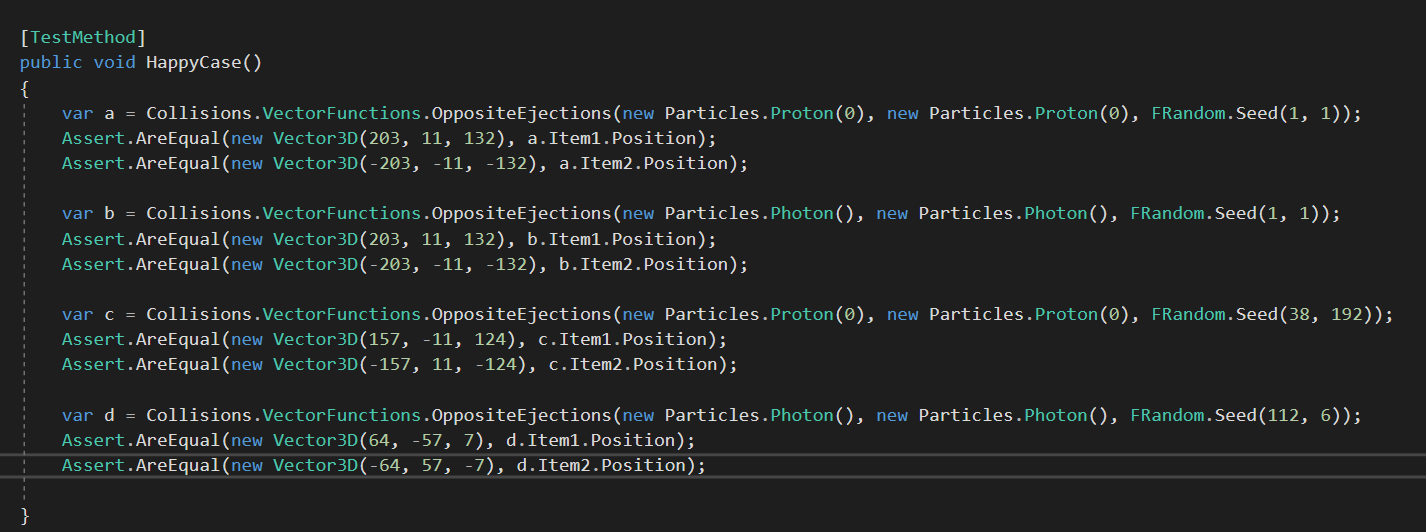
* Edge –
* Error –
* Tests Passing



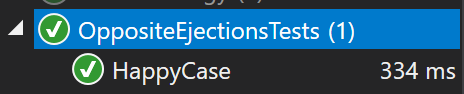
##### OppositeEjections + OppositeEjectionsPhoton

public static Tuple<Particle,Particle> OppositeEjections<T> (T P1, T P2, FRandom Rand) where T : Particle

* Normal –



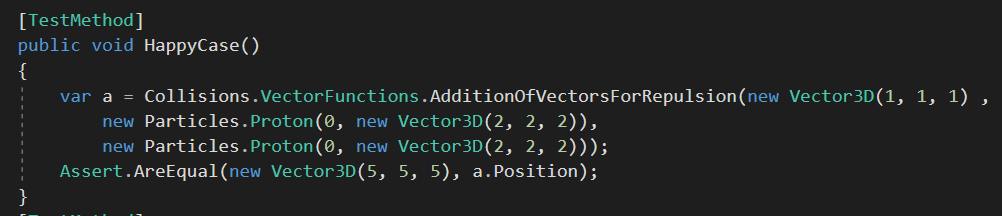
* Edge –
* Error –
* Tests Passing -



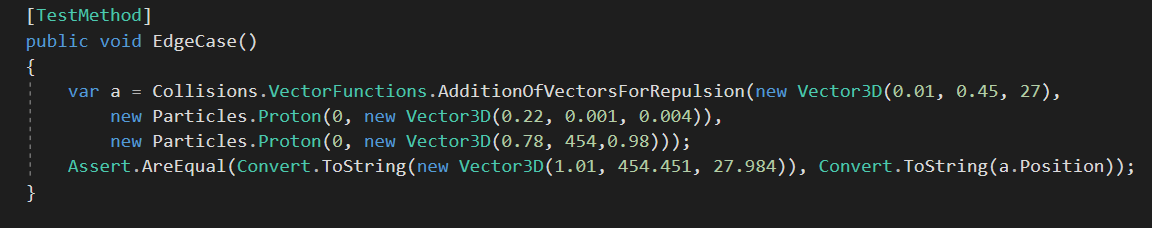
##### AdditionOfVectorsForRepulsion

public static Particle AdditionOfVectorsForRepulsion <T>(Vector3D StartingPosition,T P1, T P2) where T : Particle

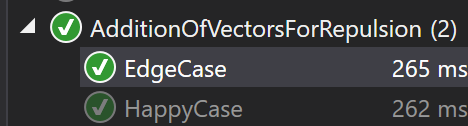
* Normal –



* Edge –



* Error –
* Tests Passing –



## Summery Of Testing

Due to one of that major benefits of functional being the ability to use unit tests to their full capability a major part of the testing comprises of unit tests. The testing done throughout the program covers most bases of possible errors, however there are still some user errors that I may not have thought of that might slip through. The extensive error handling on the UI is there to safeguard the core of the program where it is much harder to implement the error handling required because of its functional nature, therefore the “dirty” outside does a lot of the work on this part and as it is a UI is not functional at all. The functions using a random value are seen as deterministic for the tests however more extensive testing could have been done to show the range of randomness, I decided against this as I wanted to spend time covering a lot of different areas of tests, throughout the core of the program instead of focusing on a single small area. Due to the UI taking a lot of the error handling not all of the parts for the unit tests are filled out. This is because the scenarios required for that test to happen is blocked higher up in the UI layer, negating the need to test it in the functional layer. Some tests did show some errors in the program that were unforeseen beforehand such as when iterating through the cyclotron and velocity selector tests if too many particles are simulated there is a stack-overflow error which reduces the effectiveness of the simulations s more particles would be passed through the equipment that I can simulate. Overall the breadth of testing allows the whole program to be designed to a high standard whilst some more in-depth testing is done on the UI error handling helps to guide the user through it.

# Evaluation

How well did you meet your own objectives?

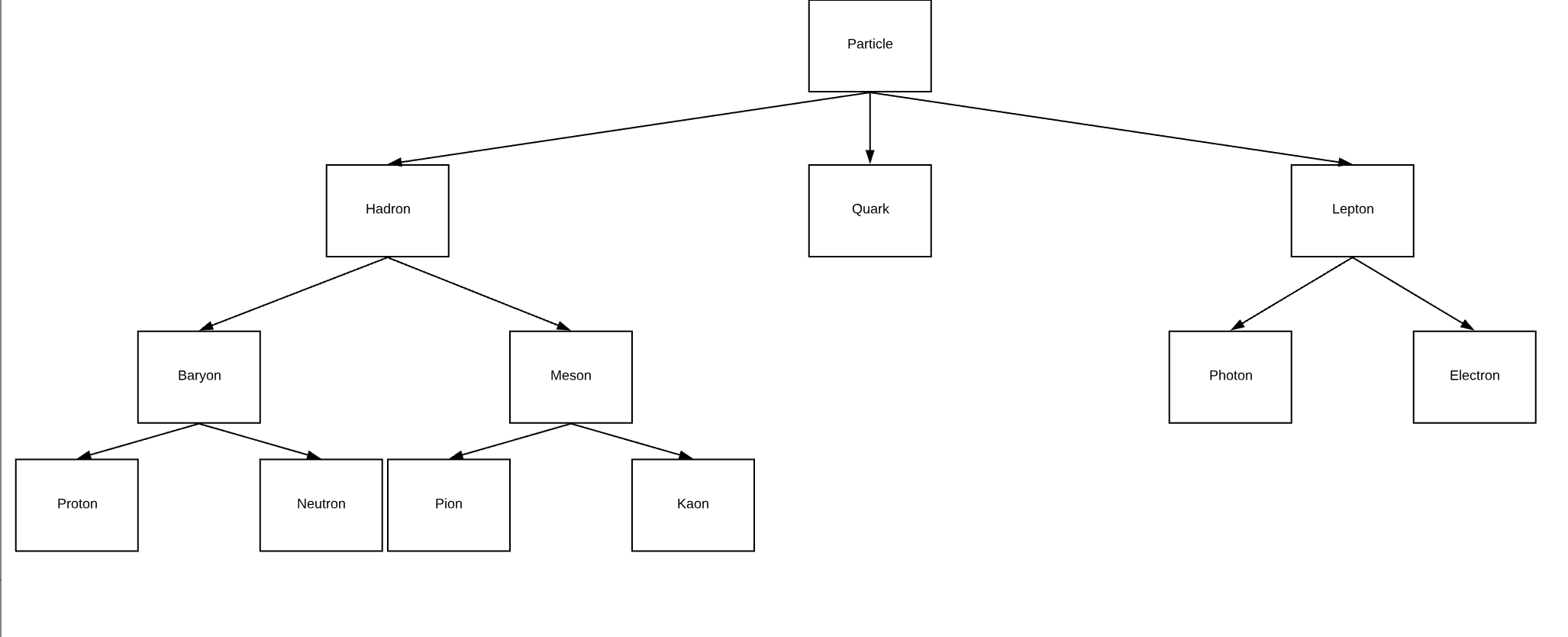
Evidence of *real* feedback from stakeholders and/or real users.?

Evidence of objective ‘reflection’ on your own project.

Further directions.

References

# Appendix 1



# Appendix 2