An Object Oriented Approach to a Population Dynamics Model

Matthew Hambley
Supervisor: Kevin Stratford

August 23, 2007

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2007
Abstract

The project described in this dissertation investigated the applicability of object oriented design and programming techniques to high performance computing by using them to develop a population dynamics model. The problem is modelled using an individual based approach rather than the continuum based approach currently in common use. There was insufficient time to reproduce all facets of the established model or to implement parallel execution however I conclude that the work presented shows strong promise for future development. My simulation was compared with the continuum model for simple cases of population growth and migration in one and two dimensions. I conclude that the techniques used in the design an implementation processes were effective and worth considering in a high performance computing context.
Contents

1 Introduction 1
  1.1 Anthropology ............................................. 1
  1.2 Population Dynamics .................................. 2
  1.3 Object Orientation .................................... 3
  1.4 Synopsis .................................................. 4

2 Design 5
  2.1 Requirements ............................................ 5
  2.2 Class Model .............................................. 5
    2.2.1 People .............................................. 6
    2.2.2 The Environment .................................... 8
    2.2.3 Mechanics .......................................... 8
  2.3 Parallelism .............................................. 11
    2.3.1 Communication ..................................... 11
    2.3.2 Halos ................................................. 13
  2.4 Summary .................................................. 13

3 Implementation 14
  3.1 People .................................................. 14
  3.2 World Geometry ......................................... 15
  3.3 Diffusion ................................................. 16
  3.4 Order of Execution ..................................... 20
  3.5 Unit Testing ............................................ 20

4 Results 23
  4.1 Population Growth ...................................... 23
  4.2 Migration in 1 Dimension ................................ 24
  4.3 Migration on a 2-D Cartesian Plane ...................... 26
  4.4 Migration on a 2-D Spherical Surface ................... 29

5 Conclusions 31
  5.1 Further Work ............................................ 31

6 Post Mortem 33
## A Conventions

A.1 Unified Modelling Language .................................................. 36
A.2 Naming Conventions ............................................................... 38

## B Source code README

39
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Original work plan.</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Original risk assessment.</td>
<td>34</td>
</tr>
</tbody>
</table>
List of Figures

2.1 A diagram showing Person and related classes . . . . . . . . . . . . . 6
2.2 The decorator pattern . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 A diagram of classes representing the environment . . . . . . . . . 9
2.4 A diagram of classes involved in the mechanics of the simulation . . 10
2.5 A class diagram of the complete system . . . . . . . . . . . . . . . . 12

3.1 Code fragment from Person.live() . . . . . . . . . . . . . . . . . . . 14
3.2 Code fragment showing Person.birthProbability() . . . . . . . . . 15
3.3 Code fragment showing Farming.birthProbabilityTerm() . . . . . 15
3.4 Explanation of Longitude and Latitude . . . . . . . . . . . . . . . . 17
3.5 An element of a Cartesian grid . . . . . . . . . . . . . . . . . . . . . 18
3.6 An element from the Northern hemisphere of a spherical grid . . . 19
3.7 The test method for the birthProbability method of Person . . . 21
3.8 The birthProbabilityTerm test method for Farming . . . . . . . 21

4.1 Population growth within a single world element . . . . . . . . . . . 24
4.2 Population shrinkage within a single world element . . . . . . . . 25
4.3 Migration of a population in 1 dimension with Easterly bias . . . . 25
4.4 Migration of a population in 1 dimension with Westerly bias . . . 26
4.5 Migration of a population in 1 dimension . . . . . . . . . . . . . . . 27
4.6 Population expanding into a 2 dimensional Cartesian space . . . . 28
4.7 Population expanding into a 2 dimensional spherical surface . . . 30

A.1 An example UML class diagram . . . . . . . . . . . . . . . . . . . . . 37
Chapter 1

Introduction

Scientific simulations, particularly those performed using high performance computer systems are predominantly implemented using functional programming languages such as Fortran or C. The first working Fortran compiler became available in 1958 [20][6]. Although C [13] was first described in 1978 [15] it is based on BCPL [17] which dates to the late 1960s [16]. Both these languages have developed over the years since then but they have not changed their fundamental procedural nature.

However, computer scientists and software engineers have been developing alternative approaches to computer programming languages. As early as the mid 60s [20] they were playing with a concept which became known as “object oriented” design and developing languages to implement such designs.

This project explores the applicability of such an approach to scientific simulation hosted on multiple processor machines. This is achieved by taking a population growth simulation implemented in the procedural style and redesigning it using object oriented techniques.

In the remainder of this chapter I shall introduce some of the theory and concepts on which my project relies. Firstly, I introduce the anthropological theory behind the simulation I have considered, then the simulation itself. Finally I give a summary of object oriented programming.

1.1 Anthropology

The problem is to model the geographical spread of a new, beneficial, technology and see how this affects the ethnic make-up of the region into which the technology is introduced.

The “wave of advance” model [11] proposed by Fisher to predict the spread of an advantageous gene was used by Ammerman and Cavalli-Sforza [8] to describe the spread of neolithic (10 000 - 5 000 BP) farming in Europe. Later work by Cohen [10] allowed
for the fact that geographical features such as desserts and mountain ranges affect pop-
ulation movement. More recently work has been carried out [5] to model the effect new
technology has on migration and ethnicity within a region. It is this model which I have
used as the basis for my work.

For example, the simulation used by Signitzer[18] models population expansion fuelled
by the spread of technology. It is based on the hypothesis that a population which has
knowledge of a superior technology, e.g. farming, out-compete an indigenous popula-
tion without it. This allows the bearers of the new technology to expand into indigenous
population’s land. However a “bow wave” of indigenous people who have learned the
new technology will build up in front of the advancing immigrants. Eventually this bow
wave becomes so great that it halts the progress of the immigrants. After this point the
technology continues to spread among the indigenous population but the encroachment
of the immigrants is halted.

As I intend to investigate programming techniques I shall be using a vastly simplified
version of this model. All it is required to do is simulate a population expanding into an
unpopulated area. However, the design is such that it should support the more compli-
cated model of the original work.

1.2 Population Dynamics

The simulation presented in [18] is based on a continuum model. The new farming
population diffuses into the hunter-gatherer population like a liquid would diffuse into
another in a fluid dynamics model. A partial differential equation known as the Fisher
Equation [11] is used to model this diffusion:

$$\frac{\partial N}{\partial t} = D \nabla^2 N + \frac{1}{\tau} N \left( 1 - \frac{N}{N_s} \right), \quad (1.1)$$

where $N$ is the population density and $N_s$ is the saturation density. $\tau$ is related to birth
rate and $D$ is the diffusion constant. The first term is the “diffusive term” and represents
population spread due to random movement, similar to Brownian motion. The second
term is the “logistic growth term” and represents the growth of the population in a
particular area, capped by that areas ability to support a population.

This model gives good results for a large, stable population but it suffers from the fact
that it treats a population of individuals as a continuous quantity. This can lead to cu-
rious results. For instance if the population is subject to an extinction event such as
volcanism or plague it is entirely possible for the Fisher equation to produce a popula-
tion of 0.5 people. What is more it is possible for this half-a-person to reproduce and
repopulate the area.

Clearly this does not accurately model real life. What is needed is a model which
considers the population to be a collection of individuals. This would more accurately
1.3 Object Orientation

Early on in the development of software engineering it became apparent that encapsulation, in the form of functions and procedures, had a number of advantages. These include easy reuse of blocks of code without duplication and clearer source code leading to better maintainability. They also allow variables to be declared local to a functional block, protecting them against interference from elsewhere in the program.

This idea was developed further with the concept of the abstract data type or ADT. An ADT is a collection of private data, private functions and public functions. The only way to access the data from out with the ADT is via the public interface functions. Effectively, it is data which knows how to operate on itself. This allows strict controls to be placed on how the data are changed, preventing it becoming corrupted. It also provides a clearly defined and well understood interface which aids re-use and hides details of implementation.

It is possible to implement ADTs using traditional procedural languages but it can be an involved process. Because of this, special languages providing tools to ease the use of ADTs were developed. These languages quickly gained features beyond simple ADTs and became known as Object Oriented (OO) languages.

One of the first languages to support ADTs was SIMULA, developed in the 1960s [20]. Later, the U.S. military developed and standardised a new language, Ada, which also allowed ADT programming [20]. Possibly the first language to start adding to the concept of the ADT was Smalltalk. This was developed at Xerox PARC (Palo Alto Research Centre) as part of their work on graphical user interfaces in the 1970s [20].

The most fundamental feature of any OO language is the class. This describes an abstract data type. The variables contained within a class are known as member variables while the functions contained within are commonly referred to as methods or member functions. At run-time a program will generate, or instantiate, an instance of a class. The class is a template from which an object can be made.

This description shows how an OO language can be used to easily create abstract data types. However, there are a number of very powerful features beyond that simple mapping. The two most important are inheritance and polymorphism.

Inheritance allows a class to be derived from another. This may be thought of as an “is a kind of” relationship. e.g. Car is-a-kind-of WheeledVehicle. It implies that the child, or derived, class has all the properties of the parent class plus specialist properties unique to itself.

Polymorphism is the ability of a child class to be treated as though it were its parent class. So an object of type Car may be assigned to a variable of type WheeledVehicle.
and all methods which are native to WheeledVehicle may be called on that Car object. This means that it is possible to have a collection of objects which are all derived from the same parent without having to know precisely what they are. For instance a WheeledVehicle variable may, in fact, be holding a Bicycle object.

1.4 Synopsis

The aim of this project was to explore the utility of object oriented design and implementation when applied to a population migration model. This was thought a valuable avenue of investigation as some current models do not match the reality of such a process very closely. In the following chapter I discuss the design of an individual-based simulation of population dynamics.
Chapter 2

Design

In this chapter I outline the requirements, both functional and technical, of this project. Following this are sections discussing the design of various features of the project.

2.1 Requirements

Before a software project may be designed it is essential to decide on the functional requirements of the system. These describe the desired capability of the system with no reference to how such capabilities might be implemented.

The functional requirements for the simulation were identified as follows:

- Simulate the growth and migration of a population into an empty, uniform environment.
- Utilise a multi-processor system to allow larger problems to be tackled or greater speed of simulation if the size is specified.

Clearly there is no reason why such a system may not be implemented using a continuum model and written using a procedural language. However to do so would be duplicating work already carried out by others. The aim of this project is to explore the possibilities of an individual based model implemented using OO design and programming languages. These become technical requirements of the project.

2.2 Class Model

At its heart, object oriented design is about breaking the problem down into a series of objects. These objects interact with each other to perform the desired task. A useful first step in this process is to identify all the nouns and verbs involved in the problem. The
A person is affected by a number of Factor objects. This is modelled by a collection of references to singleton Factor objects. Shaded classes are not implemented in this project but show what is possible.

nouns give an indication of likely classes and the verbs an indication of likely methods. For example it is known that for each time-step a person lives they may reproduce or die. In this case "Person" is a noun and "live", "reproduce" and "die" are verbs. Analysis of this kind leads to the design shown in Figure 2.1.

### 2.2.1 People

A person is affected by a number of factors. These change the likelihood of that person migrating, reproducing or dying. The original simulation contained only two technologies, hunter-gathering and farming, which a person could be affected by. It is not unreasonable, however, to assume that more complicated simulations including a greater number may be desirable.

Furthermore farming obsoletes hunter-gathering so a person is only ever affected by one of them. However it is easy to imagine a simulation where a person may be affected

---

2 An introduction to UML is given in the appendix, section A.1.
A UML diagram of the decorator pattern. Redrawn from [12]

by a number of different, unrelated, technologies. The simulation has been designed in such a way as to not prevent this.

The obvious OO solution would be to sub-class Person with the technologies they know such that HunterGatherer and Farmer. This would work well in the simple case being examined for this report but consider the addition of another, unrelated, technology such as fishing. In that case you would need not only the classes mentioned above but also HunterGathererFisher and FarmerFisher. Now consider the addition of a fourth technology, the wheel. Now classes HunterGathererWheeled, HunterGathererFisherWheeled, FarmerWheeled and FarmerFisherWheeled are needed. Already things are becoming unwieldy. It is obvious that the number of classes needed increases rapidly with the addition of new technologies and all permutations must be conceived at the time of writing.

One design solution is to use the “Decorator Pattern”[12] as outlined in Figure 2.2 to allow technologies to be applied to a person. This operates at run-time and allows technologies to be added in any order. New technologies can be added easily without the need to preconceive all permutations of technology a person may know.

Unfortunately, the problem with this pattern is that its great strength is also its great weakness. The decorator pattern ensures that the Person object need not know what technologies it is wrapped in thus making the business of applying technologies quick and easy. When it comes time to pass on knowledge to a new person there is no way of knowing what technologies have been applied to the old person. Thus the transfer process is either impossible or highly complicated depending on the features of the chosen programming language. Java provides a facility called “reflection” which allows an object to determine features of its implementation but it is complicated and liable to
To avoid this, a much simpler approach was adopted whereby the Person holds a list of the factors affecting them. With this solution in place it becomes trivial for a Person object to know what technologies they know and, as a bonus, it saves memory as all people can point to the same factor object rather than requiring a fresh instance for each.

The technologies are implemented as “singleton” classes. These are classes which can not be instantiated directly in the normal manner. Instead they contain a static method which returns a reference to the same object. This approach is only valid because factors do not have any local state. The advantage of this over instantiating an object for each Person is that memory is only required for one copy of each factor.

2.2.2 The Environment

The people in the simulation do not exist in isolation though, they exist within the confines of an environment. Figure 2.3 shows how the world in which they live is modelled.

The world consists of an area or “patch”. This can be either rectangular area taken from a plane using Cartesian co-ordinates or it can be the area bounded by lines of longitude and latitude from the surface of a sphere.

Such an area is divided into equal sized elements. These behave differently depending on whether they are working in Cartesian or spherical co-ordinate space. They may also be “halo” elements. These elements are not involved in the computation but instead allow patches held on different processors to communicate with each other. Regardless of its type a world element holds a list of the people who live in that element.

2.2.3 Mechanics

There are also the mechanics of the program to consider and these are shown in Figure 2.4. Here I make use of polymorphism. Since the master node in a parallel program generally does all the things an ordinary node does, i.e. computation, plus extra organisational work it makes sense to make the derivation shown. Doing this allows the central object, Piltdown, to hold an object of type OrdinaryNode and make calls on it even if this particular instance happens to be a MasterNode.

There is also a need for a communication channel between neighbouring processors. This is implemented by MPISwitchboard. For testing purposes it is often useful to run the program on a single processor. To avoid the need for a lot of conditional code around communications a second child of Switchboard is provided which presents the same interface but does nothing.

3Named after the famous hoax because the simulation is full of fake cavemen.
Figure 2.3: A UML class diagram showing the classes used to describe the environment in which the population exists.
Figure 2.4: A UML class diagram showing the mechanical aspects of the simulation. The fact that MasterNode is a child of OrdinaryNode allows it to perform the work of an ordinary node plus the extra work of a master node. Deriving a working communication library and a dummy one from the same abstract class allows the developer to switch between them easily without the need to alter much code.
2.3 Parallelism

There are generally two reasons to consider parallel processing. Either a program is too slow for one processor to run in a reasonable time or it is too large to fit in the memory of one processing element. This project suffers both these problems.

A Person objects state consists of an age, an ethnicity, a reference to the world element in which it lives and a collection of references to affecting factors. This state may be represented by a float, an int, a reference and a set of references. Using indicative representations for these things gives a total of 120 bytes per Person. If the simulation then generates 10 000 of these, not a great number in population terms, memory usage is already over a megabyte. Creating an object in memory for each individual in the world rapidly fills up that memory. On the other hand having each notional individual make their own decision as to what to do next means non-trivial computation for each individual. This uses substantial processor time. Therefore it is necessary to consider a parallel approach to this problem.

A common feature of OO languages is polymorphism. This is the ability to specify an interface in a parent class but have each child class implement that interface in a way specific to that class. Extensive use of polymorphism has been made in designing the support for parallel operation.

2.3.1 Communication

Polymorphism can be seen in the design of the Switchboard interface. As an interface it is an abstract class (it can not be instantiated) with method definitions but no code of its own. The implementation of this interface is provided by the two derived classes DummySwitchboard and MPISwitchboard. Both these concrete classes can be referred to as though they were instances of Switchboard but behave very differently. The first does very little and is designed to allow the simulation to work on a single processor machine with no parallelism. The second is a wrapper around the MPI communications library and allows the simulation to work across multiple processors.

The reason for implementing communication this way is that it allows the simulation to work on a single processor or multiple processors by simply instantiating the appropriate Switchboard implementation. This is a one line change, none of the rest of the program needs to know how communications are handled. If a new messaging library standard were to emerge this simulation could be modified to use it by simply writing a new child of Switchboard which implemented the appropriate interface using this new library. The rest of the program need not be touched.
Figure 2.5: A UML class diagram of the complete system.
2.3.2 Halos

It is common to use the technique known as “halo swapping” to pass information between processors working on neighbouring areas of a parallel problem. This technique involves having a ring of extra cells surrounding the work area the processor is using. This ring or “halo” is a copy of data from neighbouring processors and is updated each simulation cycle. The simulation being discussed here uses this technique as it must be possible for migrants to migrate across work boundaries.

Normally these halo cells are identical to the working cells and it is up to the simulation to deal with the fact that some of the cells it has to work with are considered “special”. Polymorphism offers an opportunity to deal with this problem in a slightly more sophisticated way.

If HaloElement is derived from the WorldPatchElement class halo cells will behave differently without any requirement for the simulation to know the difference. For instance, the advanceTime() method of the Halo cells will be empty since we do not perform simulation on halo elements but the simulator can still call it as though it were a normal element. The method performGrowth may be used to communicate with neighbouring processors although, for reasons of efficiency, it may be desirable to form communications into batches and perform them at a later point.

2.4 Summary

The model presented in this chapter is simple but judicious use of polymorphic classes allows it to be easily extended at a future date. What’s more such extension is unlikely to require modification to what has been put in place already.

Notice also that intrinsic to its nature polymorphism removes the need for a certain amount of conditional code. In a procedural language a variable may be inspected and computation carried out based on the result of that inspection. In OO programs it is often not necessary to make the check. Instead a method on the object is called and behaviour intrinsic to the OO nature of the object will ensure that the correct code is called.
Chapter 3

Implementation

This chapter is concerned with the implementation of the design described in the previous chapter. It covers some of the problems encountered during the implementation process and their solutions. It also highlights some features of the final implementation.

3.1 People

The fundamental basis for the model is the notional people from which it is made. These are embodied in the class `Person`. The most important aspect of this class is that it implement the Fisher equation in such a way as to give the probability of a person reproducing, dying or moving.

This process is largely delegated to the various factors affecting that individual. Each individual stores a list of the factors currently affecting it and when required it interrogates each in turn for a probability. These probabilities are summed to give a final probability which is then used with a random number to find out if the event happens or not.

When a person reproduces they produce a clone of themselves having all the knowledge of their parent. This is shown in Figure 3.1. As previously stated the probability is cal-

```java
if (mEntropy.nextFloat() < birthProbability(deltaT)) {
    Person lBaby = new Person(mEthnicity);
    teachTechnology(lBaby);

    mLocation.addPerson(lBaby);
}
```

Figure 3.1: Code fragment from `Person.live()` showing how births are modelled.
public float birthProbability(float deltaT)
{
    float lProbability = 0;

    for (Factor knowledge : mInfluences) {
        lProbability += knowledge.birthProbabilityTerm(this, deltaT);
    }

    return lProbability;
}

Figure 3.2: Code fragment showing Person.birthProbability().

public float birthProbabilityTerm(final Person candidate, final float deltaT)
{
    return deltaT / mTau;
}

Figure 3.3: Code fragment showing Farming.birthProbabilityTerm().

calculated by summing terms from the varies factors affecting the person. This is shown in Figure 3.2. Finally Figure 3.3 shows how the farming factor calculates its contribution.

The calculation used in this final stage comes from the continuous Fisher equation which states that

\[ \frac{\partial N}{\partial t} = \frac{1}{\tau} N \left( 1 - \frac{N}{N_s} \right). \]

This can be expanded to give

\[ \frac{\partial N}{\partial t} = \frac{1}{\tau} N - \frac{1}{\tau} \frac{N^2}{N_s}. \]

Which may be thought of as a term for births: \( \frac{1}{\tau} N \) and a term for deaths: \( \frac{1}{\tau} \frac{N^2}{N_s} \). When used in a discrete model they become probabilities that a given event will happen in a given unit time. The chance of an individual reproducing is \( \frac{\Delta t}{\tau} \) and of dying \( \frac{\Delta t}{\tau} \frac{N}{N_s} \).

3.2  World Geometry

The world in which the population lives is divided into a regular grid of elements each of which is described by its centre point and its vertical and horizontal extent.
The simple case is the Cartesian world. This world exists as a flat plane or the surface of a cylinder. Co-ordinates are given as $x, y$ pairs and any area described by a centre and extent will be a rectangle. The height of such an area will be given by $\Delta y$ and the width by $\Delta x$. The area is the product $\Delta x \Delta y$.

As the primary use of this simulation is likely to be of the surface of a planet it has to deal with the complexities introduced by a spherical co-ordinate systems.

Any location on the surface of a sphere can be described by two values: Longitude and Latitude. (See Figure 3.4) Longitude is the angle $\theta$ between a fixed North-South line (the Greenwich Meridian for Earth), the centre of the sphere and the point of interest. Latitude is the angle $\lambda$ between the equator, the centre of the sphere and the point of interest. This is simple enough and is analogous to a Cartesian co-ordinate space in which $-180 \leq x \leq 180$ and $-90 \leq y \leq 90$. The problem comes when you consider a rectangular area on the surface.

The area of a rectangle will simply be $\Delta x \Delta y$. However this is not the case for an area on the surface of a sphere described by lines of longitude and latitude. Because the lines of longitude converge at the poles the "top" and "bottom" edges of any such rectangle will have different lengths. A rectangular element is actually trapezoidal.

### 3.3 Diffusion

The probability of a person migrating across the boundary of a world element is related to the length of the perimeter and the probability of a person migrating across any particular side of the element (up, down, left or right) is related to the length of that side. In the Cartesian case this probability is simple and related to $\Delta x$ and $\Delta y$.

So the probability of moving from 1 to 2 in the situation shown in Figure 3.5 is $p = \frac{D \Delta t}{\Delta x \Delta y}$. Here $D$ is a a fudge factor relating to and performing a similar function to the diffusion constant $D$ in the Fisher equation. As this is the probability for crossing any border the chance of moving out with the element is $4p$.

Under spherical co-ordinate system, on the other hand, things become more complicated. As observed above rectangular elements on the surface of a sphere are actually trapezoidal. This means the side furthest from the pole is longer and therefore there is a higher likelihood of a person migrating across that perimeter, away from the pole.

For the example shown in Figure 3.6 the probabilities look like this:

$$p_{12} = p_{14} = \frac{D \Delta t}{\Delta x_{centre} \Delta y}, \quad p_{13} = \frac{D \Delta t}{\Delta x_{north} \Delta y}, \quad p_{15} = \frac{D \Delta t}{\Delta x_{south} \Delta y}.$$  

A theoretical wave speed can be derived from the Fisher equation (Equation 1.1) and is given by $2\sqrt{\frac{D}{\tau}}$. This indicates how quickly a population may be expected to spread out.
Figure 3.4: A point on the surface of a sphere can be described by the angles $\theta$ and $\lambda$ which are also known as Longitude and Latitude.
Figure 3.5: An element of a Cartesian grid.
Figure 3.6: An element from the Northern hemisphere of a spherical grid.
3.4 Order of Execution

The first test of the migration mechanism was a simple 1-dimensional strip. The expected result was an initially Gaussian distribution around the centre which would become clipped at the maximum density as time progressed. This basic shape was observed but there was a noticeably higher rate of diffusion in an easterly direction. This skew was also observed on the 2-dimensional migration test.

Initial suspicion fell on an incorrect calculation of the probability of travelling east or west however this was discounted after some investigation.

In fact what was happening was that people who moved in the same direction as the order of processing (West to East) were being processed 2 or more times per $\Delta T$. This was rectified by making processing a 2 pass process. In the first pass all people make their decision as to whether they will move, reproduce or die. In the second pass the movement happens. This ensures that each person is only processed once per $\Delta T$.

3.5 Unit Testing

It is considered good software engineering practice to have a unit testing regime. That is, a series of test programs which exercise the interfaces of all the software components which make up a system. This is particularly applicable to OO programming where each class represents an easily tested component.

Unit testing is a “bottom up” process in that you attempt to ensure correct functioning of the system by ensuring that all the components it is made from function correctly. For instance this project uses a LongLat class to hold co-ordinates in longitude/latitude form. There is a test which makes sure that comparing two LongLats gives the correct answer. If this test passes and somewhere else an equality between LongLats fails the developer can be pretty sure that the equality itself worked so the problem must lie with values being compared.

The purpose of unit testing is to ensure correct behaviour during development but also to provide regression testing during maintenance. It is easy to check if a change has had unexpected consequences by re-running the unit test suite. This is particularly relevant in an experimental project such as this where frequent refactoring is likely to take place.

Given the advantages of unit testing and its particular suitability for OO programming it is not surprising that the Java community has embraced it with a number of tools. This project makes use of theJUnit[2] framework to ease the development and execution of unit tests. It also uses the Maven[4] build tool which integrates unit testing into the build process. This integration is so tight that if the unit test suite fails Maven will consider this a terminal error and exit. Thus it is not possible to build a release of a project unless it passes its unit test suite.
@Test public final void testBirthProbability()
{
    Person lUUT = new Person();
    assertEquals(0, lUUT.birthProbability((float)0.0), 0);
}

Figure 3.7: The test method for the birthProbability method of Person. Note that since just the Person class is being tested the Unit Under Test(UUT) has a zero percent chance of reproducing owing to having no factors working upon it.

@Test public final void testBirthrate()
{
    Farming lUUT = Farming.instance();
    assertEquals((float)1.0 / cTau, lUUT.birthProbabilityTerm(mDummy, 1), 0);
    assertEquals((float)2.0 / cTau, lUUT.birthProbabilityTerm(mDummy, 2), 0);
}

Figure 3.8: The test method for the birthProbabilityTerm method of Farming.

Of course it is still down to the developer to write the tests. It is possible to create a skeleton test suite which tests nothing. It is also possible to comment out failing tests such that a release may be built. However doing so defeats the point of having a unit test suite.

By way of an example Figure 3.7 shows the test for the method which returns the probability of a person reproducing. This person is not affected by any factors and is therefore unable to reproduce, therefore the expected result is zero. Figure 3.8 shows the test for one of the factors, farming, which gives a person a chance of reproducing. A number of checks are made within this test to make sure that the correct answer is not being returned by chance.

Ideally these test cases should be written before the implementation by a different programmer. They should be written beforehand to comprehensively exercise a module’s full functionality so as to ensure nothing is missed come implementation. They should be written by someone other than the person who will implement the module so as to prevent the module writer relying on knowledge of how the test works. In the case of a one-man effort it is clearly not possible to fulfil the latter requirement but the former is possible and I have endeavoured to do so throughout this project.

Some classes are not tested by my unit tests. This is largely due to pressure of time.
Because unit testing is a “bottom up” strategy it is most important to test the most fundamental classes. This means LongLat, which is used extensively by most of the simulation, and Person, Factor and its descendents. Person and its surrounding classes are fundamental to the simulation so it is important to ensure they are functioning correctly.
Chapter 4

Results

A series of increasingly complex integration tests are presented in this chapter and their results discussed.

4.1 Population Growth

Before anything more complex could be considered it was important to ensure that the population of a map element would grow in the correct manner. Therefore the first test was of a single map element containing a population of one individual. This was run for a suitable length of time and the resultant population growth compared against the theoretical curve calculated using the Logistic term of the Fisher equation.

When there is only one element in the world the probability of migrating out of this element is zero. In this case the Fisher equation is reduced to only its logistic term, that is, the term which controls population growth,

\[ \frac{\partial N}{\partial t} = \frac{1}{\tau} N \left( 1 - \frac{N}{N_s} \right). \]  \hspace{1cm} (4.1)

This may be stated as a function of \( t \):

\[ N(t) = \frac{N_0 N_s e^{\frac{t}{\tau}}}{N_s + N_0 \left( e^{\frac{t}{\tau}} - 1 \right)}, \]  \hspace{1cm} (4.2)

where \( N_0 \) is the population at \( t = 0 \).

Figure 4.1 charts the result of the simulation against theoretical curves calculated using the function shown in equation 4.2. As can be seen the general shape is a good fit to the theoretical curve. The discrete nature of the model produces noise which manifests as a fluctuation around the ceiling value.
Figure 4.1: Population growth within a single world element. \( H(x) \) is the theoretical curve for hunter-gatherers using values \( N_0 = 1, N_s = 100 \) and \( \tau = 50 \). \( F(x) \) is the theoretical curve for farmers using values \( N_0 = 1, N_s = 1000 \) and \( \tau = 20 \).

In a drought situation the land’s ability to support a population may reduce. This means it is important to check that the model correctly handles the situation where the initial population is greater than the saturation level. This is shown in Figure 4.2. Both the expected and actual results describe an exponential decline converging on the saturation level. Again the simulated population exhibits random noise about the saturation level.

4.2 Migration in 1 Dimension

Once I had satisfied myself that the growth term was working the migration term had to be checked. This was achieved using a single strip of elements, effectively a 1-dimensional world.

Each trace in Figure 4.3 shows the population density along the strip. There is a trace for each 100 years simulated. This allows the spread of the population to be observed. The chart should exhibit strong symmetry around 0° longitude.

As can be seen there is a slight bias towards the right or east. This would be a problem if it were systematic but Figure 4.4 shows a second run in which is slightly biased to the left or west. There are no differences to the initial conditions so any difference observed is due to the pseudo random number generator producing a different sequence each time.
Figure 4.2: Population shrinkage within a single world element. \( H(x) \) is the theoretical curve for hunter-gatherers using values \( N_0 = 2000, N_s = 100 \) and \( \tau = 50 \). \( F(x) \) is the theoretical curve for farmers using values \( N_0 = 2000, N_s = 1000 \) and \( \tau = 20 \).

Figure 4.3: Migration of a population in 1 dimension. For this simulation \( N_0 = 1 \), \( N_s = 1562 \) and \( \tau = 20 \). The scale is 100km per degree Longitude. Notice slight bias to the East.
Figure 4.4: Migration of a population in 1 dimension. For this simulation $N_0 = 1$, $N_s = 1562$ and $\tau = 20$. The scale is 100km per degree Longitude. Notice bias to the West.

The reason for this bias is that differences in the random number sequence are statistically significant at low population levels. If this sensitivity causes the central peak to become slightly off centre while the simulation is working with low population levels that bias will remain even after the population has become large enough to render similar changes insignificant. Figure 4.5 shows what happens if the starting population is much larger. In this case the centre of the population curve does not move and the resultant chart is much more symmetrical.

The final point to consider is the speed of the wave of advance. Archaeological evidence has been used to estimate this at 1km per year[7]. This is represented in the Fisher equation (Equation 1.1) by $D$ and takes an approximate value of 10km$^2$yr$^{-1}$. Unfortunately it is not clear how this constant with units $\frac{m^2}{s}$ translates to the discrete model. Instead trail and error were used to arrive at a value for mDiffusionConstant (found in the class Farming) of 5. This gives the expected speed of advance as shown in Figure 4.5 but it can not be concluded that this value is $\frac{D}{2}$.

### 4.3 Migration on a 2-D Cartesian Plane

Having verified 1-dimensional migration the next step is obviously 2-dimensional migration. This is first tested using a Cartesian grid since success is clear in that it is circular expansion.
Figure 4.5: Migration of a population in 1 dimension. For this simulation $N_0 = 1000$, $N_s = 1562$ and $\tau = 20$. The scale is 100km per degree Longitude.

Using a very large value for $N_s$ would help to highlight any problems as in the limit $N_s \to \infty$ the continuous model should be recovered. Unfortunately this also implies impractical memory usage. The other approach is to ensure that the logistic term is dominant. This can be achieved by reducing $\tau$ but again this causes too many objects to be created, overflowing memory. The other option is to reduce $D$, the diffusion constant. This reduces the speed of diffusion.

The effect of making the $D$ term one tenth of its normal value is shown in Figure 4.6. As can be seen it displays strong symmetry suggesting that there is no bias in migration. However it does display a slight diamond shape where a circle would be expected. This is due to a feature of the Fisher equation. If the inequality $\frac{D\tau}{\delta x \delta y} < 1$ is not met then this effect will be observed.

Since inter-process communication is not yet implemented any Person objects migrating outside the simulated area fall into halo elements and are no longer processed. This means they take no further part in the simulation but they are not destroyed either so just collect in the halo. If the population were allowed to expand out to the borders this would represent a waste of memory but the extent of the simulation space can be chosen to avoid this.
Figure 4.6: Population expanding into a 2 dimensional Cartesian space. $N_0 = 100$, $N_s = 1562$ and $\tau = 20$ years. The diffusion constant is $0.5$. The square area has is 600km along a side. Note the slight diamond shape.
4.4 Migration on a 2-D Spherical Surface

Once Cartesian migration was confirmed I turned to migration on a spherical surface. Due to the nature of the co-ordinate space on the surface of a sphere an egg-shaped migration is expected with more people migrating away from the poles than towards them.

As well as reducing the $D$ term as for the Cartesian case the test area was pushed as far North as possible, $90^\circ - \frac{\Delta w}{2}$, to maximise the asymmetry. The results may be seen in Figure 4.7.

The expected egg-shape is not obviously visible but there is a bias towards the South. I suspect that if the simulation were to be run for longer the expected shape would become apparent. Unfortunately this is not possible without the ability to run in parallel due to memory constraints.

An interesting property may be observed in Figure 4.7. The central section of the populated area is speckled due to variations in population. This does not happen in the continuum model in which the population of each world element will rise to saturation and remain there. However Figure 4.1 from earlier shows that the population level will match the saturation only on average. Instantaneously it may be above or below this level by a small amount.
Figure 4.7: Population expanding into a 2 dimensional spherical surface. North is to the top of the page. $N_0 = 100$, $N_s = 90$ and $\tau = 20$ years. The diffusion constant is 0.1 and the square area is 135 km on a side. Note the slight distention of the area away from the pole (top) and the variation in intensity of the body of the population.
Chapter 5

Conclusions

With this project I have successfully implemented a simple population diffusion model using an individual based approach rather than the more traditional continuum based one. I have done this using object oriented design techniques and an object oriented language.

I have shown that such an approach is possible and that it can produce results in line with expectations founded on the continuum model. I have also shown that this new model behaves differently to the continuum model at low population levels where random events can have a noticeable impact on the results of the simulation.

On the matter of software engineering I have shown that OO design techniques offer a viable way to consider scientific modelling problems. I have also shown a number of ways in which features of OO programming languages can be utilised to increase the power and flexibility of such a model. For example the use of polymorphic classes to describe influencing factors means addition influences may be added to the simulation with little or no alteration of the core program required.

5.1 Further Work

Clearly the model itself requires considerable work to bring it up to the level of the original continuum model. Currently there is no concept of geography such as mountains and bodies of water. This would be required before the concept of land fertility could be implemented. Geography should be relatively easy to implement. Each world element needs a height above the centre of the planet and the world itself needs the sea level. With this member data in place an element can work out if it is above or below sea level and can also calculate a fertility. It would also be necessary to modify the migration probabilities depending on the terrain. For instance there is zero chance of migrating across water while there is a reduced chance of migrating over a mountain range.

One of the features of the original work was the interaction between an immigrant,
technology bearing, population and an indigenous population. This is also not currently modelled by my work. The replacement of an indigenous population by more successful immigrants should simply be an inherent artifact of the model. What is yet to be implemented is a means for the indigenous population to learn the immigrant populations technology. It would probably be necessary to consult previous work for suggestions on how this should work.

Once all these matters were implemented it would be possible to take the model beyond what has currently been done. The most obvious addition would be that of new influencing factors. In particular new technology such as fishing and sailing which would allow higher populations on water edges and faster migration along rivers and enabling migration across seas to islands.

On the technical side the most important omission is parallelism. This model requires substantial memory and processing resources to run so running it on a parallel system would almost certainly be necessary to obtain useful results. I am happy that the design work carried out for parallelism is good but that can only be verified by implementation.
Chapter 6

Post Mortem

My initial work plan (Table 6.1) was optimistic, I have not advanced the simulation as far as I had hoped. However I had identified this as a risk (Table 6.2) and put processes in place to mitigate it.

The software engineering process known as “Extreme Programming” is particularly well suited to projects where either the goals or the work necessary are poorly understood. One of the fundamental tenets of this approach is to break the problem down into small, easily understood and completed, sections. The system is required to function in its entirety upon completion of each of these sections. This ensures that even if the deadline is reached before the system is complete there will, at least, be a body of completed and functioning work. Using this technique I was able to ensure that I had enough development completed to form the basis of a useful dissertation.

Another feature of exploratory programming such as required for this project is “refactoring”. Basically this means re-writing sections of the code, however it ranges from simple cosmetic changes to complete structural overhauls. For instance it may be necessary to rename a member variable or method in light of its changing usage or it may be necessary to break apart a class hierarchy, inserting new levels and abstractions.

Such refactoring can be achieved using a simple text editor but it can be time consuming and long-winded. This has lead to the development of various tools to automate aspects of this job. I have made extensive use of the Eclipse[1] integrated development environment which provides several refactoring tools including a automatic renaming and super-class extraction.

Possibly the most fundamental tool for any software project is a competent version control system. At its most basic, such a tool simply stores copies of the project as it stood at various points in development. This means that there is always a “good” copy of the code to return to if the working copy becomes unusable through equipment failure, accidental loss or inadvised refactoring. It also becomes easy to answer such questions as “What changed between my discovering a bug just now and 3 weeks ago when I know this worked.” All that is required is for the version control system to
<table>
<thead>
<tr>
<th>Week</th>
<th>Commencing</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28/05/07</td>
<td>Design object model and document it</td>
</tr>
<tr>
<td>2</td>
<td>04/06/07</td>
<td>Prepare development environment: Version control, build tools, unit testing harness, etc</td>
</tr>
<tr>
<td>3-5</td>
<td>11/06/07</td>
<td>Implement agents wandering randomly on a uniform map, test and document</td>
</tr>
<tr>
<td>6-8</td>
<td>25/06/07</td>
<td>Add population density and agents which seek low population areas, test and document</td>
</tr>
<tr>
<td>9-11</td>
<td>09/07/07</td>
<td>Add terrain and include it in agents decision making, test and document</td>
</tr>
<tr>
<td>12</td>
<td>23/07/07</td>
<td>Further testing, tweaking and contingency time</td>
</tr>
<tr>
<td>13-15</td>
<td>30/07/07</td>
<td>Final write-up</td>
</tr>
<tr>
<td>16</td>
<td>13/08/07</td>
<td>Prepare presentation</td>
</tr>
</tbody>
</table>

Table 6.1: Original work plan.

<table>
<thead>
<tr>
<th>Risk</th>
<th>Likelihood</th>
<th>Severity</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support libraries (e.g. JOMP, MPJ) insufficiently feature rich</td>
<td>Medium</td>
<td>High</td>
<td>Investigate capabilities before commencing on project</td>
</tr>
<tr>
<td>Project proves to be more complex than expected and can not be completed in time</td>
<td>Medium</td>
<td>Medium</td>
<td>Break project down into discrete sections so there is always a body of completed work</td>
</tr>
<tr>
<td>Non-availability of resources, e.g. Lomond</td>
<td>Medium</td>
<td>High</td>
<td>Ensure code can be developed on uni-processor machine</td>
</tr>
<tr>
<td>Loss of data</td>
<td>Low</td>
<td>High</td>
<td>Regular back-ups</td>
</tr>
<tr>
<td>Severe illness or incapacity</td>
<td>Low</td>
<td>High</td>
<td>Seek an extension.</td>
</tr>
</tbody>
</table>

Table 6.2: Original risk assessment.
compare the current working copy with its stored copy of 3 weeks ago.

I used the Subversion[3] version control system with a repository stored on a university system. I was able to access this repository from home over the public Internet.

Of course simply having a version control system in place does not protect development. It has to be used correctly. To this end I committed changes to the system regularly. Generally after each functional addition. Ideally only working code should be committed to the repository but this is not always possible.

Diligent use of version control was part of my mitigation strategy against data loss. Should my home system have failed I would only have lost the work carried out since the last commit. All work from before then would be available from my Subversion repository. The university machine on which that repository was stored was backed up to tape on a regular basis meaning failure of that machine would loose only that project history not stored on tape. I would still have my working copy on the home machine holding the most recent code so only historical code would be lost.

In conclusion I am happy with the progress made. I believe it represents a solid foundation to build on should this project be taken forward. My only regret is that it was not possible to implement the parallel processing aspect of the project. From a technical point of view this interests me most.
Appendix A

Conventions

This chapter describes the naming convention I use when naming classes, variables and the like. It also introduces the UML diagraming convention which I follow when drawing class diagrams.

A.1 Unified Modelling Language

It is often useful to represent software designs diagrammatically. This is particularly true of object oriented designs where the interaction between components can be quite complex. The software community has largely standardised on the diagram style known as "Unified Modelling Language"[9] or UML.

Figure A.1 shows a simple UML class diagram. Interface is an interface class meaning it declares a series of methods but does not implement them and has no member variables. Class is an abstract class meaning it has member variables and methods but may not be instantiated itself. It has a one-to-many "aggregation" relationship with Another. This means that a Class object holds a collection of references to zero or more Another objects. This collection is a member variable with the name mThings. Note also that it is a unidirectional relationship. In other words Class knows about Another but not the other way round.

The class SubClass is derived from Class and implements Interface. It has a one-to-one "composition" relationship with More. This means that a SubClass object "contains" a More object in some way. This is usually as a member variable. This relationship is bidirectional meaning that both classes know about the other.
Figure A.1: A simple UML class diagram exhibiting the main features of such a diagram. “Class” is abstract and has an aggregation relationship with “Another”. “Subclass” inherits from “Class” and implements “Interface”. It also has a composition relationship with “More”. 
A.2 Naming Conventions

There is a naming convention known as “Hungarian Notation” [19][14]. This was developed by a Xerox PARC employee and migrated with him to Microsoft. Variations of this scheme have become popular with developers.

At its heart it is about prefixing member names with various letters to indicate scope and type. As an example you may prefix all members with “m” to indicate that they are members. All arguments passed into member functions may be prefixed with “a”. Variables (both member variables and method arguments) may be prefixed with letters indicating their type, e.g. “i” for integer, “cf” for constant float.

While I do not like the use of type prefixes I find scope prefixes useful. Therefore I have used a similar convention throughout this project. Member variables are prefixed with “m” and local variables with “l”. Method arguments are left unadorned. Constants a prefixed with “c” instead of the more usual ALL CAPS representation.

Further more I shall also be observing the convention that class names are presented in so-called "Camel Case". That is each word in the name is given an upper-case letter. e.g. DurationOfEvent. This gives the resultant compound word a series of humps, like a camels back. Class member names also use camel case except their initial letter is lower-case. e.g. durationOfEvent. Local variables follow the same rule as class members.
Appendix B

Source code README

An Object Oriented Approach to a Population Dynamics Model
Matthew Hambley, Supervisor: Kevin Stratford
MSc in High Performance Computing,
The University of Edinburgh,
Year of Presentation: 2007

The tar-ball this README came in contains the source code written as part of the MSc project shown above.

To build the project code you will need a working installation of the Maven build tool. This may be found at <http://maven.apache.org/>.

Once the source has been extracted to a suitable directory simply issue the command "mvn test" to build it.

Once this has been done the test suite applications can be run using simple script files:

population.sh : Population growth in a single world element.
strip.sh      : Diffusion in a 1-dimensional world.
cylinder.sh  : Diffusion in a Cartesian 2-dimensional world.
patch.sh     : Diffusion on the surface of a sphere.
Bibliography


[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


