Project Deliverable

**Tanzanite Engine**

*Team Members:*

Ian Cuba

Troy Gittelmacher

Matt Laws

Devan Patel

Wilson Yeung

December 8th, 2014

Learning Experience

**Ian Cuba**

In my opinion this was the best and highest quality assignment from this course. It got us the students engaged and quite active in designing and thinking about design patterns in a real world scenario. The sheer amount of debate about design and patterns facilitated my learning greatly. This assignment was a good exercise about cooperation in design a system and we all being so new to design patterns still made for great debate which lead us all to a higher understanding of the patterns. 10/10 would do again.

Honor Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Troy Gittelmacher**

Overall this project was a great demonstration of how to tackle problems in application and framework design using Design Patterns. As the framework was being built, it was easy to see the flexibility of the design in allowing growth and addition of features to the framework, as well as flexibility in which applications could utilize the framework. Learning more about evolutionary computing was also a useful learning experience, as it is a field that has many applications in computer science that I did not have any prior experience in dealing with.

As a cumulative measure of what we have learned in Design Patterns, this assignment demonstrated that design patterns should not be used just because they exist. We had to weigh the benefits and drawbacks of using certain patterns during design, and making these decisions is what this class is all about, in my opinion.

Honor Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Matt Laws**

From this assignment I gained a good understanding of what evolutionary computing is. I also learned what it takes to design a Framework, and that it’s not an easy undertaking. The hardest part of the assignment was designing the initial framework. Ideas were bounced around for a while until we eventually came up with a general consensus of what our design should consist of. Preparing the deliverables was the easiest part.

Honor Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Devan Patel**

After completing this project, I feel like I have a much better understanding of design patterns and how to integrate one another to create larger enterprise-like applications. It was interesting to learn about evolutionary computing systems and seeing how many different fields are gradually migrating to utilizing them. I feel like the hardest part with the project is designing the framework. There were many questions that needed to be hashed out and after drawing the UML for it, the applications were much easier. The easiest part was communication. I feel like the team had great communication which made the project creation much easier.

Honor Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Wilson Yeung**

From this project, I learned about evolutionary computing and how we could create an application that supports a wide variety of different implementations. I believe that the learning objective was to give the students an opportunity to apply design patterns into a mock application framework and to emulate a real world scenario to challenge the students. The hardest or most time consuming part was creating the different UMLs and fitting them all together. Applying design patterns to the framework too time to discuss the benefits and the consequences if we wanted to implement it. The easiest part of the project was thinking of applications since we had to look at applications to know what evolutionary computing was.

Honor Code \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[Introduction / Background of Evolutionary Computing](#h.512idapmfjf9)

[The Tanzanite Engine - Context](#h.rmwzch2jzi6b)

[Overview of features](#h.u6xu1erg7bc5)

[Basic structure of framework](#h.sot0u9ik1v4e)

[How to Use it](#h.idye5mwz4y6h)

[Internals Documentation and Exposition](#h.arffcwan37lk)

[UML’s](#h.214vs03oercd)

[Overview Tanzanite Engine](#h.zbgdwmkwjc0b)

[PopulationGenerator](#h.843uankj7zvd)

[FitnessTester](#h.jw1vi1wlbf79)

[RNGsusMachine](#h.4jx0k7p1pljd)

[Evolver](#h.b318b1b9wkkf)

[ContinuationDeterminer](#h.mqeprguabh3v)

[Basic API](#h.bjxv83s45xny)

[TanzaniteEngine](#h.emx553xausgc)

[Fields](#h.lbmsgw9pbsbm)

[Method Overview](#h.30y3kwo3bko2)

[Mutatable](#h.gcuuhpjxl939)

[Method Overview](#h.1ncwsqdvfb49)

[FitNode](#h.6i43eahh99c0)

[Fields](#h.q4fxztwkeg0w)

[Method Overview](#h.xpjegipbuhs3)

[ModuleFactory](#h.qy2r7q36fuby)

[Fields](#h.wzd97jrqs3f5)

[Method Overview](#h.r4plogdxmy41)

[PopulationGenerator](#h.qcmhuwspvah2)

[Fields](#h.1u376u758raw)

[Method Overview](#h.2qj0doeg67s3)

[DataAnalyzer](#h.z08jazmq4p8x)

[Fields](#h.sy78k3dbbrmk)

[Method Overview](#h.cy3xps706vxy)

[FitnessTester](#h.ym6wti5bq4d3)

[Method Overview](#h.re09gsobq5tx)

[RNGSusMachine](#h.6pjexoqb3n2s)

[Method Overview](#h.suewcupo1w2w)

[Evolver](#h.2wy95shv3xql)

[Method Overview](#h.vmqqyxjqmxd2)

[ContinuationDeterminer](#h.l9yig29b774b)

[Method Overview](#h.wbhgv8exjbtv)

[Use of Patterns analysis](#h.lt8dubrtqrgn)

[Façade Pattern](#h.3ndud8yxl0j4)

[Simple Factory](#h.va0wid9kvoxd)

[Bridge](#h.1jjdivu3qk9q)

[Adapter](#h.3z2o9yvcrqw7)

[Iterator](#h.f0ceoctpi0fj)

[Strategy](#h.h7bnpy7y06mu)

[Observer](#h.dwu2pyl8qhzy)

[Singleton](#h.5oxvdqneswix)

[Application 1: ImageEvolverApp](#h.kapjr4dv8m07)

[Overview](#h.afh1q2awyoll)

[Relevant Classes](#h.un3dxy4c8dbc)

[ImageEvolverApp](#h.2y83bpni03z6)

[ImageComponent](#h.ebcu00vx3fwv)

[ImageLeaf](#h.xgqhtya5cje6)

[ImageComposite](#h.1xzsw6h0nfvk)

[ImageDecorator](#h.7es8o9k7u34d)

[Settings](#h.t3dr57nu9qvz)

[ImageComponentFactory](#h.8ltp8dnks9ja)

[ImageConverter](#h.tfk0ghe5bokb)

[ImageGatherer](#h.gi888z6kvi2g)

[UML Diagrams](#h.9busq9bu5sh9)

[ImageEvolverApp](#h.zaby8i85xos)

[ImageComponent](#h.i47tv14q13nh)

[ImageLeaf](#h.792curhciugm)

[ImageDecorator](#h.fe8efdpbv42s)

[FilterDecorator](#h.38dc62b0uoe7)

[Settings](#h.78trclrxlook)

[ImageComponentFactory](#h.8vrpn5auyiz9)

[ImageConverter](#h.rg5a3wsso0xs)

[ImageGatherer](#h.3jbwdgph28yd)

[Mock Output](#h.af631dcio6lo)

[Application 2](#h.yeutjsi11m7f)

[CipherComponent (abstract) These are blocks](#h.jcu17c965xn3)

[CipherLeaf (abstract)](#h.c556emy6f05i)

[CipherComposite](#h.4kh036zh02vi)

[CipherDecorator (abstract)](#h.4wjfur2twj8b)

[Settings (abstract)](#h.ujxpsb1h8wjv)

[CipherComponentFactory (abstract) (Blocks)](#h.xwg41z6af7ko)

[CipherDecrypter (abstract)](#h.tqvwcqcauaxf)

[CipherGatherer (abstract)](#h.xs523hk89rft)

[CipherEvolverApp](#h.a5pfqexu4pkw)

[CipherComponent](#h.c4fwxsrrqraz)

[CipherLeaf](#h.r6qf36i58opg)

[CipherDecorator](#h.80vui1w2yu3l)

[Settings](#h.dkcogj1tnxst)

[CipherComponentFactory](#h.d2tkjfof8oxt)

[Powerpoint Slides](#h.ttmxbitnaexb)

# Introduction / Background of Evolutionary Computing

Evolutionary Computing is the process of using ideas from biological evolution in the computing field to evolve a subject, or set of subjects, in hopes to eventually develop a superior subject.

The process consists of a generation of subjects evolving into a new generation. During this evolutionary process, some subjects die off, while others advance to the next generation. The survivors may undergo a change in a few ways, by mutating, exchanging traits with another subject, or reproduction. At the end of the cycle new subjects may be introduced to the population to complete the generation. This continues on in hopes of finding a superior subject.

It is important to note that Evolutionary Computing does not typically rely on a subject’s fitness to determine whether it makes it to the next generation, that would make it similar to a greedy algorithm. Generations can get further from a solution before they get closer to one. In this way, Evolutionary Computing can attack a problem from a number of different angles, not solely those one expects to generate favorable results.

Surviving a generation also has an aspect of chance to it. The fittest subject(s) will not always live to see the next generation, though they do have a better chance of survival. Just like in the real world, the fittest subjects tend to survive, but sometimes they don’t make it due to external forces.

Our framework supports the use of all the main aspects of evolutionary computing to evolve a set of objects from one generation to the next, and hopefully after a series of generations, into a generation that contains an optimal subject.

# The Tanzanite Engine - Context

## Overview of features

Features of the framework include:

* Singular Engine to perform evolutionary operations on a Mutatable population
* Modular evolutionary components for specified operations on the population
* Mutatable Interface adapter for clients to use for conforming objects to the engine specification
* Data analysis tools to help display data, and see how the fitness of subjects changes from one generation to the next.

## Basic structure of framework

The framework consists of a single engine that manages all the evolutionary components of the system. The framework delegates processes to classes for a more modular approach in the evolution. The engine works with a Collection of Mutatable objects that is contained in the main class and passed by reference to each of the evolutionary modules.

## How to Use it

In order to use the Tanzanite Framework to evolve their object the user must have their object implement the Mutatable interface. This consists of a few methods that determine how to deal with altering the object, such as how to mutate attributes of the object or how to crossbreed the object with another.

The user creates a TanzaniteEngine to be used with their Mutatable object. The user will then specify parameters using the methods provided by the TanzaniteEngine. One of the required parameters that has to be set is the Mutatable Object or Collection to be evolved. If one of these parameters is not set, the engine cannot perform an evolutionary process without a concrete object. After the user has finished setting up the engine, they can run the evolve method. This will cause the engine to create all the components it needs to evolve the population and it will run until the stopping condition in the ContinuationDeterminer is satisfied.

The DataAnalyzer is notified during the evolutionary process to observe changes in each generation. The parameters that the user has set will dictate how the DataAnalyzer module works. After the evolve method is finished, the DataAnalyzer will be safe to extract using the appropriate get method in the TanzaniteEngine.

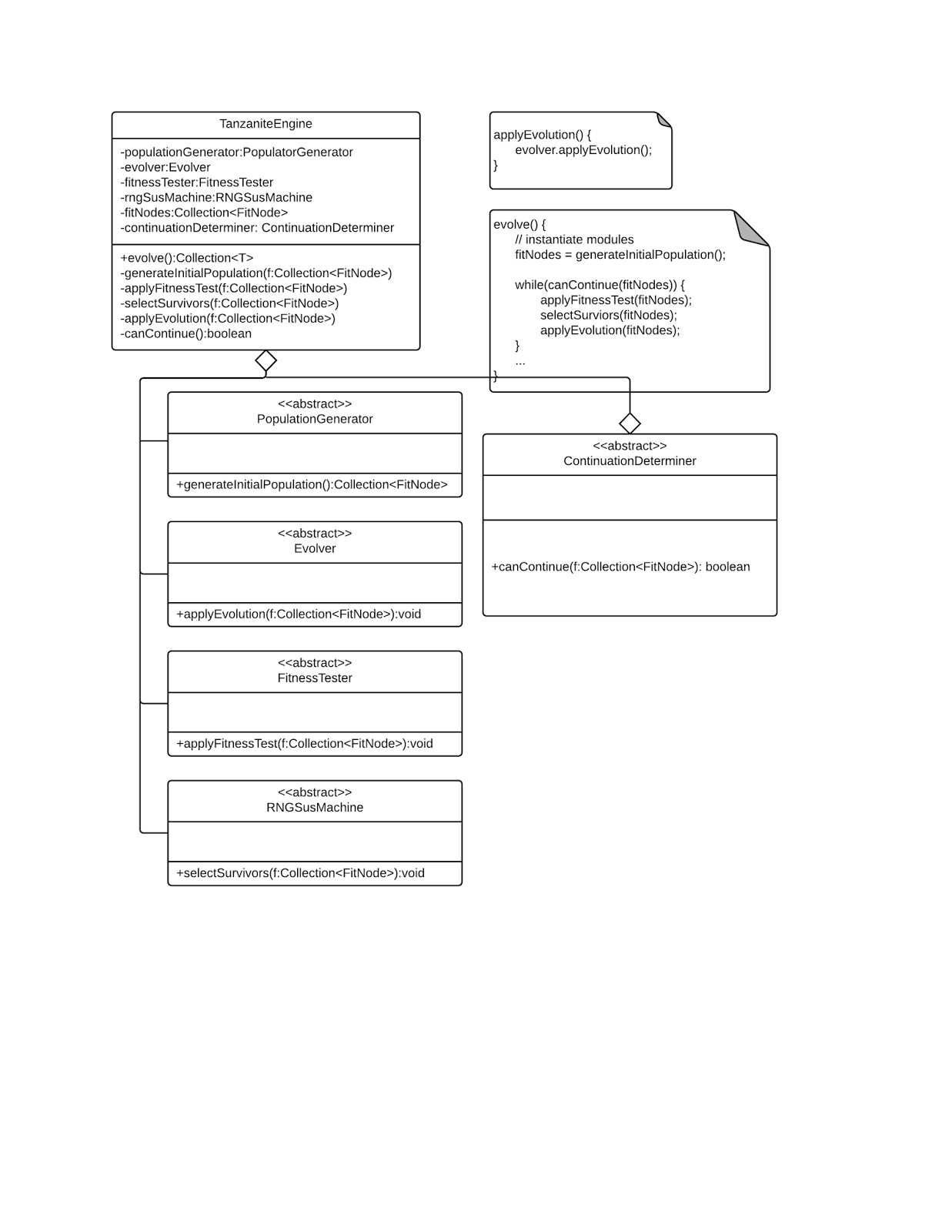
The evolve method will return a collection of the client’s object that have had the generational evolutions performed on it.

# 

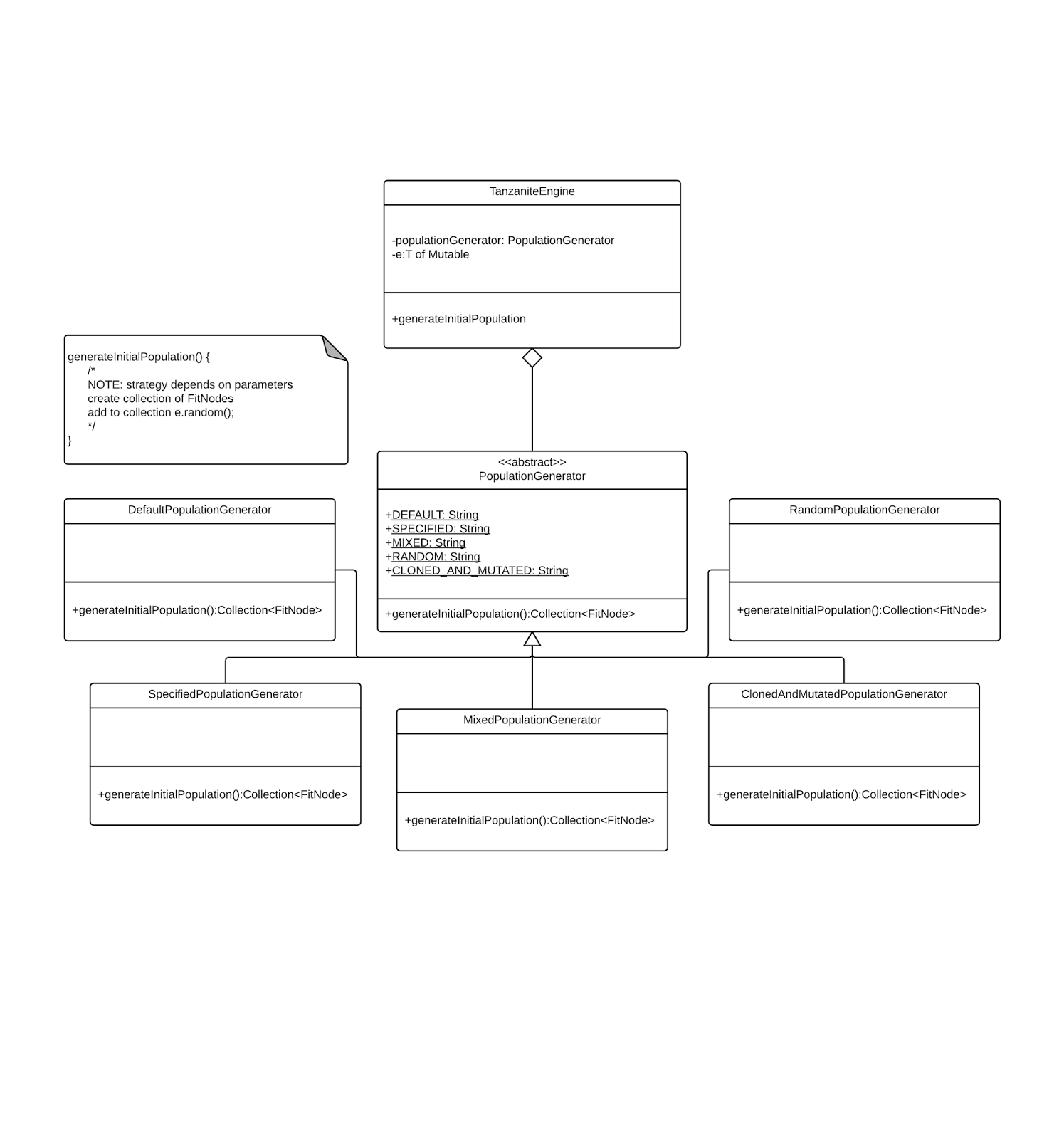
# Internals Documentation and Exposition

## UML’s

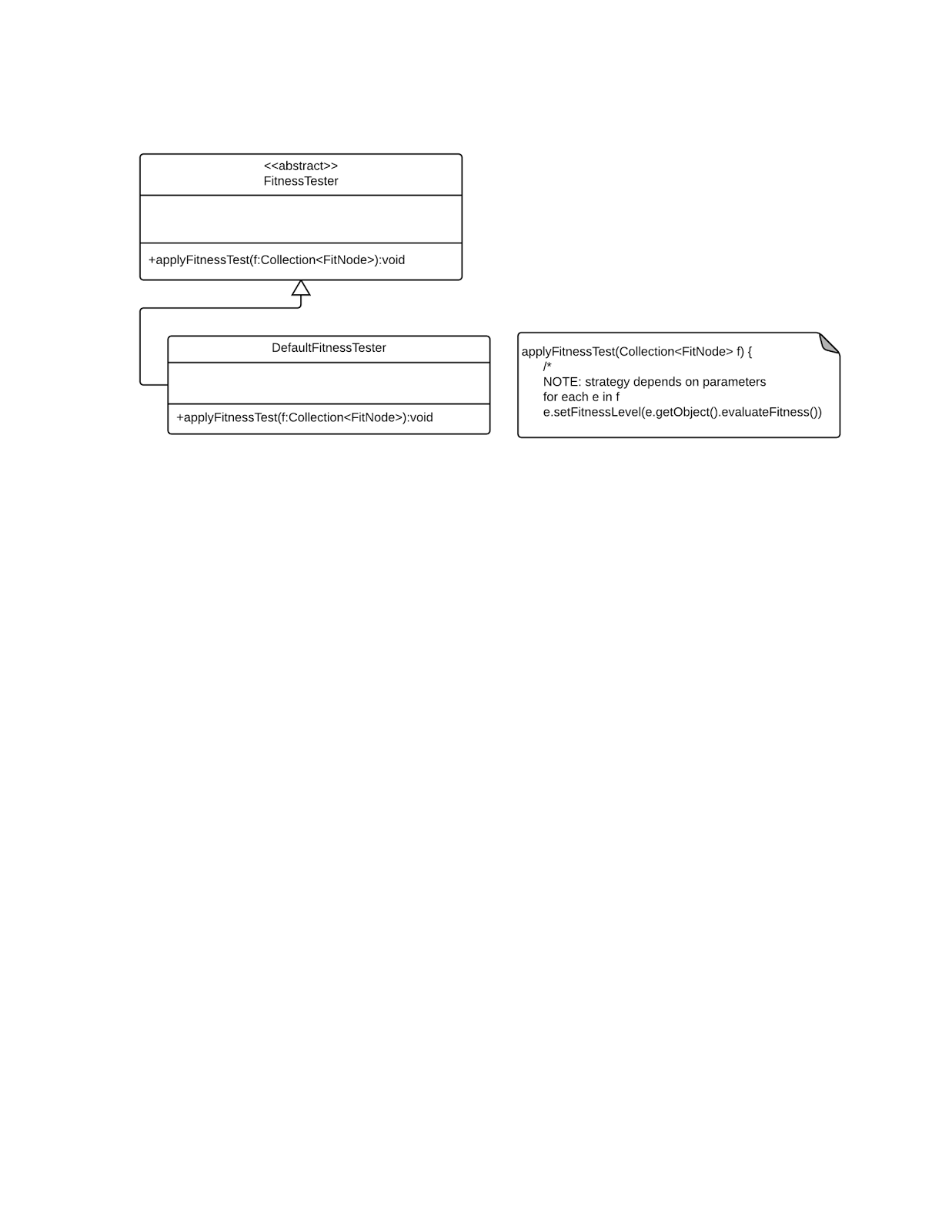
### Overview Tanzanite Engine



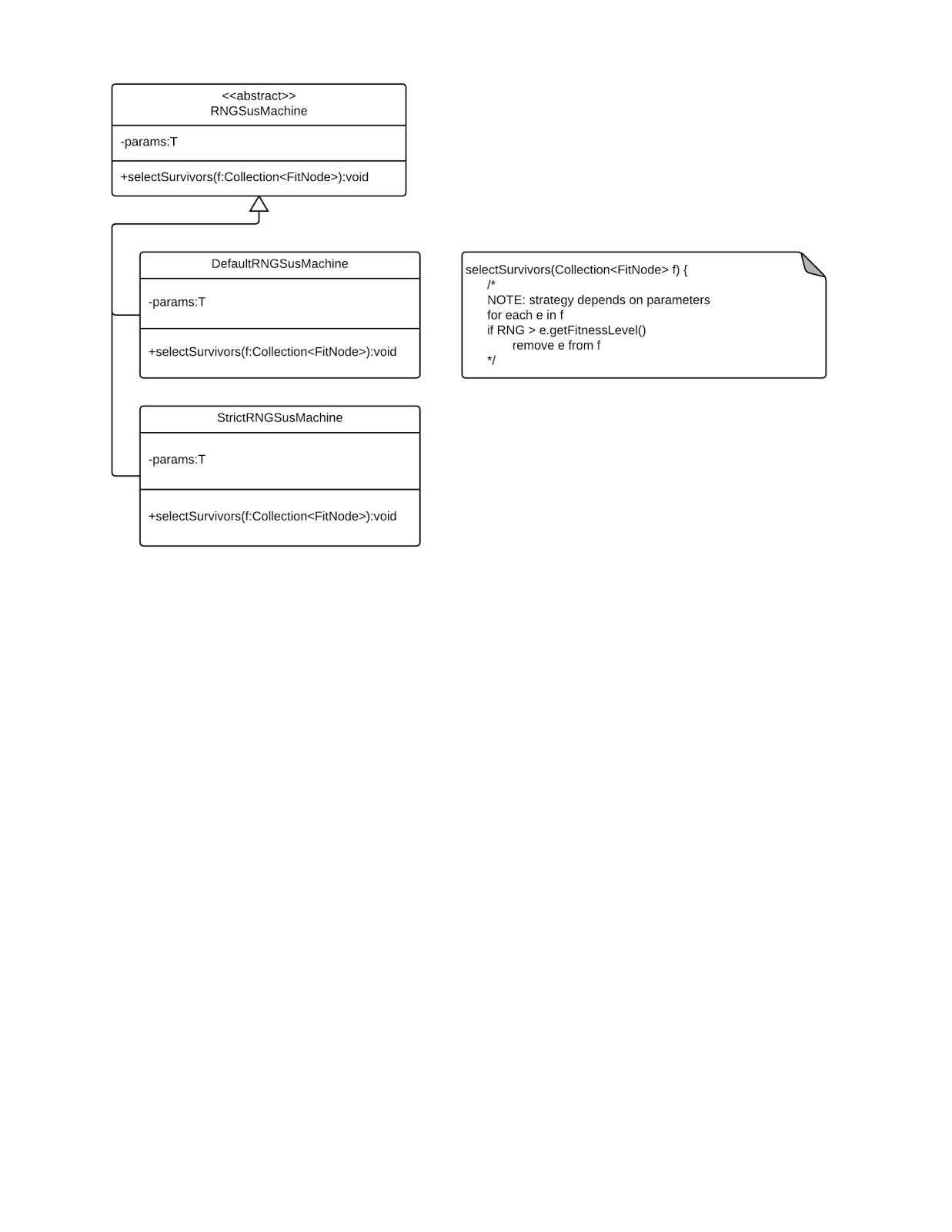
### PopulationGenerator



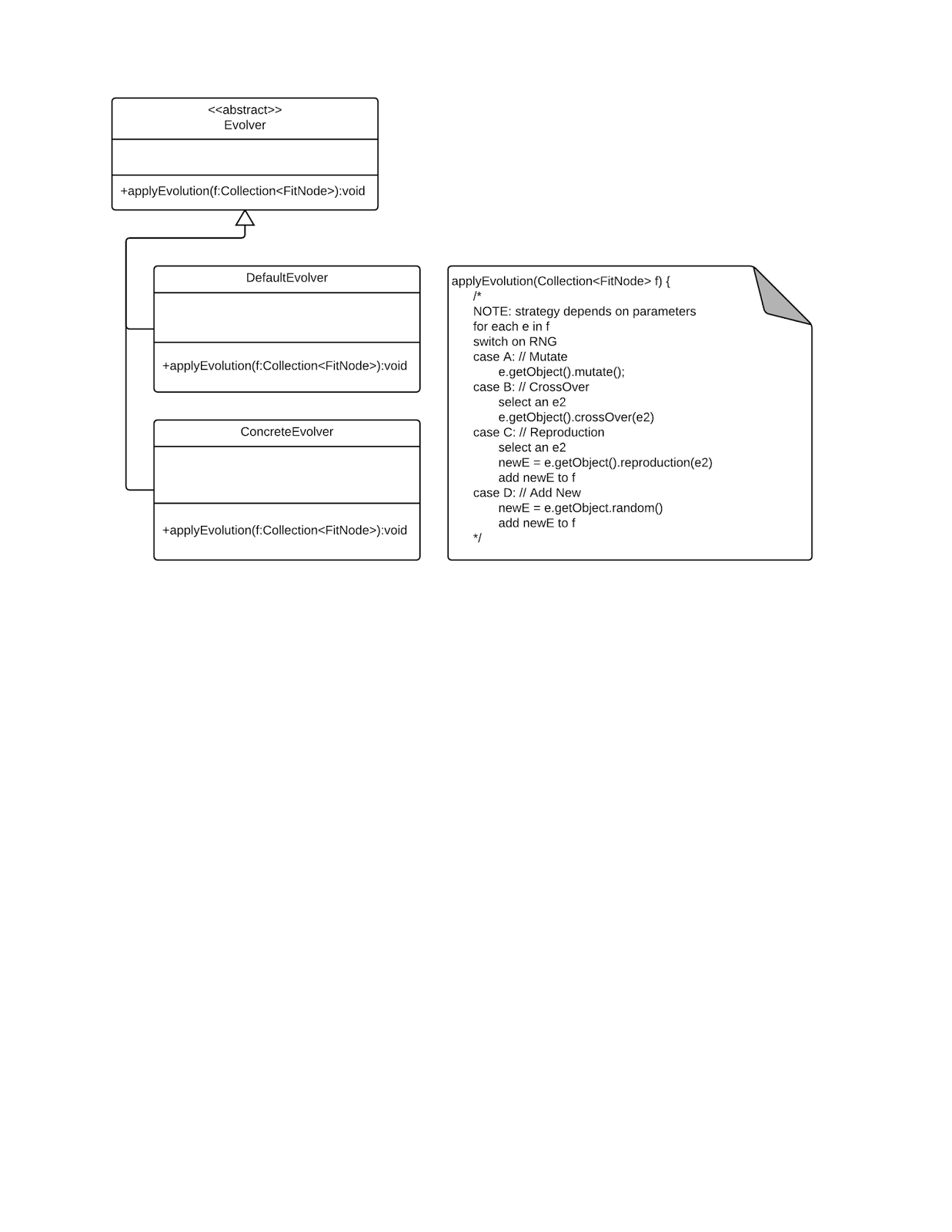
### FitnessTester



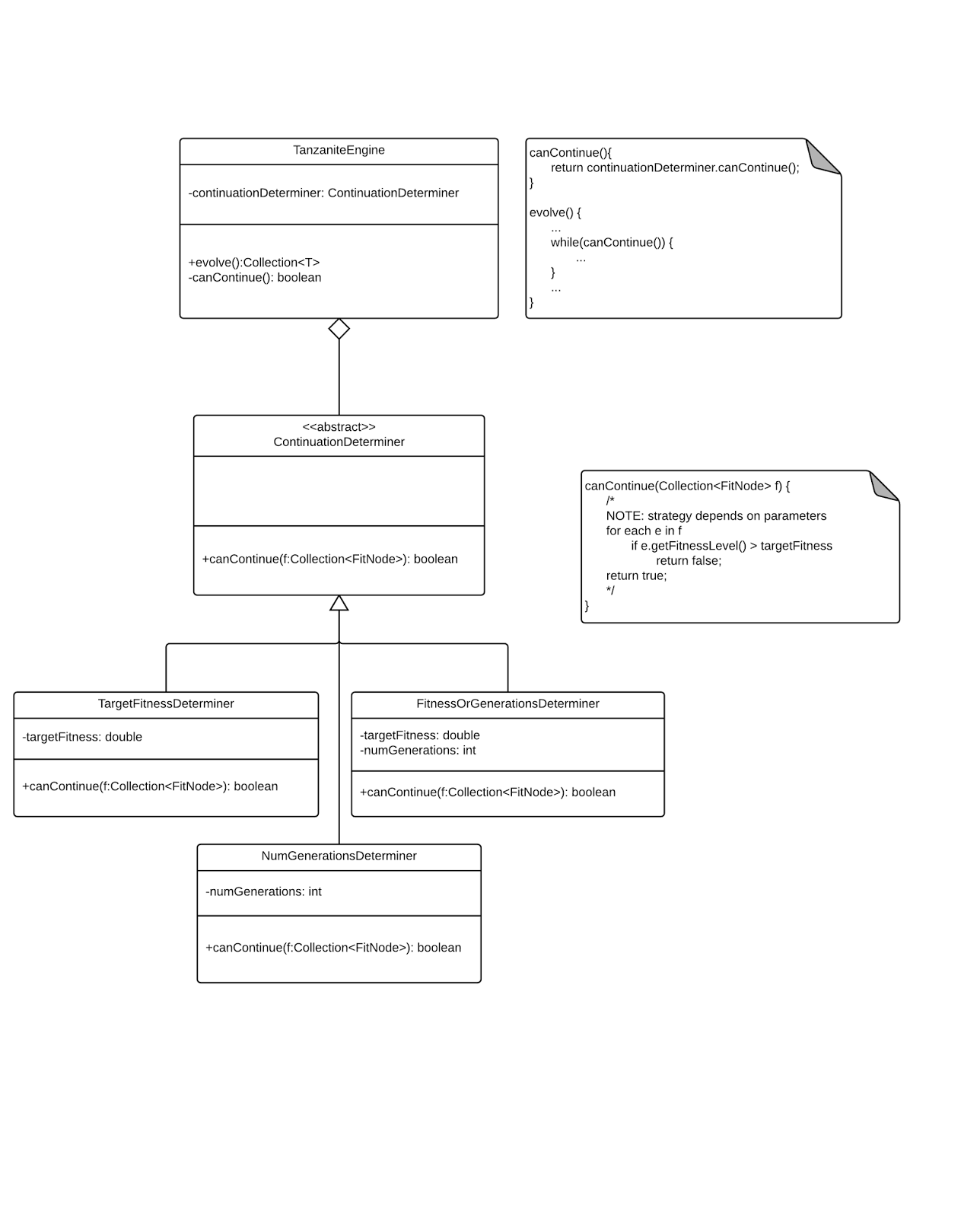
### RNGsusMachine



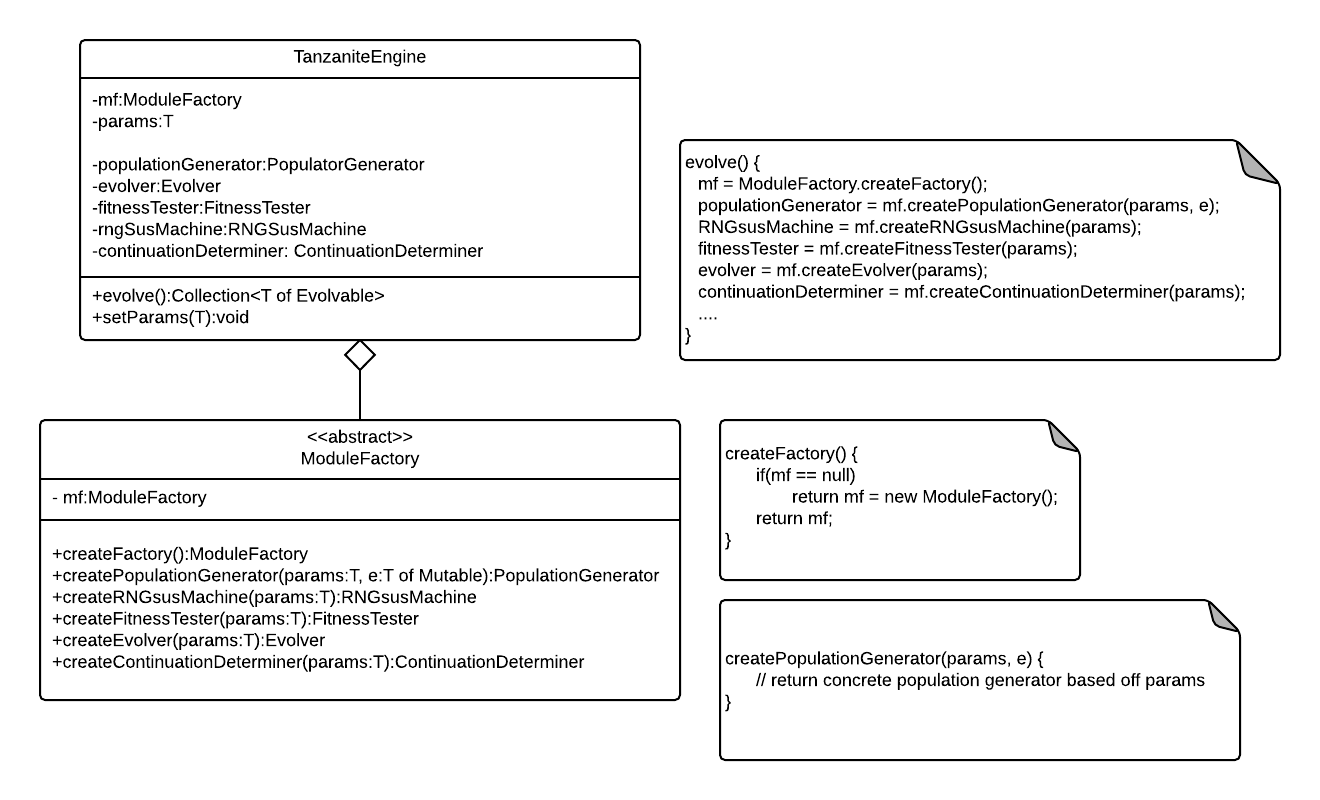
### Evolver



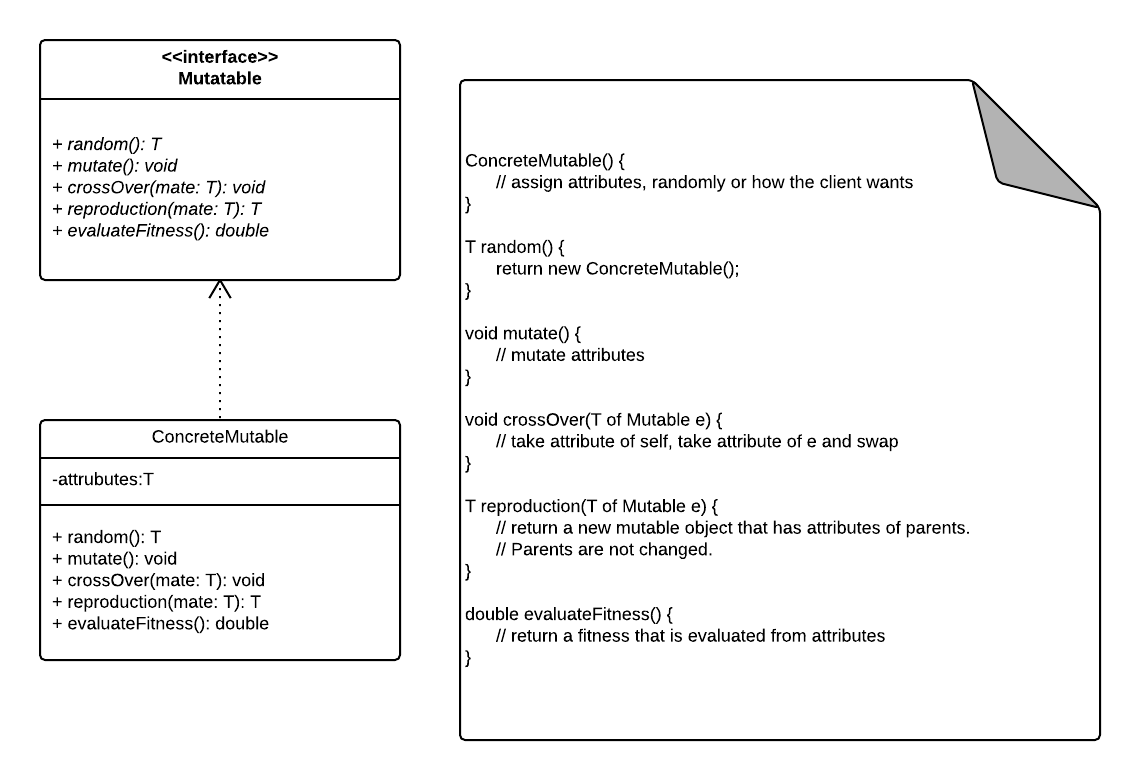
### ContinuationDeterminer



**ModuleFactory**



**MutatableInterface**



## Basic API

### TanzaniteEngine

#### Fields

**private populationGenerator:PopulationGenerator**

* The object used to generate the initial population and replenish the population

**private evolver:Evolver**

* Used to iterate through and mutate the subjects

**private fitnessTester:FitnessTester**

* Used to iterate through and evaluate fitness on the subjects

**private rngSUSMachine:RNGSusMachine**

* Used to determine what subjects survive to the next round randomly

**private fitNodes:Collection<FitNode>**

* Stores the subjects with their fitness level

**private continuationDeterminer:ContinuationDeterminer**

* Used to determine when to stop the evolutionary process

**private dataAnalyzer:DataAnalyzer**

* Used to extract and hold the evolutionary data

**private moduleFactory:ModuleFactory**

* Used to extract and hold the evolutionary data

#### Method Overview

**public evolve():Collection<T>**

* The only public method of the Engine. Once called the Evolutionary Components of the Engine are instantiated. It then starts the evolution process and runs until the ContinuationDeterminer is satisfied. Returning a Collection<T> containing the optimal solution object.

**private generateInitialPopulation(f:Collection<FitNode>)**

* Generates the initial population of the subjects as FitNodes. These FitNodes store the subject as well as a double to hold the subjects fitness level.

**private applyFitnessTest(f:Collection<FitNode>)**

* Evaluates fitness on all the subjects by iterating through them and running evaluateFitness().

**private selectSurvivors(f:Collection<FitNode>)**

* Called on the RNGsusMachine to determine what subjects make it to the next generation.

**private applyEvolution(f:Collection<FitNode>)**

* Called on the Evolver which iterates through the surviving subjects and alters them in various ways.

**private canContinue():boolean**

* Called on the ContinuationDeterminer which returns a boolean specifying whether the Engine can continue evolving.

**private getDataAnalyzer(): DataAnalyzer**

* Used to retrieve a DataAnalyzer for use in examining collected data.

### Mutatable

#### Method Overview

**public random():T**

* Method to be implemented by client. Used to return a new random Mutatable object.

**public mutate():void**

* Method to be implemented by client. Used to mutate attributes of itself.

**public crossOver(mate:T):void**

* Method to be implemented by client. Used to crossover attributes between Mutatable objects. This method changes attributes of itself and the Mutatable object that is passed as a parameter.

**public reproduction(mate:T):T**

* Method to be implemented by client. Used to create a new Mutatable object that has attributes of the parents.

**public evaluateFitness():double**

* Method to be implemented by client. Used to return a double fitness based upon the Mutatable objects attributes. The range of the fitness should be from zero to one.

### FitNode

#### Fields

**private object:T**

* Stores the clients Mutatable object

**private fitnessLevel:double**

* Stores the fitness level of the subject. which is null until evaluateFitness() is ran on the subject.

#### Method Overview

**public getObject():T**

* Returns the subject held within the FitNode container.

**public getFitnessLevel():double**

* Returns the subjects fitness

**public setObject(object:T):void**

* Sets the subject in the container

**public setFitnessLevel(fitnessLevel:double):void**

* Sets the fitness level of the subject in the container after evaluateFitness() is ran

**public normalize(): double**

* Normalizes the data to 0 (inclusive) and 1 (non inclusive) for later calculations.

### ModuleFactory

#### Fields

**private moduleFactory:ModuleFactory**

* Stores the singleton ModuleFactory object

#### Method Overview

**public createFactory():ModuleFactory**

* Returns a single instance of ModuleFactory. If the instance is already created, return that instance. If not, return a new ModuleFactory.

**public createPopulationGenerator(parameters):PopulationGenerator**

* Return a Concrete Population Generator based on the parameters.

**public createRNGsusMachine(parameters):RNGsusMachine**

* + Return a Concrete RNGsusMachine based on the parameters.

**public createFitnessTester(parameters):FitnessTester**

* + Return a Concrete FitnessTester based on the parameters.

**public createEvolver(parameters):Evolver**

* + Return a Concrete Evolver based on the parameters.

**public createContinuationDeterminer(parameters):ContinuationDeterminer**

* + Return a Concrete ContinuationDeterminer based on the parameters.

**public createDataAnalyzer(parameters):DataAnalyzer**

* + Return a Concrete DataAnalyzer based on the parameters.

### PopulationGenerator

#### Fields

**public static parameter strings**

* Used to determine the type of populationGenerator, the number of objects to create, etc.

#### Method Overview

**public generateInitialPopulation():Collection<Fitnode>**

* Generates the initial population given the parameters. The strategy that is used to create the collection and the type of collection that is returned is based off the parameters.

### DataAnalyzer

#### Fields

**private Map<gen : double, col: Collection<FitNode>>**

* Used to retain collected data generationally.

#### Method Overview

**public *Update(): void***

* Used to inform the DataAnalyzer to changes to fitNodes.

**private *sort(): void***

* Used to sort the Collection as desired. Client may use with a CustomAnalyzer to implement their own sort.

**private getAverageFit() : Collection<E>**

* Used to return a Collection of Objects that match the average fitness.

**private getHighestFit() : Collection<E>**

* Used to return a Collection of Objects that match the highest fitness.

**private getLowestFit() : Collection<E>**

* Used to return a Collection of Objects that match the lowest fitness.

**private getFitness(f: E): double**

* Used to determine the fitness of a specified object.

**private getNumberOfDeaths(gen: double): double**

* Used to determine the number of fitNodes that did not reach the final generation.

**private getNumberOfMutations(gen:double): double**

* Used to determine the number of object mutations that occurred.

**private getNumberOfCrossBreeds(gen: double): double**

* Used to determine the number of crossbred objects.

**private getNumberOfReproduced(gen: double): double**

* Used to determine the number of newly created objects.

**private getGeneration(gen: double): Collection<E>**

* Returns a collection of E objects from the specified generation.

**private getGenerationSize(): double**

* Used to determine the number of objects in a generation.

### FitnessTester

#### Method Overview

**public applyFitnessTest(f:Collection<FitNode>):void**

* Iterates through the collection of FitNodes and runs evalutateFitness on each one. It calls setFitness to update the fitness contained in the corresponding FitNode.

### RNGSusMachine

#### Method Overview

**public selectSurvivors(f:Collection<FitNode>):void**

* Iterates through the Collection of FitNodes and generates random numbers for each one. If the fitness is higher than the random number then the FitNode survives to the next generation.

### Evolver

#### Method Overview

**public applyEvolution(f:Collection<FitNode>)**

* Iterates through the Collection of survivor FitNodes and randomly decides to alter the contained objects structure by calling one of the methods defined in the Mutatable interface.

### ContinuationDeterminer

#### Method Overview

**public canContinue():boolean**

* Returns the condition on whether the Engine should continue evolving

# Use of Patterns analysis

## Façade Pattern

The TanzaniteEngine itself is essentially a Façade. It provides an easy way to interact with all the evolutionary components involved with the evolutionary process including the PopulationGenerator, FitnessTester, Evolver and RNGsusMachine.

## Simple Factory

The TanzaniteEngine has the user send in parameters to specify the type of evolutionary components it will create. Once the user runs evolve, then the specified objects are created using the ModuleFactory class. Depending on the parameters that are set, the ModuleFactory object will create the corresponding evolutionary component instances.

This is also an example of lazy instantiation, we don’t create the evolutionary components for the engine until they are first needed. This reduces the overhead of the resources used and doesn’t use up any resources until they will be used.

## Bridge

The TanzaniteEngine implements the different steps of the evolutionary process using the Bridge pattern. The algorithm calls the private methods declared within the engine and then each of those private methods calls its implementation from the evolutionary modules. An example of this is the private method applyFitnessTest that uses the implementation of the FitnessTester object to perform the operation.

## Adapter

The Mutatable interface is an example of the Adapter pattern. More specifically it is a Pluggable Object Adapter. It provides the engine with knowledge about how to deal with the users object, which is unknown during compilation.

## Iterator

We use the iterator pattern to iterate through the Collection of FitNodes in each of the evolutionary components.

## Strategy

There are a few examples of the Strategy Pattern in the Framework. The implementation of the actual method is deferred to the concrete instances of the evolutionary components. Each subclass of the component has a different implementation of the overall abstraction of the method.

The ContinuationDeterminer is an example of the Strategy Pattern. It allows the user to choose what to base the end condition of the evolution process on. The user can choose to have the engine stop after a certain number of generations or after reaching a desired fitness level. Based on their choice of stopping condition, the corresponding ConcreteContinuationDeterminer will be instantiated and used.

The PopulationGenerator is also an example of the Strategy Pattern. The user can specify how it wants the population to be created, and the corresponding ConcretePopulationGenerator will be used to generate the initial population.

## Observer

The data analysis portion of the Framework utilizes the Observer pattern. When the RNGsusMachine has determined the survivors of a generation. It notifies the data analyzer which then updates itself based on the FitNodes of the next generation. Observer was appropriate in this instance because it decoupled the analysis of the evolution process from the engine itself, which already has many responsibilities.

## Singleton

ModuleFactory is an example of the Singleton Pattern. Only one ModuleFactory is needed to instantiate all the evolutionary components that work with the engine or multiple engines. It solved the potential problem of having more than one of ModuleFactory made if the user calls evolve multiple times using different engines.

# Application 1: ImageEvolverApp

## Overview

The application, called ImageEvolverApp, is an image building framework that allows for creation of ImageComponents, which are elements of an image that can be represented on a screen, such as brush strokes, shapes, gradients, and other visual elements. These visual elements can have various filters applied to them and other rendering effects to modify overall appearance.

This application is not intended for solely using the Tanzanite Engine. Indeed, the internals of ImageEvolverApp are reusable by any application, so other applications, such as a canvas type application like Photoshop, can utilize the framework to draw images and alter them as they desire. However, in the context of using the Tanzanite Engine, the object that composes the images implements the Mutatable Interface necessary to work with the engine.

Because the visual elements as well as the filtering effects inherit from the same ImageComponent object, evolution is supported to a very high degree because all components can implement Mutatable differently, to account for a large possibility pool of results with a given starting generation.

## Relevant Classes

### ImageEvolverApp

This class provides the mainframe for utilizing the Image building framework, along with utilizing the Tanzanite Engine to provide a means for evolution.

Subclasses

Non subclasses are intended for ImageEvolverApp.

Patterns

* Façade: This class provides an interface for interacting with the image building framework subsystem as well as the evolution framework, and is thus a Façade.

### ImageComponent

The highest level building block for images. All physical drawing components as well as filters and rendering effects are treated as ImageComponents. This allows for customization of the implementation of Mutatable interface methods to account for high evolution variability.

Subclasses

* ImageLeaf
* ImageComposite
* ImageDecorator

Patterns

* Composite: If it is an ImageComposite, it can have children, and a reference to its parent is maintained. This allows for complex tree structuring and dividing of ImageComponents.
* Template Method: defers implementation of steps of algorithms to subclasses
  + For instance, how to add children, or drawing.
* Iterator: provides a means for traversing the tree structure by utilizing the list of children as well as references to parents.

### ImageLeaf

This class represents the most basic building block for images, which are actual painted components that can be represented graphically on a canvas. For instance, brush strokes and shapes can be considered ImageLeaf’s. These simple objects typically can be composed or decorated to create complex structures.

Subclasses

* BrushStrokeLeaf
* ShapeLeaf
* GradientLeaf
* etc.

Patterns

* Composite: represents the Leaf in a Composite pattern implementation.

### ImageComposite

This class represents a container for other ImageComponents. Any number of Leaf’s, Composites, or Decorators can be contained within an ImageComposite, thus allowing for a tree structure.

Subclasses

No subclasses for ImageComposite are intended.

Patterns

* Composite: this class represents the Composite in the Composite pattern.

### ImageDecorator

This class represents all filtering and similar applied modifications to ImageComponents that can occur. These filtering effects can be exchanged, added, removed, and edited freely and their effects will cascade to the ImageComponent tree underneath.

Subclasses

* BrightnessContrastDecorator
* HueSaturationDecorator
* LevelsDecorator
* CurvesDecorator
* FilterDecorator
* etc.

Patterns

* Decorator: applies modifications to ImageComponents and represents a Decorator.

### Settings

Specific settings for ImageDecorators are abstracted and defined here. Sort of “waving the hand”, since implementation of the filters is out of the scope of this project.

Subclasses

* BrightnessContrastSettings
* HueSaturationSettings
* etc.

Patterns

* Bridge: decouples the representation of how a filter operates from the Decorator that carries it. Allows filters to be applied independently, not solely as a Decorator.

### ImageComponentFactory

Provides a factory for instantiating all ImageComponent objects. Specifically, a Simple Factory implementation is used, with String keys that determine the class to be instantiated. This allows for use of a random number generator to select a key in the evolution process.

Subclasses

* ImageLeafFactory
* ImageCompositeFactory
* ImageDecoratorFactory

Patterns

* Simple Factory (Factory Method): use of String typecodes determines class to be instantiated.

### ImageConverter

Allows composed ImageComponents to be converted into standard image file formats, such as .jpeg, .png, etc.

Subclasses

* JPEGConverter
* PNGConverter
* etc.

Patterns

* Strategy: The algorithm of image conversion is encapsulated in the ImageConverter object, and then independently implemented in subclasses.

### ImageGatherer

Provides different mechanisms for supplying the initial population for an evolution. Users can select files, draw an image using an assumed drawing application, or generate random populations for starting evolution.

Subclasses

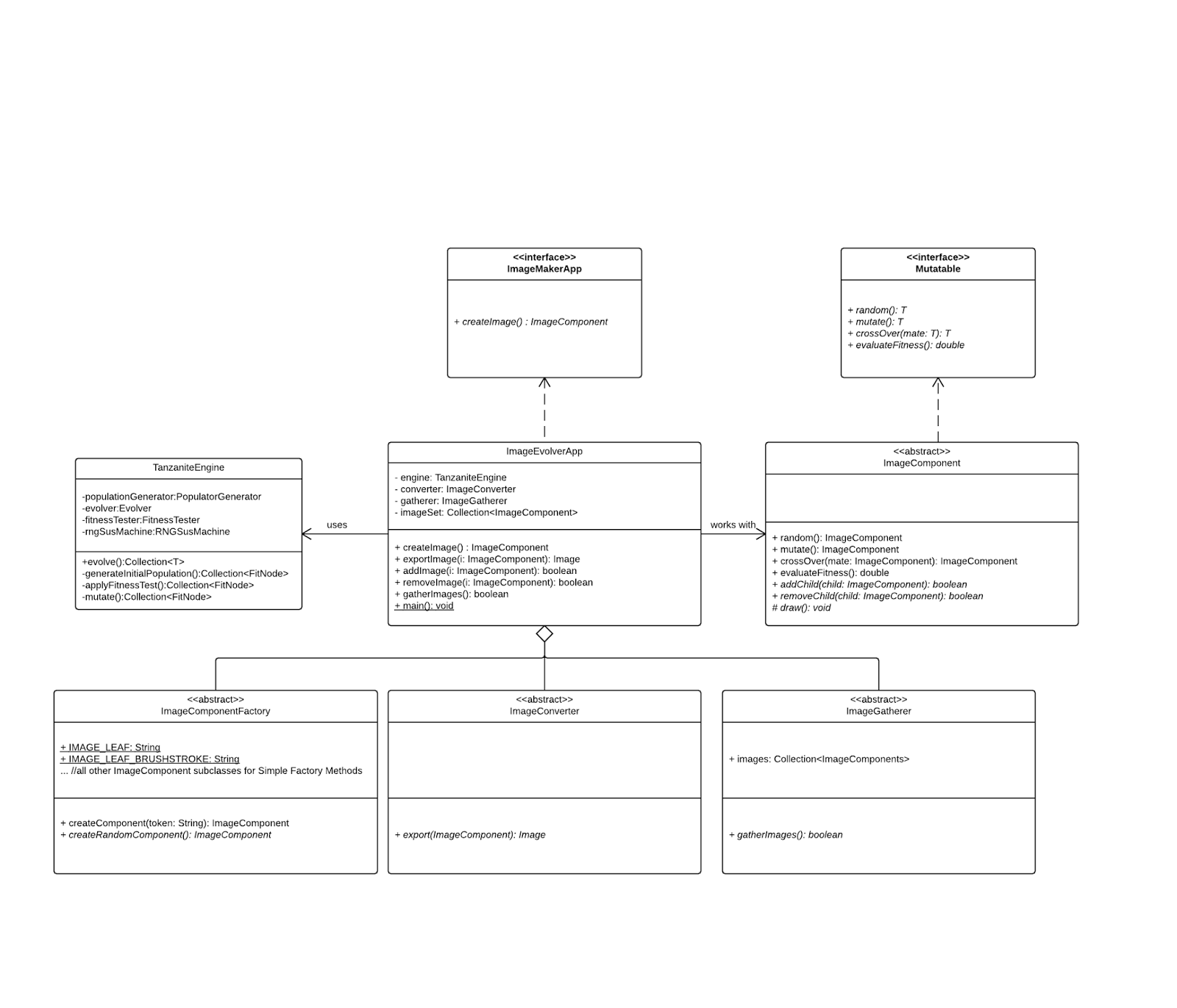
* FileSelectorGatherer
* ImageDrawGatherer
* RandomGatherer
* etc.

Patterns

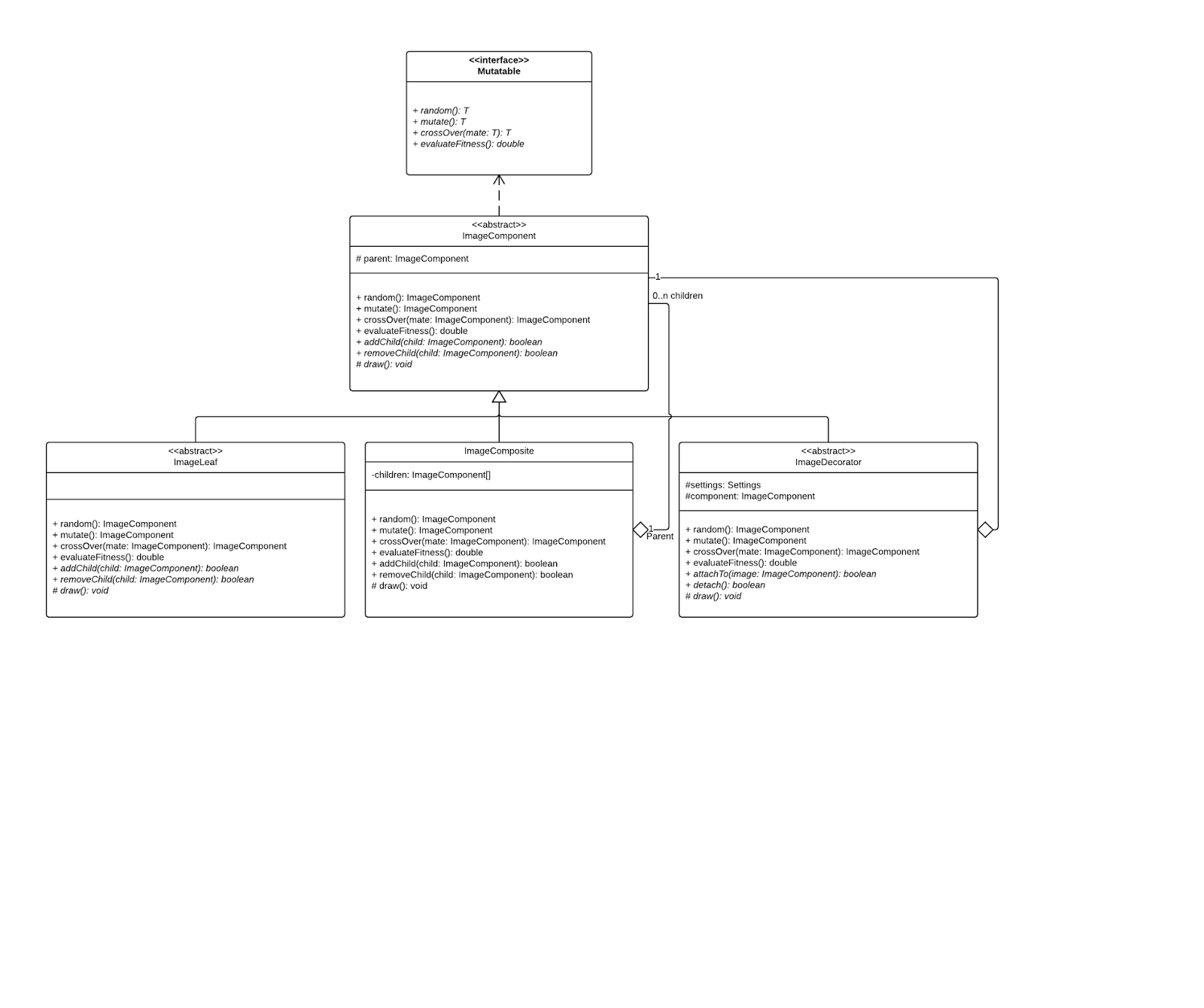
* Strategy: similar to ImageConverter, ImageGatherer allows the algorithm of gathering initial population Images to vary.

## UML Diagrams

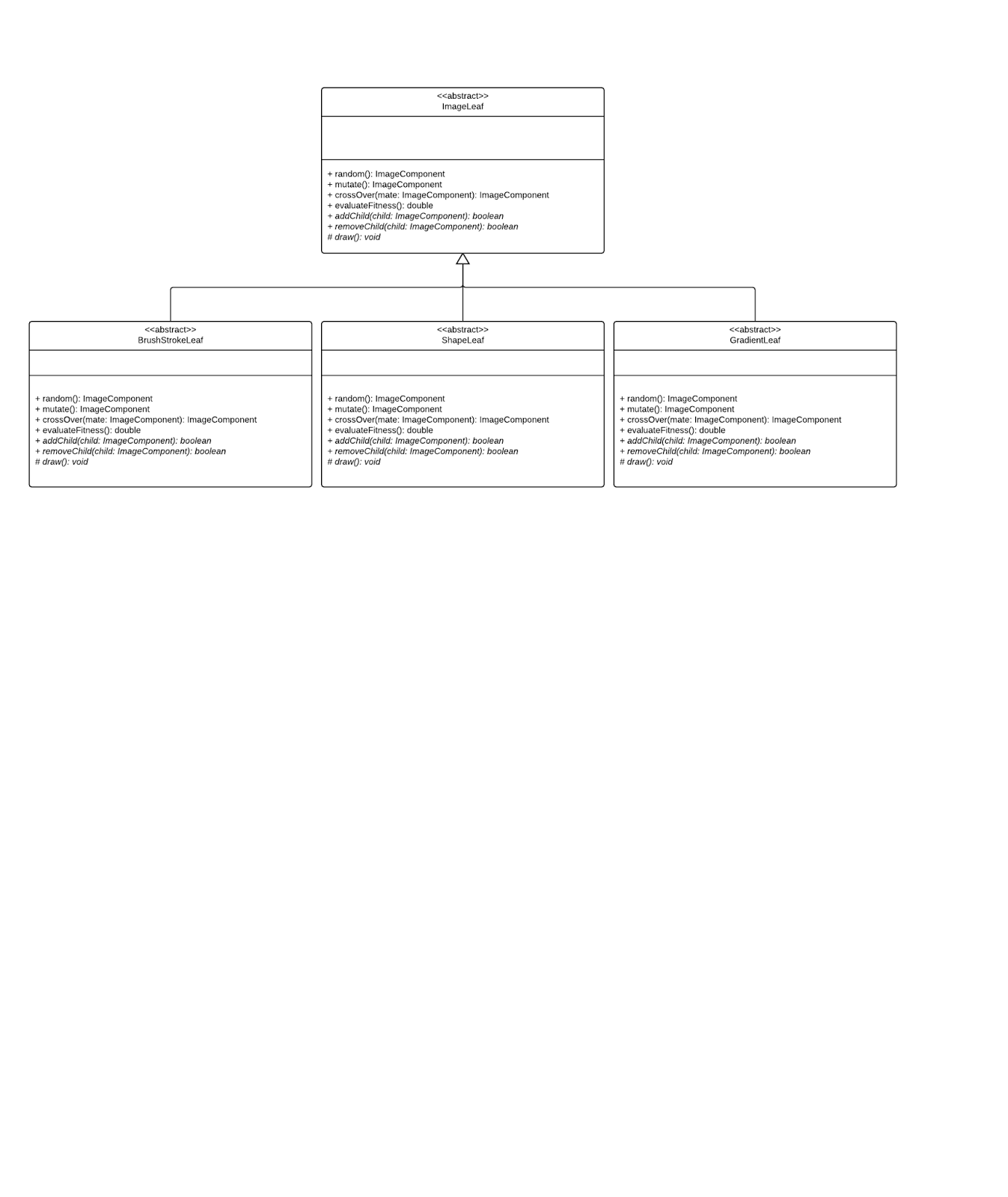
### ImageEvolverApp



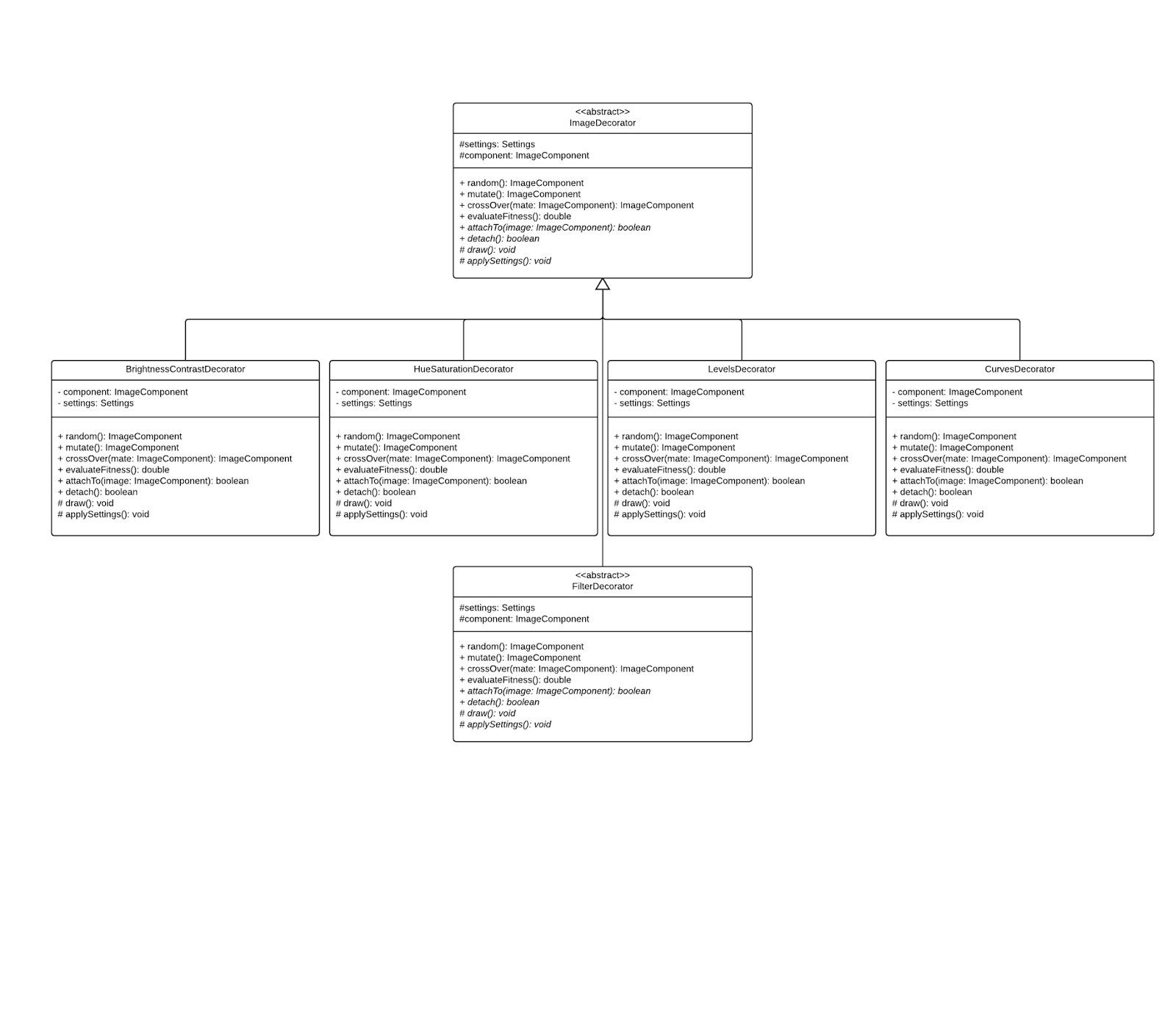
### ImageComponent



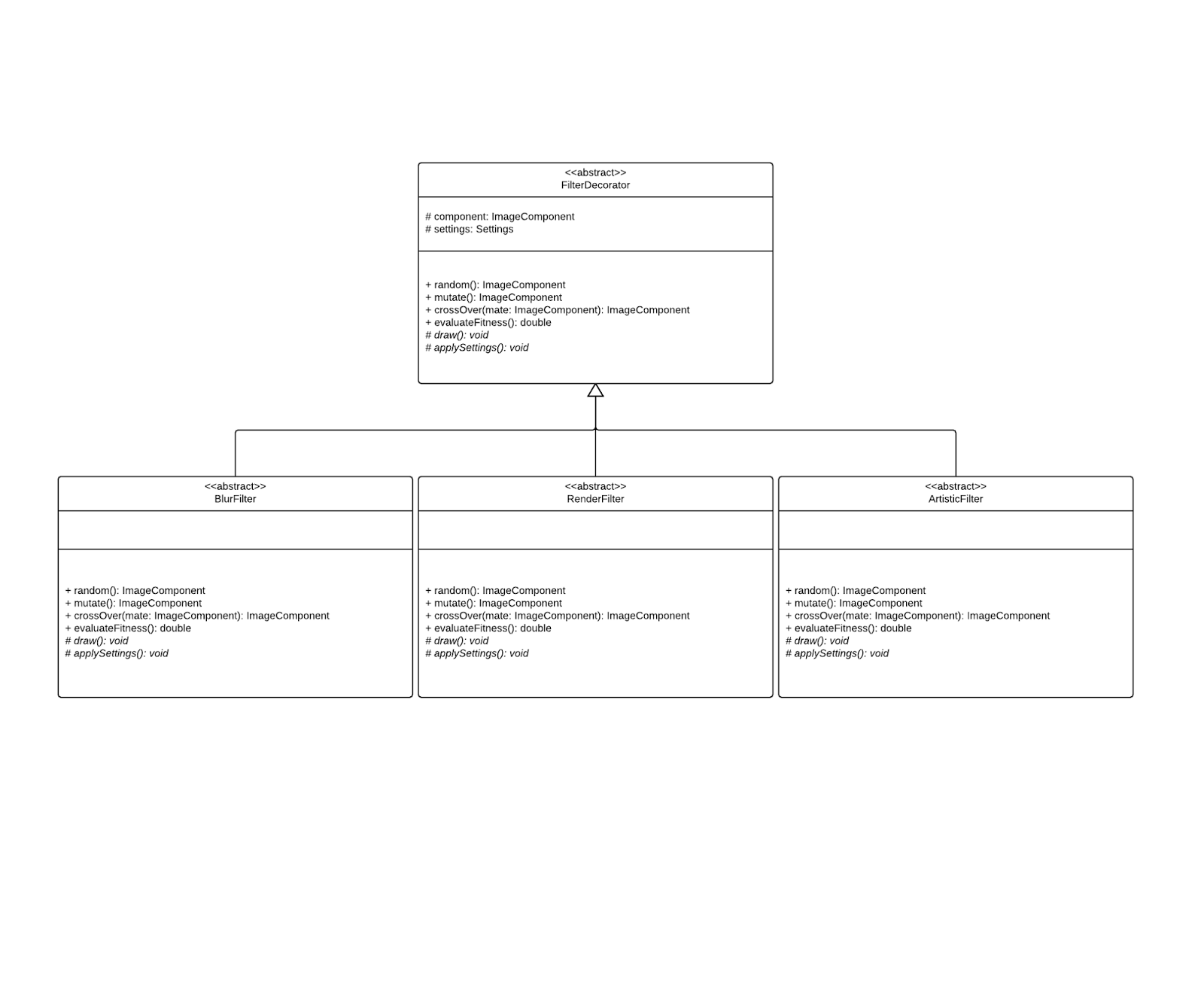
### ImageLeaf



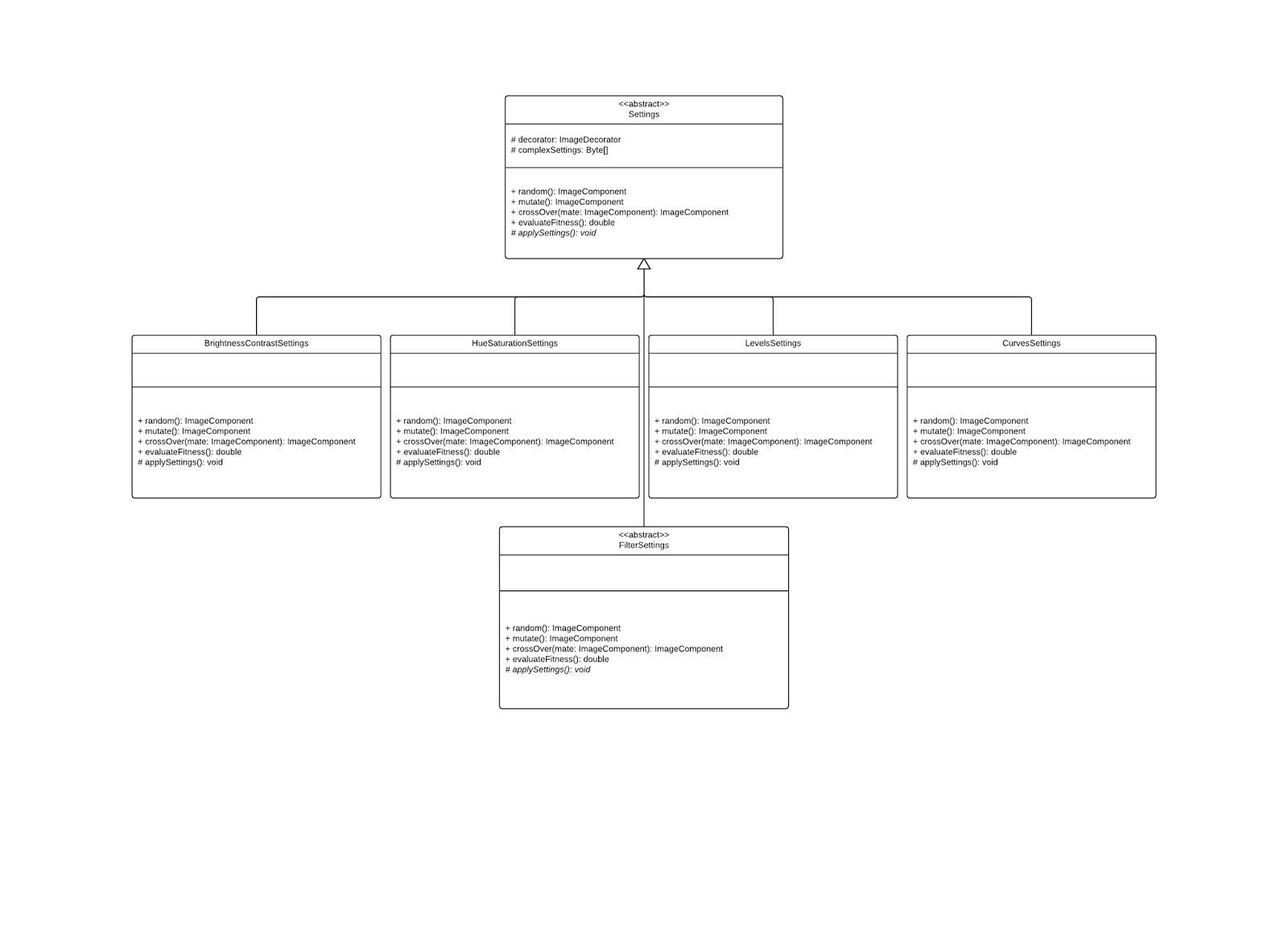
### ImageDecorator



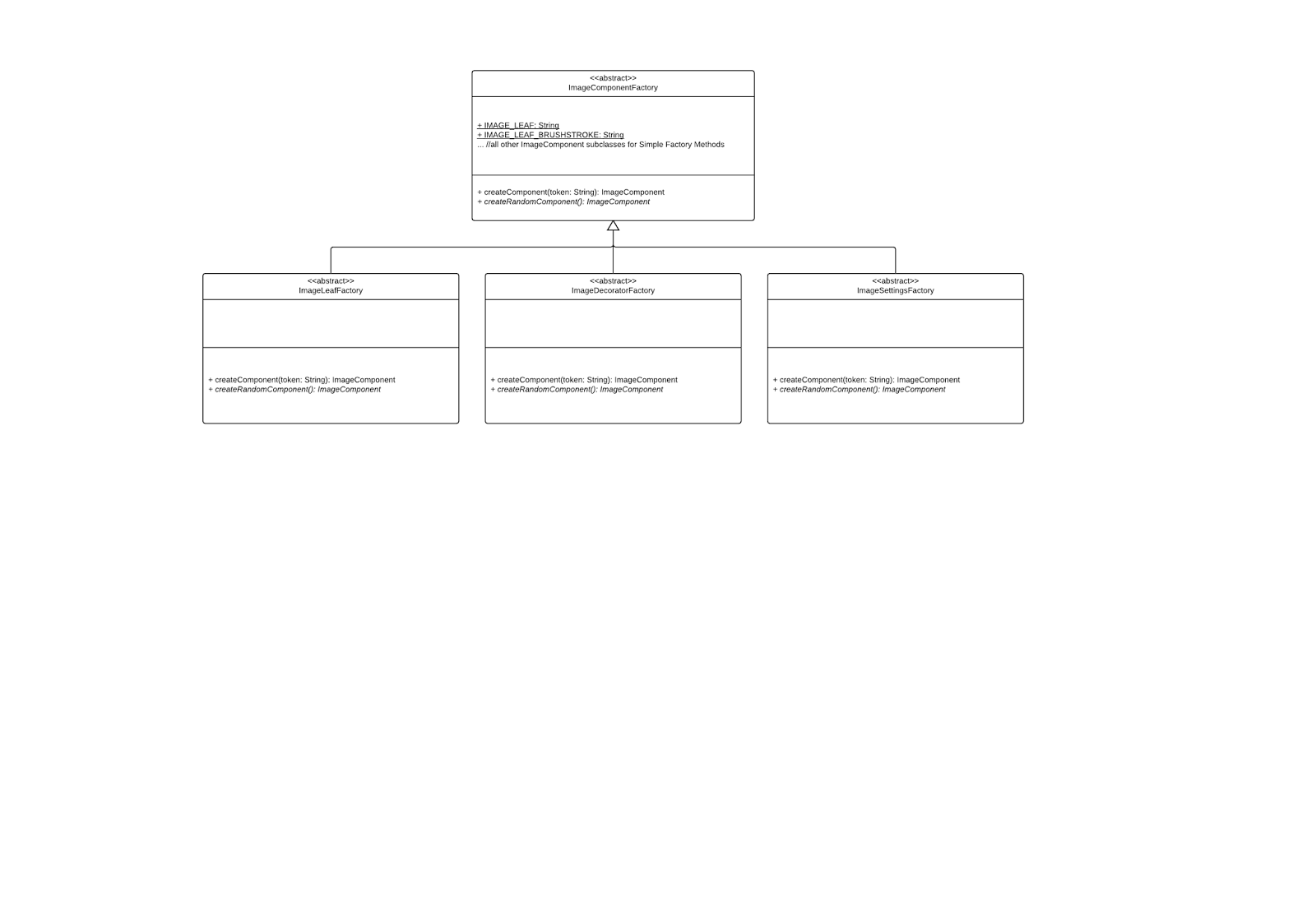
### FilterDecorator



### Settings

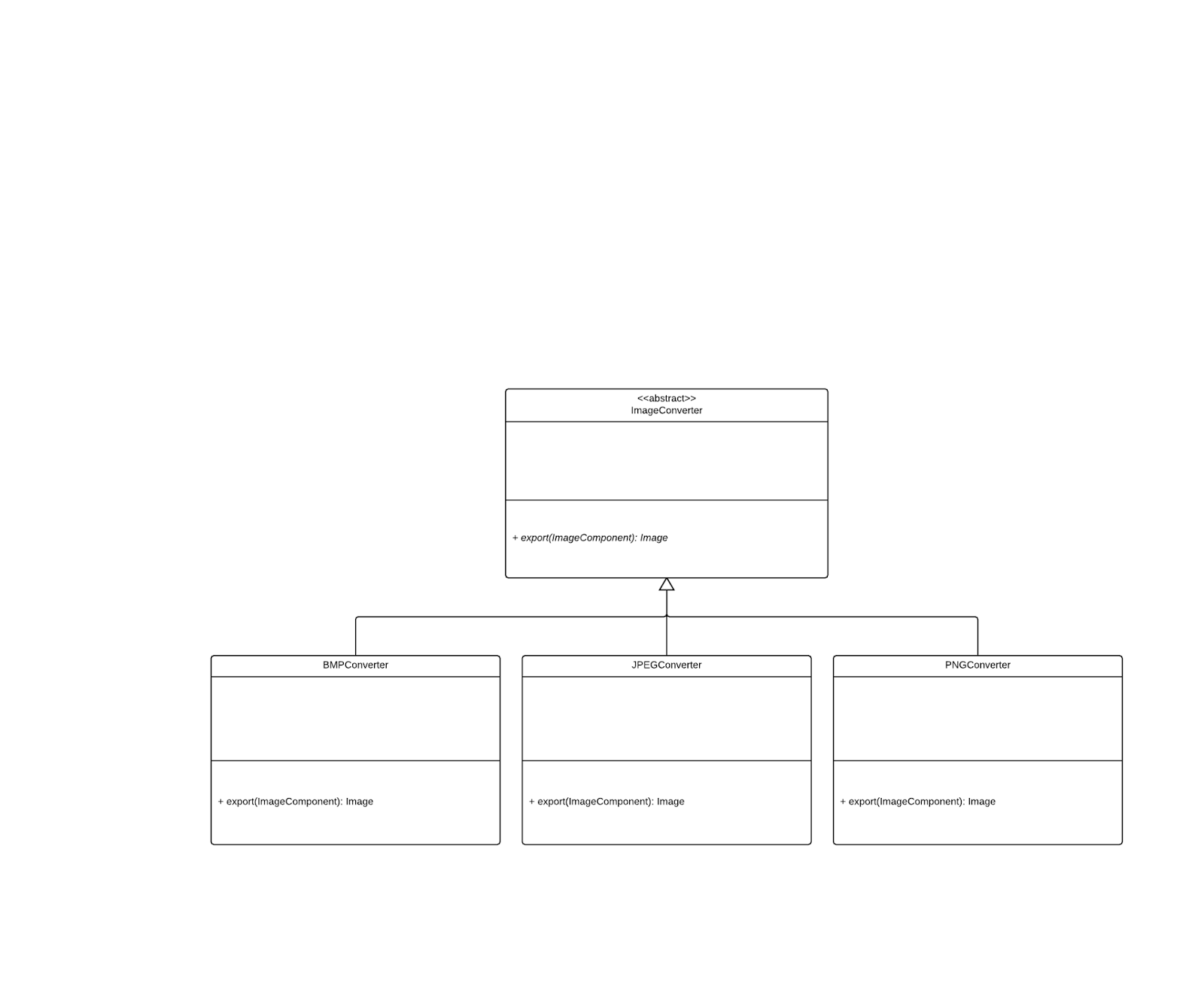


### ImageComponentFactory



### 

### ImageConverter

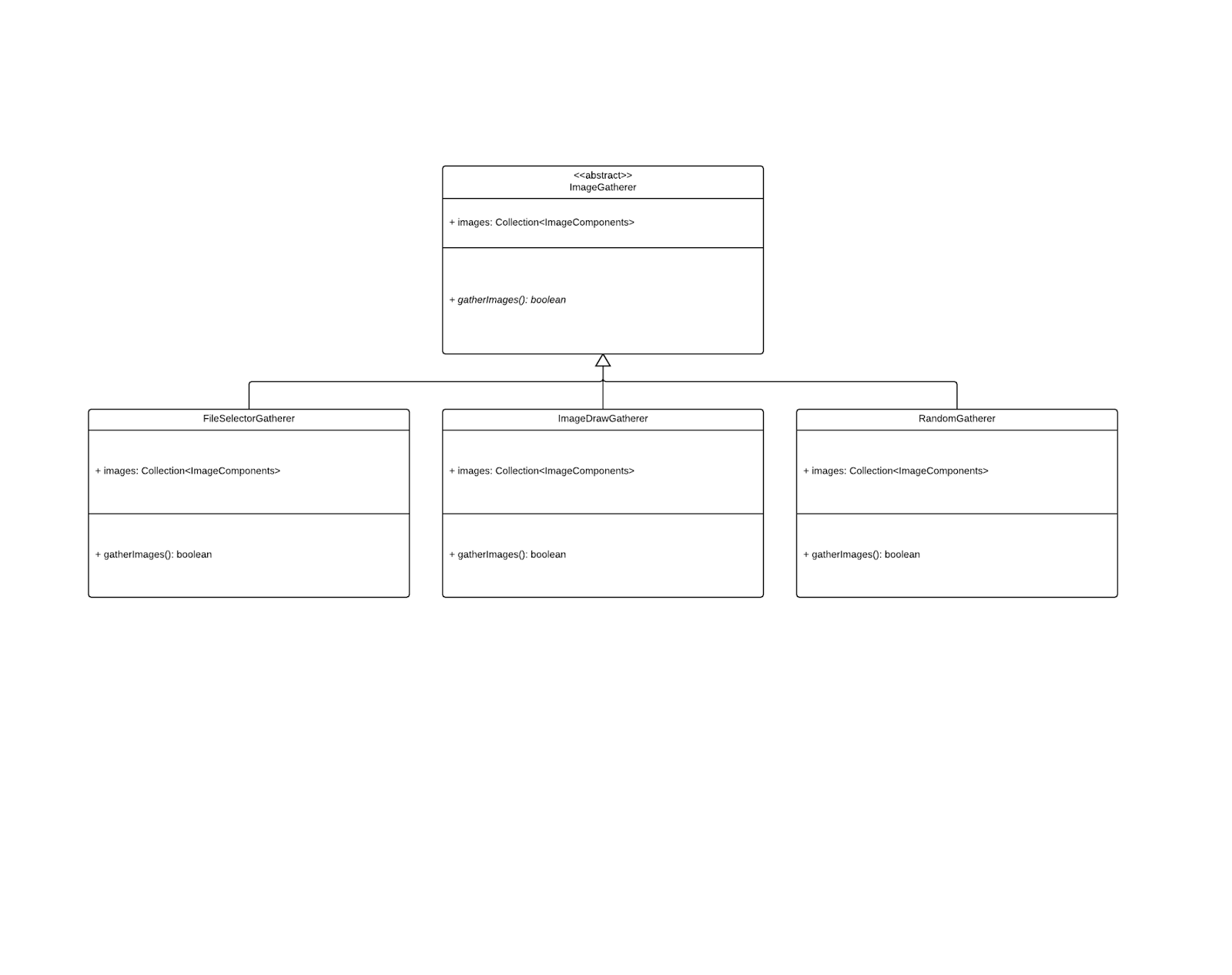


### 

### 

### 

### **ImageGatherer**

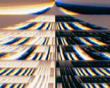
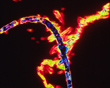
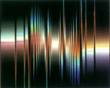
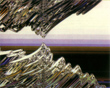


## Mock Output

Evolving image objects allows for a nearly infinite number of results. Karl Sims’s homepage demonstrates an exhibit where visitors choose which images they like from 16 original images, which influences the evolutionary chain used to synthesize new images. These images start out relatively simple, with general theme or pattern images, such as sine waves, color themes, textures, and similar concepts. Then, as users select which images they prefer, 16 evolutions combining those images are created, and this process continues until the user has created a desirable image.



ImageEvolverApp can have a similar output result to this if implemented correctly. You might expect several different kinds of outputs from possibly similar initial input, due to evolutionary tendencies. Possible resulting images are shown below.



# Application 2

**Overview**

With this application, the user is able to generate cipher texts used for symmetric key encryption. This is based off of the ICIGA (Improved Cryptography Inspired by Genetic Algorithms) block cipher system. The purpose is to create cipher texts while randomly generating keys to determine more secure protocols. Essentially, the user provides three things: key length, and plaintext, fitness level, and the output is a ciphertext with a give fitness level.

The all-encompassing class CipherComponent implements the Mutatable interface. By implementing Mutatable, there is the ability to override key functions to allow for specific customization for different encryption algorithms or other practices. We also gave default attributes and behaviors such as key length, block size, etc. to provide functionality to all subclasses, however if needed, the user is able to customize to their content.

The user can extend this application to generate Mutatable objects from a starting plaintext string or even within a larger application that supports symmetric key encryption. CipherEvolverApp can be a plaintext modifying feature present within this application, a module that allows users to create a plaintext string and then allow the TanzaniteEngine to carry out evolutions on the ciphertexts.

**Relevant Classes**

**CipherEvolverApp**

* The client of the TanzaniteEngine Framework
* A Façade to the underlying picture creation/representation framework

### CipherComponent (abstract) These are blocks

* Implements Mutatable
  + able to build itself randomly, mutate, swap subtrees (crossover), etc
* Utilizes Composite Pattern to construct itself

### CipherLeaf (abstract)

* Represents different abstract ways to construct a block
  + Hexadecimal, Octal, Decimal, Binary, etc.

### CipherComposite

* Composite pattern realization, holds up to N CipherComponents

### CipherDecorator (abstract)

* Capable of applying a certain encryption to modify the existing text string.
  + AES, DES3, SHA-256

### 

### Settings (abstract)

* Holds information about how to decorate a subtree
* This exact information is beyond scope of application design (very complex), but will approximately represented using a byte array
* Utilizes Bridge Pattern to separate application of settings from the Decorator
* Settings are applied to CipherComponents (Blocks)

### CipherComponentFactory (abstract) (Blocks)

* Used to create CipherComponents (Blocks)
* Allows for random key generation, useful for evolution

### CipherDecrypter (abstract)

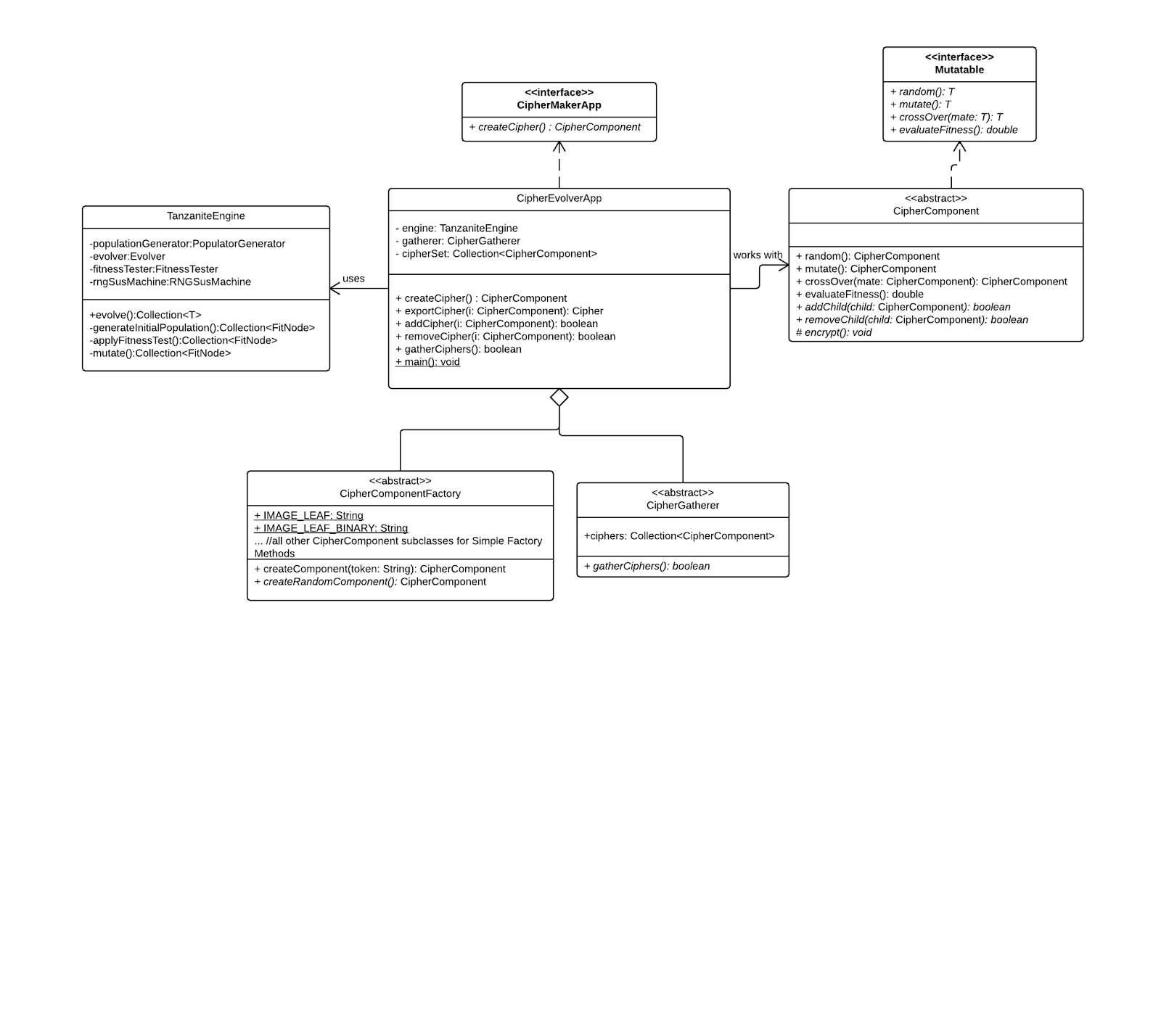
* Clients will not be able to read blocks (string of bits). Converting the bits and returning a common String type is necessary via exporting

### CipherGatherer (abstract)

* Initial populations for evolution can be selected using various kinds of CipherGatherers

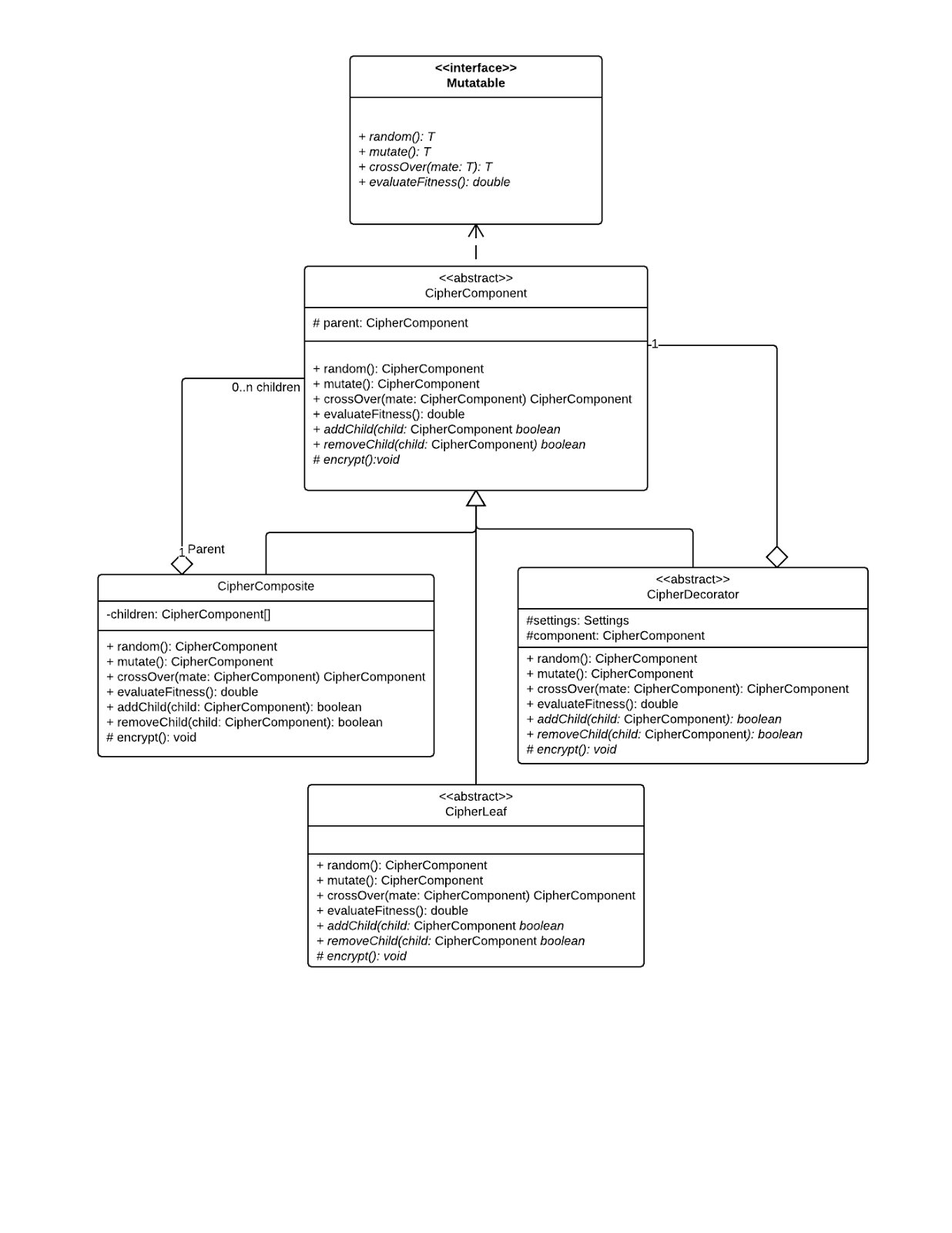
**UML Diagrams**

### CipherEvolverApp

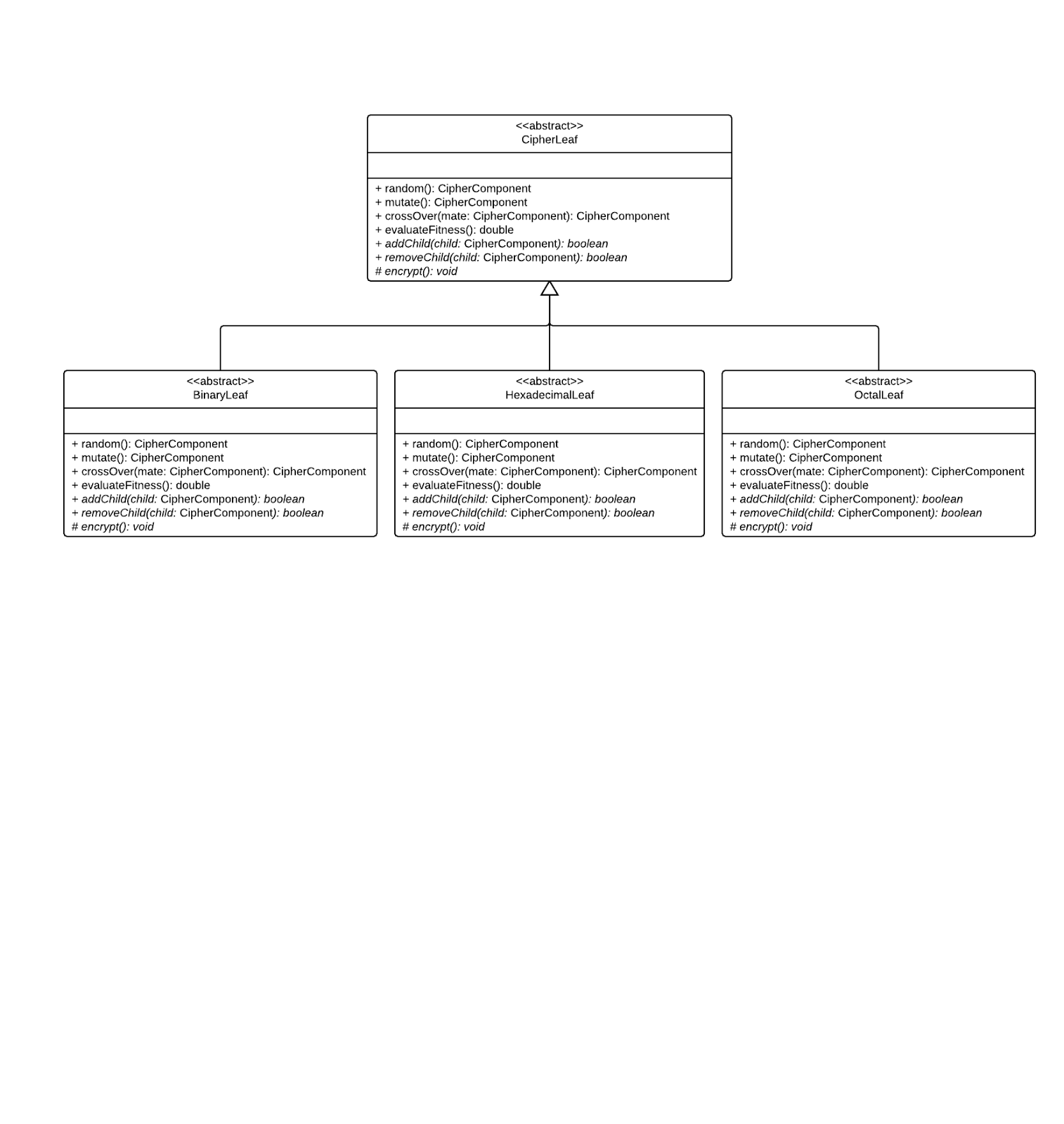


### 

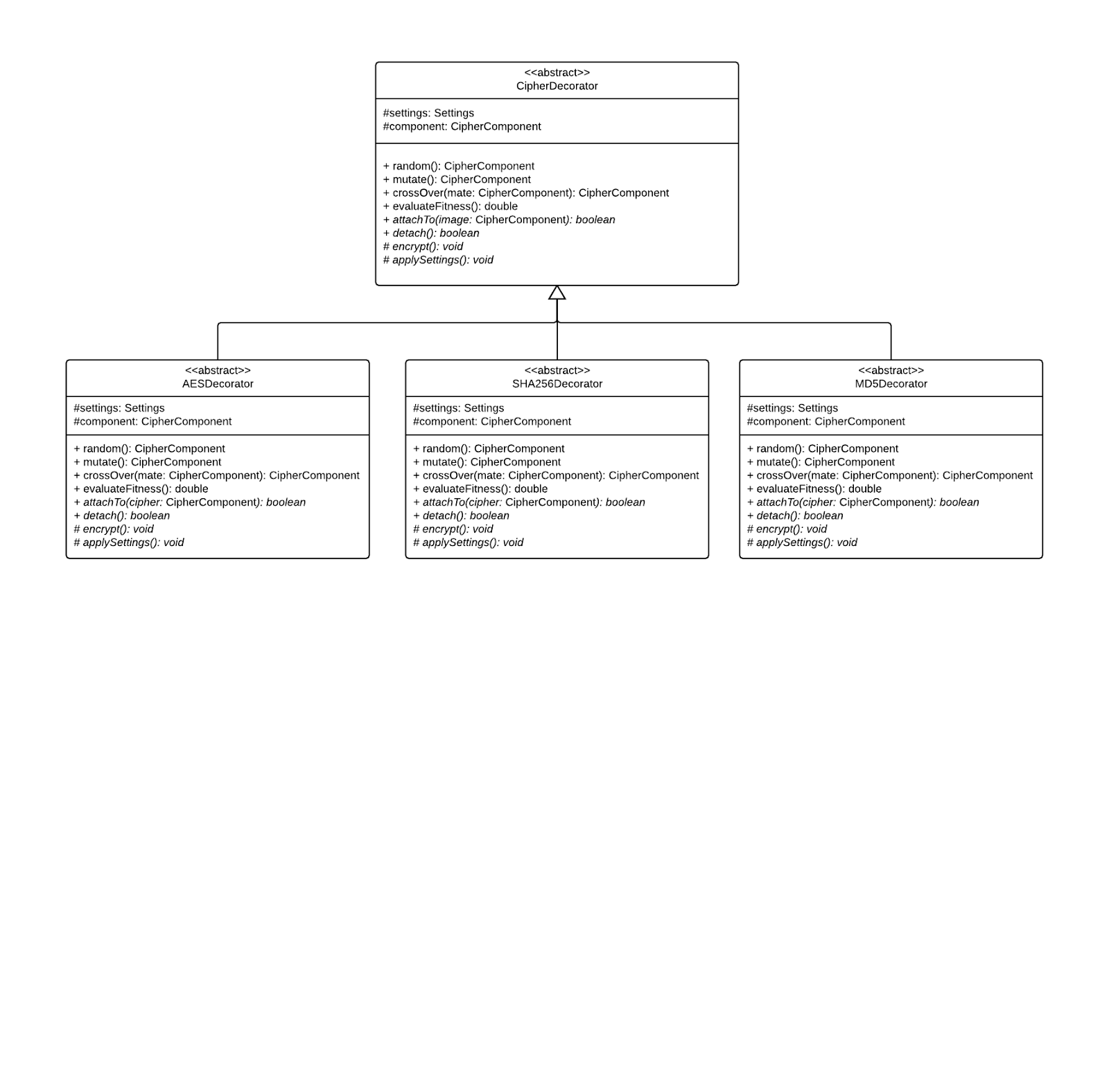
### CipherComponent



### CipherLeaf



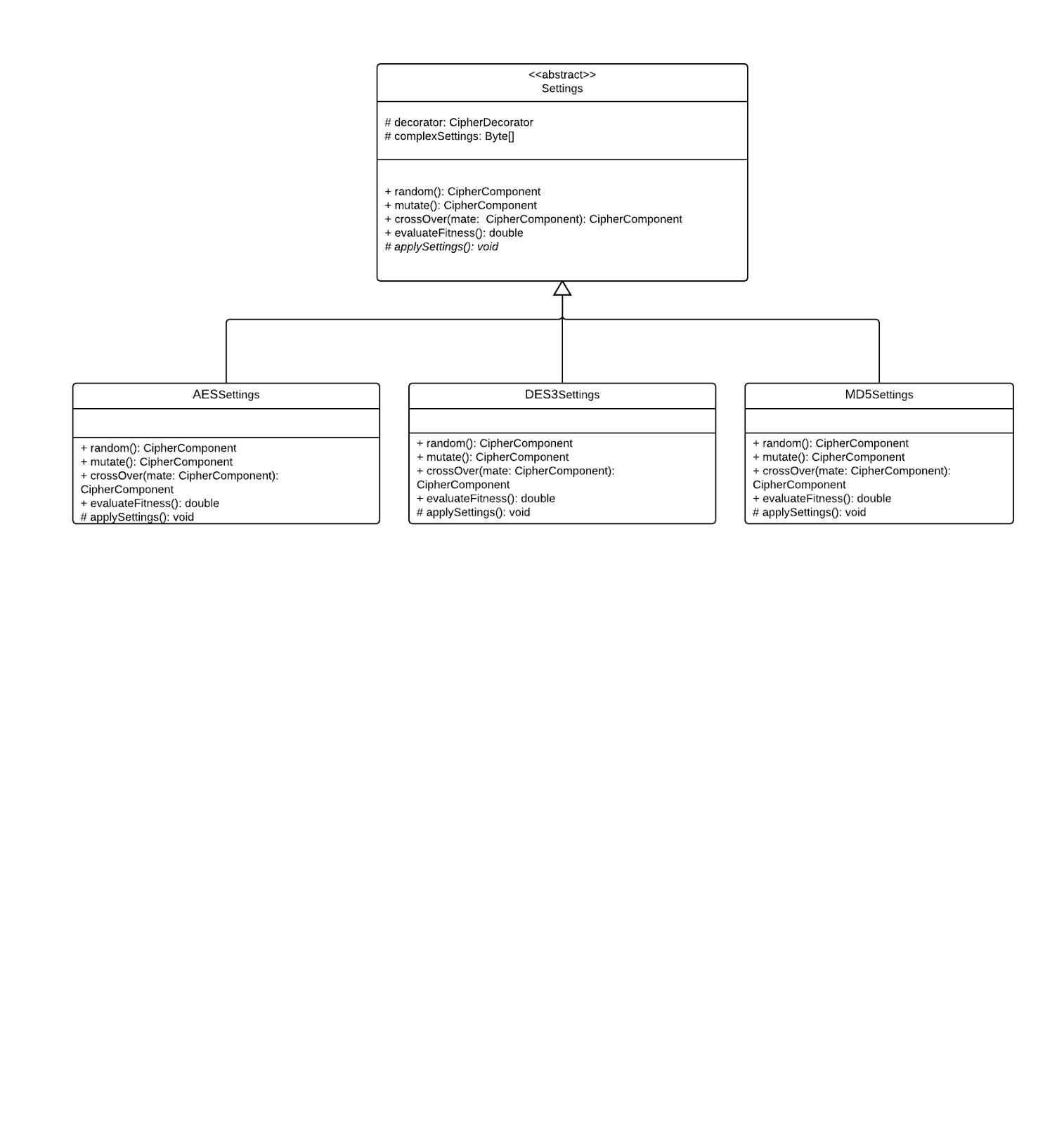
### CipherDecorator



### 

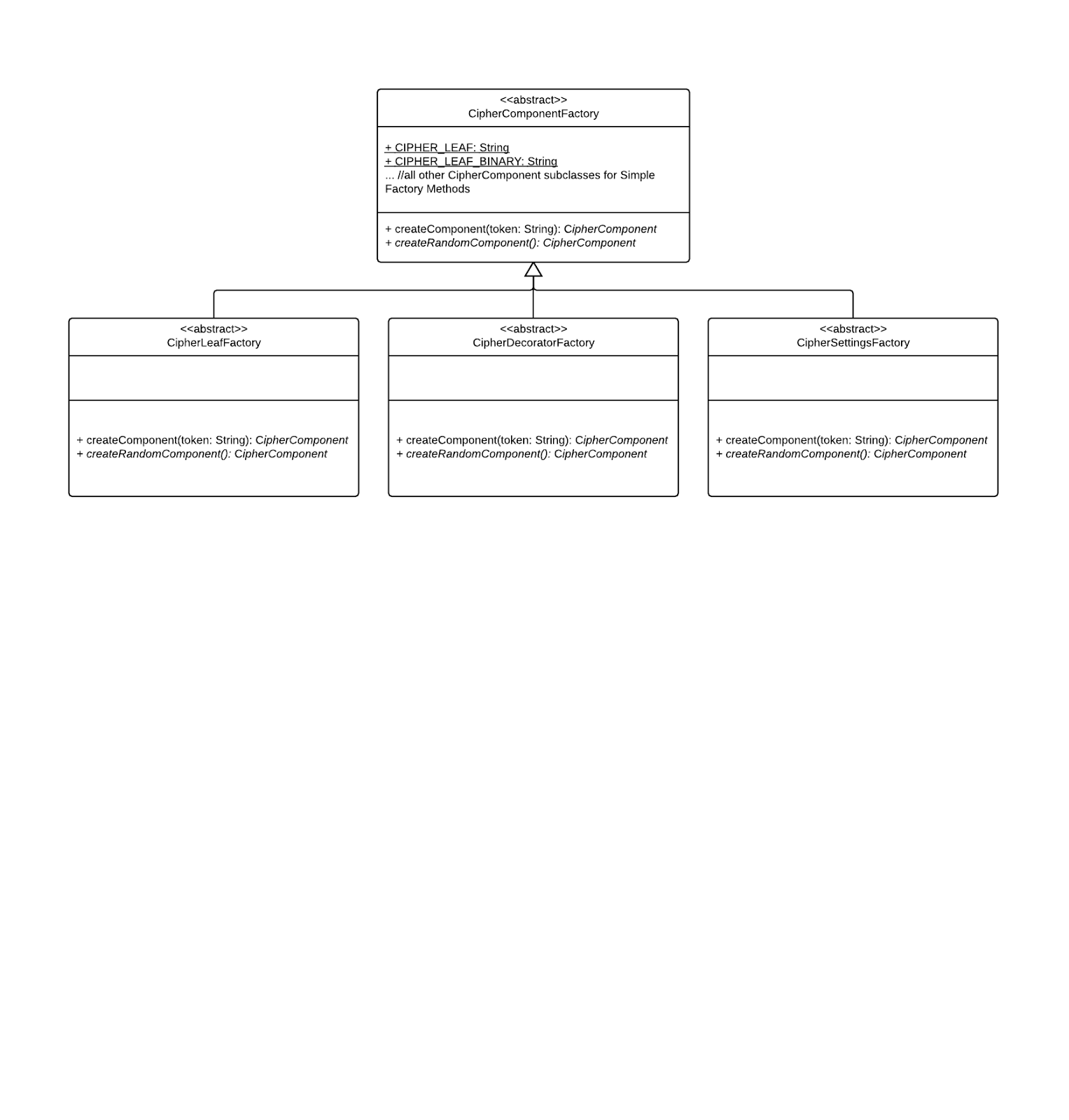
### 

### Settings



### 

### CipherComponentFactory



### 

### 

# Powerpoint Slides