University of Exeter

College of Engineering, Mathematics and Physical Sciences

ECM2414

Software Development

Continuous Assessment

Date Set: 10th November 2014
Date Due: 12th December 2014

This CA comprises 30% of the overall module assessment.

This is a **pair** exercise, and your attention is drawn to the guidelines on collaboration and plagiarism in the College Handbook
(`https://student-harrison.emps.ex.ac.uk/`).

This assessment covers the use and implementation of a range of software development techniques and constructs covered in ECM2414, including nested classes, concurrency, generic programming, testing, and pair programming. The assignment is *summative*. Please ensure you read the entire document before you begin the assessment. Note there is an intermediate deadline on 17 $^{\text{th}}$November.

Please Turn Over

1 Development paradigm

As mentioned in the lecture series, pair programming is a popular development approach, primarily associated with *agile* development, but is used across the software industry. In pair programming, two software developers work together to generate the solution to a given problem. One (the *driver*) physically writes the code while the other (the *observer*) reviews each line of code as it is generated. During the solution development, the roles are switched between the two programmers regularly. The aim of the split role is for the two programmers to concern themselves with different aspects of the software being developed, with the observer considering the strategic direction of the work (how it fits with the whole, and the deliverables), and the driver principally focussed on tactical aspects of the current task at hand (block, method, class, etc.), as well as allowing useful discussion between the developers regarding different possible solutions and design approaches.

Research into pair programming has indicated that it leads to fewer bugs and more concise (i.e., shorter) programs. Additionally, it also facilitates knowledge sharing between developers, which can be crucial for a software house, often pairs are generated by cycling through developers on a team, so everyone is eventually paired with everyone else at some point.

This assignment will introduce you to the paired programming approach in a practical fashion, through the development of a threaded Java program. As such you will need to form pairs. If the numbers mean there needs to be a single group of three, this team will be assessed more stringently according to the mark scheme to correspond with the advantage of having an extra person. On physical submission you will need to attach the BART sheets of both/all members to the document. Only a single member should submit electronically (though the student submitting electronically must correspond to the student whose BART sheet is attached to the very front of the hardcopy submission).

Each and every student must inform the module coordinator (Dr Fieldsend), via email who their pair partner is by midday on 17 $^{\text{th}}$November, or, if you cannot find a partner please email the module coordinator to this effect by the same deadline. The subject line of both these types of email should be "ECM2414: partner selection". Individuals who miss this deadline will incur a penalty of 10 marks.

Individuals who email the coordinator that they cannot find a partner, or who miss this deadline, will be placed into pairs by the module coordinator. In the case of an odd number of students on the module, the last three students to email the module coordinator (or fail to email), will be placed in a group of three.

2 Assignment

A popular optimisation heuristic used in computer science is the *genetic algorithm* (GA). This *nature inspired* stochastic approach mimics the biological evolutionary process, via maintaining a *population* of design solutions, which are recombined (crossed-over) and mutated from one generation to the next, in order to improve their performance in their environment (i.e. improve the design performance). This is facilitated by preferentially recombining *fitter* design solutions (those which receive a higher value on the problem function), or by maintaining a separate pool of the best designs seen so far, and recombining the search population at each generation with members from this *elite* population.

In this assignment you will be developing an elitist *multi-threaded* GA to optimise *generic* problems. This will use a number of classes provided as part of the assignment, as well as needing you to develop your own classes. The processing in a GA is traditionally seen as a sequence of routines for the transformation and assessment of the population it maintains. A population is initialised, evaluated, ranked, and the population members are *recombined* to create the next generation, which is then evaluated, ranked, recombined, etc. The process keeps looping in this fashion until some stopping criteria are reached (typically a limit on the total number of evaluations of the problem).

You are tasked with implementing a GA which is *multi-threaded* so that it can, for example, exploit multiple cores. This will take a different form to the traditional GA as certain aspects may therefore happen in parallel. The general design is shown in Figure 1. In the multi-threaded situation, each population member is a separate thread. Once these threads have started they will *each* sequentially:

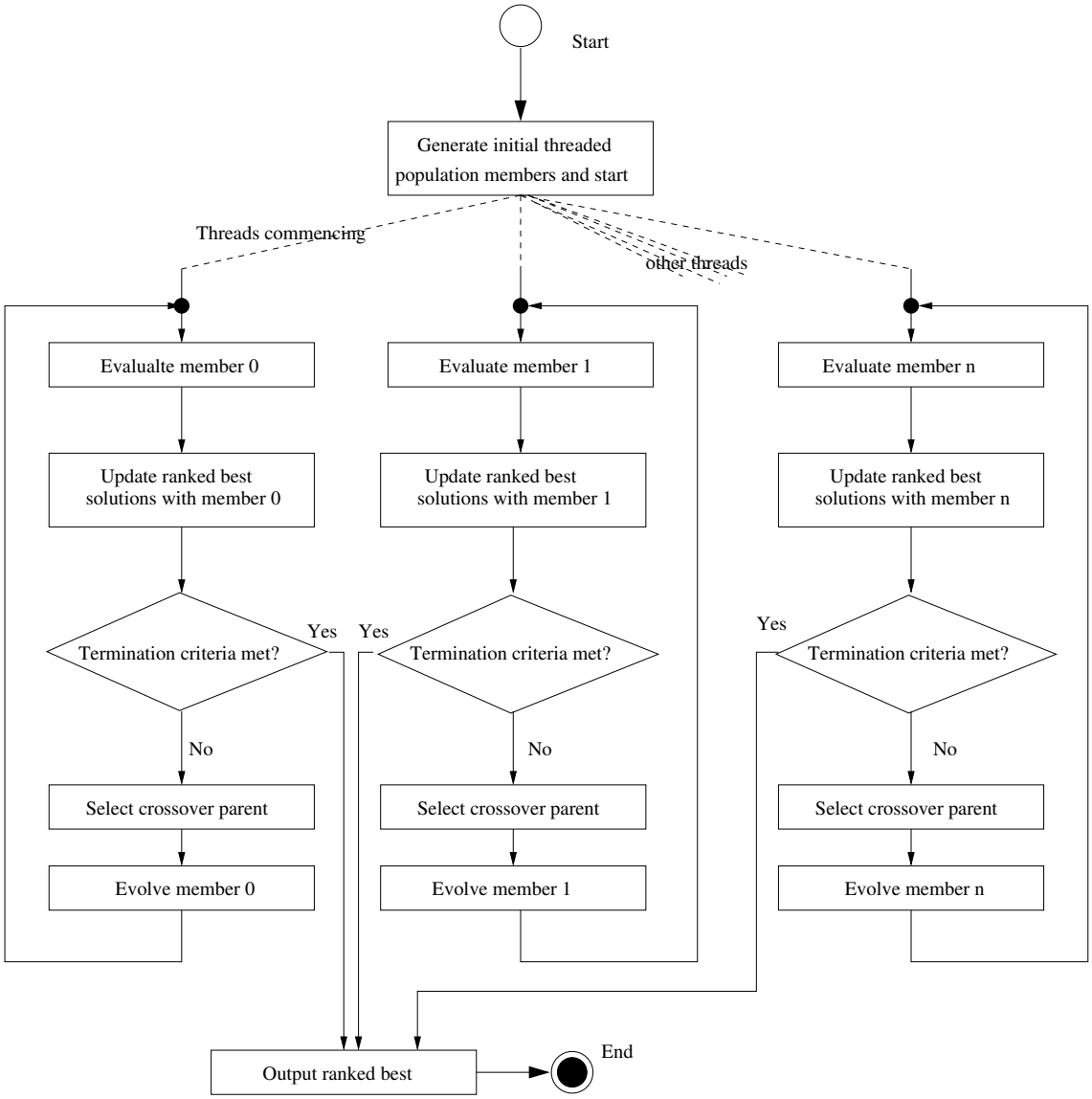

Figure 1: The multi-threaded genetic algorithm. The evaluation method will be provided by implementor of the `Problem` and the evolve method is provided in the `Design` class (both provided).

1. Evaluate their current design on the problem function.
2. Update the elite top ranked solutions (designs) maintained across threads.
3. Check if the termination criterion has been met: if it has been met, output the best ranked solutions and exit the optimiser, otherwise proceed to 4.
4. Select a parent member from the elite top ranked solutions to *crossover* with the current solution maintained by the thread. If the elite top ranked solutions are currently inaccessible, rather than waiting the thread should instead engage with a *copy* of the most recent version of the elite top ranked solutions (maintained by the optimiser) to source a parent. You will need to consider how this is made thread safe.
5. Replace the current solution with an *evolved* solution (via crossover and mutation).
6. Go back to 1.

Each population member will therefore carry out some instructions which are independent of each other thread, and some which are dependent on some shared objects. It is your task to implement a class in the `ga` package called `MultiThreadedGeneticAlgorithm`, and any additional supporting classes you deem necessary. This must enable `MultiThreadedGeneticAlgorithm` to be run as a Java program at the command line, with the `Problem` name, the number of population

members, the number of best solutions maintained, the crossover rate, the mutation rate, and the total number of problem evaluations completed before termination as arguments. For instance, if I were to invoke the following at the command line (given that `MultiThreadedGeneticAlgorithm` has been specified as the main class in the `ga` package jar file):

```
>> java -jar ga.jar EngineDesignProblem 20 50 0.2 0.1 10000
```

the `main` method in `MultiThreadedGeneticAlgorithm` would be processed, which would try and create an instance of the class named `EngineDesignProblem` which should implement the `Problem` interface (the `Helper` class provided as part of the CA will aid you with this aspect). It should then start 20 threads concerned with evolving 20 designs, each of which has a crossover probability of 0.2 and a mutation probability of 0.1. The GA will complete once it has completed 10000 function evaluations (at which point it will save the 50 best designs it has found as an array into a serialised file called '*designs.ser*', and also write a text file called '*results.txt*' containing the corresponding 50 best values, one on each row, ordered from best (highest) to worst (lowest)).

I would recommend that as part of this assessment you examine the published API of the member classes of the Java collections framework – there are a number of classes and methods you will find useful there.

There are a number of constraints you need to ensure are adhered to for your program to run, otherwise you need to inform the user:

1. The class name entered must be a subtype of `ga.Problem`
2. The population size must be ≥ 1
3. The elite set size must be ≥ 1
4. The crossover probability must be on the range $[0, 1]$
5. The mutation probability must be on the range $[0, 1]$
6. The number of function evaluations termination value must be $\geq$ the elite set size

There are no restrictions on the standard Java libraries that you may use. A number of pre-written classes to help you with this task are provided at the end of this document, and are available to download from ELE. **You must not alter these classes. If you change these provided package members you will receive a mark of 0 for the operation mark in your assessment, as these classes form part of the published API which you must adhere to. You do not have to include these classes in the hardcopy printout of your submission, but they must be in the electronic submission of the executable 'jarred' `ga` package.**

3 Submission

You will be expected to submit a copy of your finished program electronically, using electronic submission at empslocal.ex.ac.uk/submit, to the folder 'ECM2414 summative CA', in an executable jar file named ga.jar. The jar file should include both the bytecode (*.class*) and source files (*.java*) of your submission.

You must also hand in a paper submission, using BART, to the student services office. The paper submission must include:

- A cover page which details how you would like the final mark to be allocated to the developers, based upon your agreed input (i.e. 50:50 if both parties took equal roles, or perhaps 55:45 if you both agree that one party may have contributed a bit more than the other – do remember however that in pair programming both the driver and observer roles are vital, and should be switched frequently between developers). The maximum divergence allowed is 60:40, although non-contributors will receive a zero.
- A development log, which includes date, time and duration of pair programming sessions, and which role(s) developers took in these sessions, with each log entry signed by both members. If you are working individually for parts of the project you <u>are not</u> conducting pair programming – therefore all log entries must include both students.
- A printout of the source files you have written for the `ga` package. Feedback will be written directly upon this paper copy of the source code on specific code blocks/structure.

4 Marking Criteria

This assessment will be marked using the following criteria.

Criteria	**Comments (ideal submission)**	**Marks Available**
	The Electronic Submission (direct feedback will also be written on the source in the physical submission)	
Documentation comments & annotations.	The Java source files submitted have the appropriate JavaDoc comments and annotations inserted.	/5
Code comments.	Code comments are useful and informative, and at the appropriate level (i.e., it should not contain spurious comments, or ones that to not serve an explanatory purpose).	/5
Implementation.	The degree to which the code is well structured and presented, with a coherent design and clear and appropriate management of object states. The degree to which classes are thread-safe where required, and thread management and communication is accomplished cleanly, efficiently, and without potential error. The degree to which the use of data structures is appropriate, along with the use of other Java constructs and techniques, and adhered to conventions.	/15 /25 /10
Operation.	The submitted jar file is directly executable, and performs correctly across a range of valid inputs, and deals effectively with invalid inputs as specified in the CA document and in the provided `ga` package members.	/40
Pair weighting.	Non-submission of cover page with weightings will lead to a 50:50 weighting being used.	
Pair Penalty.	Non-submission of development log.	−10
Individual Penalty.	Failure to email coordinator by 17th November deadline.	−10

5 Provided ga package members

The `Problem` interface – all problems passed to the optimiser must implement this interface.

```
package ga;

import java.util.ArrayList;

/**
 * Any design problem to be solved by the MultiThreadedGeneticAlgorithm
 * must implement this interface. Additionally, all implementors must provide a
 * no-argument constructor.
 *
 * @author Jonathan Fieldsend
 * @version 1.0
 */
public interface Problem
{
    /**
     * Method to enable the evaluation of the given Design on this problem
     *
     * @param  d   A Design containing the design to evaluate on this problem
     * @return a Number containing the corresponding evaluation
     */
    Number evaluate(Design d);

    /**
     * Method to generate a random boolean vector (representing a binary string)
     * of the appropriate length to be used by this problem
     *
     * @return a random design vector
     */
    ArrayList<Boolean> getRandomDesignVector();
}
```

The `Helper` class – provides a static method to enable the generation of a `Problem` instance given a String with its name (enabling the `Problem` to be resolved at runtime).

```
package ga;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

/**
 * The Helper class is soley to provide a static method to help with the conversion of a String
 * passed to your program into a valid Problem instance. It exploits reflection, which we will
 * see more of toward the end of the ECM2414 lecture series
 *
 * @author Jonathan Fieldsend
 * @version 1.0
 */
public class Helper
{
    /**
     * Given the name argument, the method will attempt to initialise a Problem instance which
     * corresponds to the name (and will through an exception if the name does not match an
     * available class, is not a problem instance, does not provide a no argument constructor
     * as required by the Problem API, etc.)
     *
     * @param name a String holding the fully qualified name of the Problem class to consruct
     * @return a Problem instance of the type specified in name
     * @throws GAInitiationException if there are any problems generating the requested class
     */
    public static Problem getProblem(String name) throws GAInitiationException {
        /*
         * Use reflection to create instance of a Problem implementor used by the optimiser
         */
        Object object = null;
        try {
            Class<?> c = Class.forName(name);
            Constructor<?> constructor = c.getConstructor(); // get no argument constructor
            object = constructor.newInstance();
        } catch(IndexOutOfBoundsException e) {
            throw new GAInitiationException("Must invoke MultiThreadedGeneticAlgorithm program "
                + "with an argument containing the class name of the ga.Problem implementor to "
                + "be used by the GA");
        } catch(ClassNotFoundException e) {
            throw new GAInitiationException("Class definition not found matching name entered");
        } catch(NoSuchMethodException e) {
            throw new GAInitiationException("All implementors of the ga.Problem interface must "
                + "provide a public no-argument constructor");
        } catch(InstantiationException e) {
            throw new GAInitiationException("Problem encounterd generating an instance -- have "
                + "you checked that it is not an abstract class or an interface you have "
                + "entered the details for?");
         } catch(IllegalAccessException e) {
            throw new GAInitiationException("Problem encounterd generating an instance -- have "
                + "ensured the correct path to the compiled bytecode of the specified class "
                + "file was entered/is accessible");
```

```
        } catch(InvocationTargetException e) {
            throw new GAInitiationException("Problem encounterd generating an instance -- "
                + "Constructor threw an exception. Details follow: ... " + e.getMessage());
        }

        if (object instanceof Problem) {
            return (Problem) object;
        }
        throw new GAInitiationException("Class name entered must be a subtype of ga.Problem");
    }
}
```

The `GAInitiationException`, which will contain any error messages generated when using the `getProblem` method of the `Helper` class.

```
package ga;

/**
 * ClassNotProblemTypeException, holds details when a type name is passed for use
 * but it is not a Problem subtype.
 *
 * @author Jonathan Fieldsend
 * @version 1.0
 */
public class GAInitiationException extends Exception
{
   /**
    * Constructor accepts a message to be contained in this instance
    *
    * @param message the message to contain
    */
   public GAInitiationException(String message) {
       super(message);
   }
}
```

The `Design` class – encapsulates the design in a binary array, and provides methods for the evaluation of a `Design` and its comparison with other `Design`s.

```
package ga;

import java.util.ArrayList;
import java.util.Random;
import java.io.Serializable;

/**
 * The Design class encapsulates a representation of a solution to a Problem, and provides
 * methods to compare to other Designs
 *
 * @author Jonathan Fieldsend
 * @version 1.0
 */
public class Design implements Comparable<Design>, Serializable
{
    // members for comparison and random number generation
    private static final int SMALLER = -1;
    private static final int EQUAL = 0;
    private static final int BIGGER = 1;
    private static final Random rng = new Random();
    // state members
    private Number value; // value (quality) of this design
    private Problem problem; // problem being solved
    private ArrayList<Boolean> designVector; // solution representation

    /**
     * Constructs this design initially with a random solution for the given Problem argument
     *
     * @param problem Problem that this design will be tackling
     */
    Design(Problem problem){
        this.problem = problem;
        this.designVector = problem.getRandomDesignVector();
    }

    /**
     * Evaluates this design on its problem
     */
    synchronized void evaluate() {
        if (this.value == null)
            this.value = this.problem.evaluate(this);
    }

    /** {@InheritDoc}
     */
    @Override
    public int compareTo(Design otherDesign){
        if (this == otherDesign)
            return Design.EQUAL;
        if (this.equals(otherDesign)) //need to ensure consistency with equals
```

```
            return Design.EQUAL;
        if (this.value.doubleValue() < otherDesign.value.doubleValue())
            return Design.SMALLER;
        else if (this.value.doubleValue() < otherDesign.value.doubleValue())
            return Design.BIGGER;
        return Design.EQUAL;
    }

    /** {@InheritDoc}
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (this == obj)
            return true;
        if (obj instanceof Design)
            if (this.designVector.equals(((Design) obj).designVector))
                return true;

        return false;
    }

    /** {@InheritDoc}
     */
    @Override
    public int hashCode(){
        return designVector.hashCode();
    }

    /**
     * Method returns whether this design stored in this Design has been evaluated yet
     *
     * @return true if evaluated, false otherwise
     */
    boolean isEvaluated() {
        return value == null;
    }

    /**
     * Method modifies the design stored in this Design be evolving it using another Design
     * with the provided crossover probability, and then mutates elements of the child
     * produced with the mutation probability. This child will replace the design held in this
     * Design.
     *
     * @param otherDesign design to use as the other parent alsongside this design
     * @param crossoverProb probability of crossing-over an element from the otherDesign,
     * must be on the range [0,1]
     * @param crossoverProb probability of mutating an element from this design,
     * must be on the range [0,1]
     */
    void evolve(Design otherDesign, double crossoverProb, double mutationProb) {
        this.crossover(otherDesign, crossoverProb);
        this.mutate(mutationProb);
        this.value = null;
    }

    /*
     * Method crosses over this design with the otherDesign with crossoverProb probability
     * for each element (uses Uniform Crossover)
     */
    private void crossover(Design otherDesign, double crossoverProb){
        for (int i=0; i < this.designVector.size(); i++){
            if (this.rng.nextDouble() < crossoverProb){
                this.designVector.set(i, otherDesign.designVector.get(i));
            }
        }
    }

    /*
     * Method mutates this design with mutationProb probability (uses bit flip mutation)
     */
    private void mutate(double mutationProb){
        for (int i=0; i < this.designVector.size(); i++){
            if (this.rng.nextDouble() < mutationProb){
                if (this.designVector.get(i)) {
                    this.designVector.set(i, false);
                } else {
                    this.designVector.set(i, true);
                }
            }
        }
    }
}
```