1. INTRODUCTION

This paper introduces the EpochX system, a freely available, open source, object-oriented software framework for genetic programming, written in Java. The software provides packages for several variants of genetic programming: strongly-typed tree GP [6, 7], context-free grammar GP [9], and grammatical evolution [8]. A large number of commonly used operators are provided, and several benchmark problems have been implemented as examples. EpochX has been extensively used in experimental work [2, 3, 4].

The system has a number of benefits. It has been written to fully exploit the object-oriented nature of the Java language. In particular, new problem types and operators are created in a Java-natural style by the use of inheritance.

A number of distinctive features are provided, designed to facilitate the analysis of GP runs. One such feature is the provision of an event framework, based on the Java event handling model, which allows actions to be associated with particular events such as the start or end of generations or the performance of particular operators. Another feature is a rich statistics (also referred to as stats) provision, allowing each problem implementation to define its own set of statistics measures befitting the problem being tackled.

It is often the case that in research applications, information about the GP runs needs to be recorded for future analysis. One simple approach to generate output—and perhaps the most common one—is to insert print statements in to the code, which print information either to the terminal or to a file. While this seems simple, it is easy to see that it can become rather complex as the information to be outputted, which usually requires additional statements to be generated and formatted, occur in multiple places in the code. This leads to blocks of statements spread through the code for the purpose of generating and outputting information about the run. Simple tasks such as changing the output format then become more complex, since these multiple occurrences of output statements need to be updated. The use of logging facilities can centralise the output mechanism, but it still requires the addition of log statements and other statements to generate the information, throughout the code. Additionally, there is no mechanism to easily integrate textual and graphical output.

Apart from providing a facility to output information, there is a requirement to provide a mechanism to gather statistics data about a run. For example, ECJ [1, 10]—a popular evolutionary computation framework—provides statistics hooks, in which ECJ calls predefined methods on a Statistics object during the run of the evolutionary algorithm. Subclasses of the Statistics class can provide custom implementations of these hooks (methods), such as:

- preInitializationStatistics(EvolutionState state);
- preEvaluationStatistics(EvolutionState state);
- finalStatistics(EvolutionState state, int result);

To generate additional information about the run. At the same time that the use of hooks provide a customizable way of gathering and outputting statistics information, they define an arbitrary (rigid) structure related to when the information is generated. In this paper, we discuss how statistics information about the run of an evolutionary algorithm is generated and outputted in the EpochX framework.

The remainder of the paper is structured as follows. Section 2 gives details of the basic ideas and design principles of the EpochX framework. It also outlines the original statistics system (Stats system) and highlights its limitations.
Figure 1: An illustration the organisational structure of EpochX. Each representation package depends on the common module which in turn depends on the core evolutionary framework.

Section 3 discusses the new event-driven Stats system with consideration for how it overcomes these problems. Section 4 discusses how the information gathered can be displayed to the user. Section 5 lists a complete example code to solve one commonly used benchmark problem with EpochX and generate output. This is followed by some conclusions presented in Section 6.

2. EPOCHX

EpochX has been designed with a modular structure, comprised of a core evolutionary framework and separate packages to support specific representations. Figure 1 shows the layered structure of the modules, where the representation packages are built on top of the common module, which in turn depends on the core framework. The core framework of EpochX includes the components of the evolutionary algorithm itself and provides an infrastructure for modifying most aspects of this algorithm. The common module extends this framework to add a library of commonly used tools and utilities, that are shared by multiple representations. The representation packages define the representation of an individual and supply the operators and other components necessary to support that representation.

An evolutionary run is performed by creating an Evolver object and calling its run() method:

Evolver evolver = new Evolver();
evolver.run();

The Evolver class is composed of a pipeline of components, which are executed sequentially when the evolver’s run() method is called. Each component is passed a population of individuals as input and after performing some processing, returns a population of individuals which the pipeline will pass on to the next component. By default, the evolver’s pipeline is composed of three components:

1. Initialiser
2. FitnessEvaluator
3. EvolutionaryStrategy

An empty population is passed to the first component, which would typically generate an initial population. The FitnessEvaluator component will then use a fitness function provided by the user to assign a Fitness to each individual in the population. Then, this initial population will be passed to the EvolutionaryStrategy, which is responsible for evolving the population. Different implementations of each of these components are possible, and the pipeline may be modified to use a different set of components entirely.

For a normal generational evolutionary algorithm, a GenerationalStrategy is defined. This uses a second pipeline which is executed multiple times until any of a set of user defined termination criteria are met. Each iteration is considered to be a ‘generation’. This pipeline would typically be made up of components that apply genetic operators to produce a new population from a previous population, and a FitnessEvaluator to assign a Fitness to each individual in a generation’s population.

Prior to calling the run() method of the evolver, the framework should be set up with control parameters that are suitable for the problem to be solved. A configuration repository called Config, is provided, which is used to specify the initial settings of control parameters and to allow them to be modified as a run progresses. Values are stored in the configuration repository as key/value pairs, with Java’s generics feature used to enforce the type safety of values and avoid errors occurring. For instance, a selection operator may define this configuration key:

```
ConfigKey<Integer> TOURNAMENT_SIZE =
    new ConfigKey<Integer>();
```

The repository will enforce the generic type constraint of the key, and ensure that only integer values are set with it. All components and operators are written to obtain their configuration from this repository. Where possible, the framework and operators are written to insert sensible defaults for their configuration options into the repository. These defaults may be overwritten as necessary. The following code would overwrite the default tournament size, using the key defined above:

```
// gets the singleton Config instance
Config config = Config.getInstance();

// initialises it with default values
config.defaults();

// overwrites the default tournament size
config.set(TournamentSelector.TOURNAMENT_SIZE, 3);
```
The core framework is kept as lightweight as possible. Many of the operators it relies on are provided as interfaces which define the functionality required. For example, the Operator interface, which models a genetic operator, defines methods for obtaining the number of individuals it operates on and for applying the operator to an array of individuals. The framework is able to use these methods without any knowledge of the representation in use. Implementations of these operators are supplied by the representation packages and common module. The most commonly used operators are supplied for each representation, but new operators are easily added by implementing the Operator interface and setting the new operator in the configuration repository.

As a run progresses, data about the progress of the run is accessible through two facilities that are provided by the framework: (1) an event system allows components and other user written code to be notified about each stage of the algorithm and to receive related data; (2) more in-depth information and statistics are provided through the Stats system, which presents a convenient and efficient way of generating additional data to be outputted to the screen and/or file, either as textual or graphical information. Event management in EpochX is described in detail in Section 2.1, but the main focus of this paper is the Stats system. Section 2.2 describes the original (previous) Stats system, as well as highlighting some of the serious limitations that it suffers from. However, the Stats system was completely redesigned for version 2.0 of EpochX to overcome these flaws. There was also a desire to make it easier for a user to extend the available ‘stats’, to provide their own data through the same infrastructure. This new Stats system will be described in detail in Section 3.

2.1 Events

The event management in EpochX is implemented using the observer design pattern [5]. The observer pattern defines a one-to-many dependency between an observable object and a set of observer objects, allowing observer objects to be notified of changes of the observable object. EpochX’s event management uses events (observable objects) to notify listeners (observer objects) that something of interest has occurred during the run of the algorithm—e.g., the event of starting a generation or the event when a genetic operator has been performed. This allows the listeners to perform additional actions in response to events.

A singleton EventManager instance is responsible for registering listeners and also it provides the mechanism to fire events. The EventManager allows:

- An event to have multiple listeners. When an event has multiple listeners, the listeners are notified sequentially when an event is fired. Events that have no listeners, are never fired.
- Notification of events respecting the class hierarchy. When an event is fired, the EventManager notifies the listeners of the specified event and also the listeners of any of the superclasses of the event—e.g., firing a CrossoverEvent will notify listeners of the CrossoverEvent and also the listeners of the OperatorEvent.

Custom events can be easily created by implementing the empty Event interface and events can encapsulate any data to be passed to the listeners. Custom listeners are created by implementing the Listener interface:

```java
public interface Listener<T extends Event> {
    public void onEvent(T event);
}
```

The onEvent method is called by the EventManager to notify that an event has occurred. The listener receives the (typed) event object and can perform an action in response to the event. Events are fired using the `EventManager`’s `fire` method:

```java
EventManager.getInstance().fire(new EndGeneration(generation, population));
```

where the `fire` method’s parameter is the event object. In the above example, the EndGeneration event encapsulates the generation number and the population of the generation that has just finished.

The core framework provides events for the general lifecycle of an evolutionary algorithm—such as the start/end of a run, initialisation of the population, generation and genetic operators—and custom events can be easily incorporated. For example, the core framework provides a base class for the creation of genetic operators, which automatically fires start and end events:

```java
public abstract class AbstractOperator implements Operator {
    public final Individual[] apply(Individual ... individuals) {
        Individual[] parents = clone(individuals);

        // fires the start event
        StartOperator start = getStartEvent(individuals);
        EventManager.getInstance().fire(start);

        // performs the operator
        parents = perform(parents);

        // fires the end event
        EndOperator end = getEndEvent(individuals);
        end.setChildren(clone(parents));
        EventManager.getInstance().fire(end);

        return parents;
    }

    // ...
}
```

Subclasses of the AbstractOperator must implement the perform method, which is responsible for applying the genetic operator, and may optionally override the `getStartEvent` and `getEndEvent` methods to provide alternative implementations of the start and end operator events. The default implementation of the StartEvent provides the information of the parent individuals (the individuals undergoing the genetic operator), while the default implementation of the EndEvent provides the information of the parent individuals and also the information of the offspring (the individuals produced by the genetic operator).
2.2 Stats

The original (previous) Stats system, used in EpochX releases prior to the one described in this paper, uses a central repository to store data about the progress of an evolutionary run. Certain items of raw data, such as a generation’s population of individuals or the duration of an evaluation, are inserted into this repository by the framework. Each item of data is referenced using a key, which is an object that implements a Stat interface. This key can be used to retrieve any data stored to it in the repository. But it also has an alternative use to generate the stat if it does not already exist. If the repository does not contain any data for a required stat, then the repository makes a method call on the stat object itself, requesting for it to be generated. The stat is then responsible for producing and returning a value for the key it represents.

To produce a value, a stat object needs to obtain data to process. It can do this by requesting data from the repository, using other stats as keys. These stats are its dependencies. This leads to a situation where stats are chained, so that each one may be dependent on several others, which are themselves dependent on other stats, and so on. Requesting a stat value from the repository can trigger a whole chain of stats to be generated. For example, the GEN_FITNESS_STDEV stat is responsible for producing the standard deviation of a population’s fitness scores. It is dependent on two other stats: GEN_FITNESSES, which provides a list of fitness values for the population and GEN_FITNESS_AVE, which gives the mean fitness of the population. The GEN_FITNESS_STDEV stat could just calculate the mean itself, but by using the stat system, it ensures that the mean fitness only needs to be calculated once. If the user was to output both the mean fitness and the standard deviation for each generation, then they would reuse the same partial calculation. This efficiency becomes even more valuable where more complex analysis is being performed.

When data has been generated, it is stored in the repository so that it does not need to be generated again. However, there are times when the value for a stat changes, and the stored value must be updated. For example, when a generation ends, how does the repository know that the average generation fitness data is out of date? To overcome this issue, each stat is associated with an event which indicates a point of expiry. The repository then listens for events that are fired by the EpochX framework, and clears all the data associated with an event when it occurs. This prevents the repository returning old data, and new values get generated instead.

As well as providing a mechanism for gathering data, the Stats system enables a convenient way of printing the data to screen or to a file. The repository exposes print methods for outputting to an OutputStream, with System.out used by default to print to the console. These methods take a variable number of Stat objects as arguments, which define the fields to be printed. Values for these stats are retrieved or generated as required, and then printed to the output stream in a delimiter separated row, in the same order as the stat keys were provided:

```java
Stats.getInstance().print(
    GEN_NUMBER,
    GEN_FITNESS_MIN,
    GEN_FITNESS_AVE);
```

This provides a simple way of outputting data in a convenient format. A typical use case is for data to be printed at regular intervals, such as after every generation. This is supported by combining use of the Stats system with events, to call a print method on each occurrence of an event. An idiomatic way of doing this is with an anonymous listener class:

```java
EventManager.getInstance().add(
    EndGeneration.class,
    new Listener<EndGeneration>() {
        public void onEvent(EndGeneration event) {
            Stats.getInstance().print(
                GEN_NUMBER,
                GEN_FITNESS_MIN,
                GEN_FITNESS_AVE);
        }
    });
```

In this example, a listener is added to the EndGeneration event. When the event is fired, the onEvent method will be called, which will print the latest row of stats data.

The major strengths of the Stats system are that it processes data efficiently and it provides a simple and consistent interface for obtaining data about all aspects of an evolutionary run. But, this original implementation suffers from some significant limitations:

- Each stat can only depend on either the raw data which is supplied by the framework, or on other stats which are themselves directly or indirectly dependent on the raw data.
- It is not possible to create stats which are based on data from multiple occurrences of an event. For example, a stat cannot provide the mean time to perform a crossover in a generation, because at the end of each generation only data from the last crossover will be available.

The next section presents the redesigned Stat system that addresses the aforementioned limitations.

3. NEW EVENT-DRIVEN STATS SYSTEM

EpochX’s Stats system has been redesigned to take advantage of the event model, described in Subsection 2.1. Stat objects are implemented to work as listeners and each stat is associated with an event, which triggers the stat object to calculate/update its information. This not only provides a more flexible Stats system but also addresses the limitations discussed in Section 2—i.e., stats do not depend on raw data provided by the framework and can collect data from any event during the run. Additionally, there is no need for a central repository to store raw data about the progress of the run, since data is gathered and stored by stats objects.

Stats are implemented by subclassing the AbstractStat class. The AbstractStat class is not only the base class for stats, but also provides a mechanism to register and retrieve stat objects. Stats classes are not instantiated directly, instead they are registered in a repository. This allows the Stats system to make sure that there is only one instance of the stat, since it does not make sense to have more than one
object calculating/storing the same information. The central repository of stats facilitates the management of their dependencies—when a stat is instantiated by the Stat system, all its dependencies are automatically instantiated if they are not in the repository—and the prevention of circular dependencies. For example, if stat A depends on stat B, and stat B depends on stat A, any attempt to register either stat A or B will generate an exception.

The new Stats system also makes it easier for a user to provide custom stats, based on the information of the events provided by the core framework, custom events or other stats objects. Although stats classes do not implement the Listener interface, each stat object has a listener registered by the superclass AbstractStat to receive the notification of the event that triggers the stat to be calculated/updated.

Let us consider a simple example of a stat that stores the current generation number. The basic implementation is:

```java
public class GenerationNumber
    extends AbstractStat<StartGeneration> {

    private int current;

    public void refresh(StartGeneration event) {
        current = event.getGeneration();
    }

    public int getCurrent() {
        return current;
    }
}
```

The above example highlights the connection between events and stat objects. The extends AbstractStat<StartGeneration> clause indicates that the GenerationNumber stat is triggered by the StartGeneration event. The information collected by the GenerationNumber stat is also provided by the StartGeneration event, corresponding to the generation number associated with the event. The core framework fires the StartGeneration event at the beginning of each generation. Every time a new StartGeneration event is fired, the GenerationNumber stat is updated to store the current generation number. The information of a stat object can be queried by retrieving the stat object from the central repository:

```java
GenerationNumber stat = AbstractStat.get(GenerationNumber.class);
int generation = stat.getCurrent();
```

At any point of the execution of the algorithm, a stat object can be retrieved from the repository and have its information queried.

The previous example shows how simple it is to implement a new stat and retrieve its information, although it does not illustrate the full flexibility of the Stat system. The Stat system provides a dependency mechanism and more elaborate stats can be created without increasing their complexity by chaining stat objects. The chaining of stat objects is achieved by defining dependencies amongst them, i.e., a stat object can depend on one or more stat objects. The basic idea is to have different stats representing specific units of information, which can be reused by other stats to derive different information.

Let us consider an example of the commonly used stat to compute the population’s average fitness of a generation. The basic implementation is:

```java
public class GenerationAverageFitness
    extends AbstractStat<EndGeneration> {

    private double average;

    public GenerationAverageFitness() {
        super(GenerationFitnesses.class);
    }

    public void refresh(EndGeneration event) {
        Fitness[] fitnesses = AbstractStat.get(GenerationFitnesses.class).getFitnesses();
        average = 0.0;
        for (Fitness fitness: fitnesses) {
            average += fitness.getValue();
        }
        average /= fitnesses.length;
    }

    public double getAverage() {
        return average;
    }
}
```

The GenerationAverageFitness stat is triggered by the EndGeneration event, as indicated by the extends AbstractStat<EndGeneration> clause. The super constructor call super(GenerationFitnesses.class) informs the AbstractStat superclass that this stat depends on the GenerationFitnesses stat. By indicating its dependencies, the Stat system guarantees that the GenerationFitnesses stat is present in the repository and it can be retrieved using the call AbstractStat.get(GenerationFitnesses.class). The GenerationFitnesses is a simple stat that holds the population’s fitnesses of the current generation. The GenerationAverageFitness queries the information about the population’s fitnesses to calculate and store the average fitness value. Note that in contrast to our previous GenerationNumber stat example, the information provided by the event which triggers the GenerationAverageFitness stat (the EndGeneration event) is not actually used—the event is only used to determine when the GenerationAverageFitness stat should be calculated.

Similarly to the above GenerationAverageFitness example, the population’s fitnesses information provided by the GenerationFitnesses stat can be used to determine the standard deviation of the population’s fitness values of a generation:

```java
public class GenerationStandardDeviationFitness
    extends AbstractStat<EndGeneration> {
```
Figure 2: An illustration of EpochX’s Stat system. The dependency amongst stat objects creates a chain (represented by a dashed line): the value of the GenerationFitnesses stat is calculated first, and its information is then used by the GenerationAverageFitness; finally, the information of both GenerationFitnesses and GenerationAverageFitness stats is used by the GenerationStandardDeviationFitness stat.

As can be seen in its constructor, the GenerationStandardDeviationFitness stat depends on the GenerationFitnesses stat—from where the population’s fitness values will be retrieved—and on the GenerationAverageFitness stat—from where the population’s average fitness value will be retrieved. The dependency amongst stat objects creates a chain, i.e., the value of the GenerationFitnesses stat is calculated first, and its information is then used by the GenerationAverageFitness. Finally, the information of both GenerationFitnesses and GenerationAverageFitness stats is used by the GenerationStandardDeviationFitness stat. This is illustrated in Figure 2.

4. GENERATING OUTPUT BASED ON STATS INFORMATION

So far we have discussed how information about the execution of an evolutionary algorithm can be created and stored based on events. In most cases, the user is interested in seeing this information in a friendly way, printing it to the screen or saving it to a file for further analysis, and also visualizing it in a graphical form.

The first approach that may come to mind (and probably the one with which most users will be familiar) is to add print statements in the code. It requires retrieving the stat object from the repository and adding the corresponding print statement (either to the terminal or a file). While this seems a simple approach, it lacks elegance and flexibility. Firstly,
forms of output may be required, e.g., sending the information through a network connection or presenting the information in a graphical form.

In EpochX, different output mechanisms can be easily implemented using a similar approach to the StatPrinter without incurring modifications to the code of the algorithm. For example, the StatPrinter can be used to send the information through a network connection instead of the screen or a file, by using a network-connected OutputStream. Graphical output can be generated with custom listeners that are triggered by events to update a GUI element based on the information of stat objects, taking advantage of both the event model and Stats system of EpochX. The listing below shows an example of a listener that updates a JLabel component based on the information of a stat object:

```java
public class LabelUpdater<T extends Event> implements Listener<T>
{
    private JLabel label;
    private Class<? extends AbstractStat<?>> klass;

    public LabelUpdater(JLabel label, Class<? extends AbstractStat<?>> klass)
    {
        this.label = label;
        this.klass = klass;
    }

    public void onEvent(T event)
    {
        label.setText(AbstractStat
                             .get(klass).toString());
    }
}
```

After creating a LabelUpdater object to update the value of a JLabel component, the listener must be registered in the EventManager to start receiving the events that trigger the update:

```java
JLabel generation = new JLabel();
EventManager.getInstance().add(
    EndGeneration.class,
    new LabelUpdater<EndGeneration>(label,
        GenerationNumber.class));
```

The above example shows how custom listeners can be created to update GUI components using the information from stat objects. Specialised listeners (e.g., listeners that take information from multiple stat objects) can be implemented to display more elaborate output, such as graphs showing the progress of the best fitness value over the generations. There are no limits in the number of listeners used and they can be combined with the StatPrinter and other output mechanisms, providing different visualisation of the run information.

5. A COMPLETE EXAMPLE

This section includes a typical example of how the EpochX framework can be used to evolve solutions to the even-5 parity problem. Even parity is one of the benchmark problems...
that is included in the framework, because it is so frequently
used in genetic programming research. However, other prob-
lems can be easily implemented by using an appropriate fit-
ess evaluator.

The configuration repository is set up with the control pa-
rameters for solving the problem, including a list of genetic
operators, an initialisation procedure and settings for pa-
rameters such as population size and maximum depth. Ter-
mination criteria are also set to use both a maximum num-
ber of generations and a fitness target. A run will progress
until one or more of the termination criteria are met. To
guid output about the progress of the run, a StatPrinter is
created to print information each generation. Finally,
the runs are started by constructing an Evolver instance
and calling its run method. The listing below shows the se-
quence of statements required to run a GP for the even-5
parity problem:

```java
// initialises default configuration
Config config = Config.getInstance();
config.defaults();

// defines the problem with fitness function
Problem problem = new EvenParity(5);
config.set(Initialiser.METHOD, new RampedHalfAndHalf(prob, problem.getFunctionSet(),
problem.getTerminalSet()));
config.set(FitnessEvaluator.FUNCTION, problem);

// sets control parameters
config.set(Population.SIZE, 100);
config.set(Crossover.PROBABILITY, 0.9);
config.set(Reproduction.PROBABILITY, 0.1);
config.set(BranchBreeder.ELITISM, 0);
config.set(TreeFactory.MAX_DEPTH, 17);
config.set(TreeFactory.INITIAL_DEPTH, 6);
config.set(TournamentSelector.TOURNAMENT_SIZE, 7);
config.set(MaximumGenerations.MAX_GENERATIONS, 50);
config.set(TerminationFitness.TARGET, new DoubleFitness(0));

// sets genetic operators
List<Operator> operators =
new ArrayList<Operator>();
operators.add(Reproduction());
operators.add(SubtreeCrossover());
config.set(BranchBreeder.OPERATORS, operators);

// sets termination criteria
List<TerminationCriteria> criteria =
new ArrayList<TerminationCriteria>();
criteria.add(new MaximumGenerations());
criteria.add(new TerminationFitness());
config.set(GenerationalStrategy.TERMINATION_CRITERIA, criteria);

// defines the stats to print
StatPrinter printer = new StatPrinter();
printer.add(GenerationNumber.class);
printer.add(GenerationBestFitness.class);
printer.add(GenerationAverageFitness.class);
printer.add(GenerationFitnessVariety.class);
printer.printOnEvent(EndGeneration.class);
```

6. CONCLUSION

We have presented an overview of EpochX, a Genetic Pro-
gramming framework which exploits Java’s object-oriented
feature set to provide a rich set of methods for analysing
volutionary algorithm runs. We have demonstrated the
use of EpochX’s event and statistics management facilities
for analysing GP runs in an efficient and flexible way. The
event-driven Stats system is highly customizable, making it
easier to gather a vast range of information about the exe-
cution of an evolutionary algorithm in a convenient format.
EpochX is available for download, including source code

7. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the financial support
from the EPSRC grant EP/H020217/1.

8. REFERENCES

[1] ECJ: A Java-based Evolutionary Computation
program initialisation in genetic programming. Genetic Programming and Evolvable Machines,
imperative program trees with strongly formed genetic
programming. In Proceedings of the 15th European
Conference on Genetic Programming, EuroGP 2012,
2012.
Programming of Computers by Means of Natural
IEEE Transactions on Evolutionary Computation,
programming. In Proceedings of the Workshop on
Genetic Programming: From Theory to Real-World
Genetic Programming and Evolvable Machines,