

Candidate Name: Owen Cox

Candidate Number: 57804

Degree Course: Computer Science

Supervisor: Phil Husbands

Alife System With A Novel Gene Model

2013

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: _____

Date: _____

Summary:

This report covers the software engineering process used to create the genome project. The report begins with the introduction, background and professional considerations, then the project plan. Section 5 is an informal description of how the user needs to be able to interact with the system. Next is the requirements documentation. After this is the Design document, followed by a discussion of the various implementation issues that came up, including reasoning about changes from the design and data structures used. Next is the testing section, followed by some demonstrations of the system. After this is the conclusion and the appendices.

Contents:

1. Introduction	6
2. Background	6
3. Professional Considerations	7
4. Project Plan	8
4.1 Introduction	8
5. Requirements	16
5.1 Introduction:	16
5.2 Desired Behaviour:	16
5.3 Specification:	17
5.4 User Interaction:	25
5.5 CRC Cards:	26
5.6 UML Class Diagrams:	31
5.7 Inheritance Diagram	34
5.8 Use Case Diagram	35
5.9 Use Case Scenarios	35
5.10 Acceptance Tests	37
6. Design Document:	40
6.1 Introduction:	40
6.2 Updated Class Diagram:	41
6.3 Changes:	44
6.4 Important Methods:	47
6.5 GUI Design:	51
7. Implementation:	53
7.1 Introduction:	53
7.2 Changes From Design:	53
7.3 Data structures:	55
7.3.1 WorldState:	55
7.3.2 Creature:	57
7.3.3 Gene:	58
7.3.4 WorldObject:	58
7.3.5 Tile:	58
7.3.6 OptionsMenuState:	58
7.3.7 JudgingState:	59

8. Testing:	60
8.1 Introduction:	60
8.2 Testing procedure:	60
8.3 Genome Cell Testing (#define CELLTEST in Simulation):	61
8.4 Genome Pattern Testing (#define PATTERNTEST in Simulation):	62
8.5 Genome Polling Testing (#define POLLTEST in Simulation):	65
8.6 Need Fulfilment Testing:	67
8.7 Genome Efficiency Testing (#define DIETTEST in Simulation):	67
8.8 Breeding testing (#define BREEDTEST in Simulation(For test 2)):	69
8.9 Visualisation testing:	69
8.10 Simulation Rule Modification Testing:	69
9. Demonstration:	70
9.1 Normal Execution:	70
9.2 Carnivore's Paradise:	70
9.3 Trees instead of shrubs:	70
9.4 Conclusion:	71
10. Conclusion:	72
10.1 Goals:	72
10.1.1 Achieved:	72
10.1.2 Not Achieved:	72
10.2 Alternate Implementations:	73
10.3 Possible extensions:	74
10.4 Experimentation:	75
11. References:	76
Appendix A - User manual:	77
Appendix B: Project Proposal	81
Appendix C: Project Log:	84
Appendix D: Scenarios / Responses Used:	85

1. Introduction:

The purpose of this project is to design and build an alife system, where the behaviour and abilities of a set of simulated creatures are governed by an abstract genome. According to Thomas Ray (1993) artificial life, or alife, is 'the enterprise of understanding biology by constructing biological phenomena out of artificial components' The study of alife is varied and includes hardware methods such as evolutionary robotics (Flozano, Husbands and Nolfi, 2008) as well as software methods such as the program that will be written for this project. Some approaches to evolutionary life are very low level and involve rewriting machine code to allow it to be more flexible, and thus less likely to break when mutating and evolving. (Ray 1993)

This project on the other hand is more high level, similar to the work done by Pichler and Canamero (2007), though it is far simpler. The creatures, their genes and the environment they live in will all be objects in a program. The problem of being unable to handle mutation is dealt with by the fact that the way the creature's genomes can be expressed are both limited, in that they will always be similar enough to be readable by the program and very flexible, and so able to produce a wide variety of different creatures. This diversity will hopefully allow the evolution of many varied approaches to living in the world the creatures find themselves in.

This project is mainly focussed on software development. A brief demonstration of the program running with different parameters is included to show that parameters do have an effect, however the purpose of the project is not so much to do research as to create a tool that could be used by future researchers.

The creatures live in a pseudo randomly generated world and can move about and interact with each other and the world itself in interesting ways. The simulation will be designed in such a way that many of the parameters of the world and creatures will be able to be modified by the user, allowing researchers to gain insights into what patterns of behaviour and what abilities are more useful under different conditions, as well as allowing them to modify the parameters to better fit the goals of their research.

Since the behaviour and abilities of all creatures are determined by the same genome and the markers for ability and behaviour can overlap observations can also be made about if the system values ability or behaviour more in the creatures.

2. Background:

Competition is a fundamental aspect of natural selection. Without competition for limited resources the weak are not weeded out, and evolution cannot take place. But while it is important to realise the effect competition has on ability, with organisms evolving to fit into their niche, it is also important to recognise that competition has an effect on behaviour. Whether it's as simple as cricket chirps (Desutter-Grandcolas and Robillard, 2004) or as complex as bird migration. This project is designed with physical fitness and behaviour tied to genetics, to allow complex interaction between the two and to allow some conclusions to be made over how important each is to survival within the simulation.

The fitness of creatures will be improved using an evolutionary approach. Apart from competition evolution relies on reproduction, diversity and gradual change over time. If organisms do not change over time, or are not different from each other, or do not reproduce they will not evolve. So it is important these conditions are met in the simulation.

Other pieces of software I have looked to for inspiration include the evolutionary GenePool simulation (Ventrella 2003) While both this simulation and the GenePool simulation are based around animats this project is far simpler, not requiring simulated physics or much of the more complex systems.

Another inspiration for my project is a fairly common evolutionary robotics project, that of co-evolved predator prey robots. This is closer to what I want to achieve with the project than the GenePool simulation. The main difference is that I want to avoid explicitly assigning creature roles. I'd much rather the ecosystem develop in such a way that creatures naturally evolve to fill these niches. My project will again be simpler than this.

3. Professional Considerations:

While working on this project it is important that I follow the guidelines laid down in the BCS code of conduct. Specific parts of the code that will be most important to this project are 3e, which states that I must not:

misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others. (BCS Code of Conduct 2011)

I will avoid this by creating a simple user manual for my project that will provide information on the function of the system and allow anyone familiar with computers to use it. Another point that must be borne in mind is section 2f, which is about avoiding 'injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction.' (BCS Code of Good Practice 2004) The main way I could cause damage to someone would be if the program causes some problem with the computer that damages it. During testing I will run some tests that check the program causes no damage to the computer while running. While I will pay particular attention to these parts of it, the whole code of conduct should be considered for each piece of work produced.

In addition to the code of conduct it is important to also examine the BCS code of good practice when working on the project. I will endeavour to stick to the common practices such as efficient workflow management and using appropriate methods and tools.

Since I am managing this project, since I am the only person working on it, I should consider section 3.1 of the code of practice which is about project management, including planning, managing risks, tracking progress and closing a project. Once I start coding I plan to use a publically visible git repository for the version control, and this should allow me to fairly easily track my project progress in a visible way.

Something else from this section that I plan to hold to is not to assume overruns can be recovered later in the project, since I have done so in the past and it has almost led to disaster.

4. Project Plan:

4.1 Introduction:

This section contains the project plan, both in the form of a list of tasks and a PERT chart to explain requirements for each tasks and to examine dependencies.

Proposal Phase (27/9/12 - 9/10/12):

Task	Subtasks	Requires
Find supervisor for project		
Write project proposal	Decide on a working title Write extensive list of possible goals Decide on primary goals Decide on Stretch goals Decide on resource requirements Decide on reading materials	Supervisor

Planning Phase (10/10/12 - 17/10/12):

Task	Subtasks	Requires
Write up list of tasks	Consider necessary phases for development Consider necessary tasks for each phase Consider necessary subtasks for each task	Project proposal
Create PERT chart for project	Assign prerequisites to all tasks Create chart Work out critical path Work out start and end times for each phase Allocate time to each task	List of tasks

Requirements Phase (17/10/12 - 6/11/12):

Task	Subtasks	Requires
Create detailed description of proposed systems in simulation	<p>Write how movement will be handled</p> <p>Write how competition will be handled</p> <p>Write how breeding will be handled</p> <p>Write how combat will be handled</p>	Project proposal
Create CRC cards	<p>Consider design patterns that could be used</p> <p>Create list of candidate classes</p> <p>Consider what information classes will need</p> <p>Consider what the purpose of each class is</p> <p>Consider what other classes each class will need to support it</p>	Detailed description of systems
Create high level UML class diagram	Consider connections between classes	CRC cards
Write use case scenarios		Detailed description of systems
Create use case diagram		Use case scenarios
Write acceptance tests		Project Proposal
Collect all work into requirements document		High level UML diagram, use case diagrams, state diagrams, acceptance tests

Design Phase (7/11/12 – 7/12/12):

Task	Subtasks	Requires
Refine high level class diagrams into low level class diagrams	<p>Consider classes that could be split to improve cohesion</p> <p>Consider links between classes that could be removed to decrease coupling</p> <p>Split classes into loosely coupled modules</p> <p>Consider opportunities to use inheritance and create inheritance diagrams</p>	Requirements document
Create communication diagrams	<p>Consider most important communications in program</p> <p>Work through how the classes will communicate</p>	Detailed class diagrams
Create state diagrams	Consider which are the most important state changes in the system	Detailed class diagrams
Create sequence diagrams	<p>Consider most important sequences in program</p> <p>Work through actions needed during this sequence</p>	Detailed class diagrams
Write up design phase	Document and explain any changes made to design during design process	Detailed class diagrams, object diagrams, communication diagrams, sequence diagrams.

Programming Phase (8/12/12 – 8/3/13):

Task	Subtasks	Requires
Set up git repository to backup work		
Write each module of the program	Write code of each part of the program Debug each part of the program	Design document, git repository
Write up implementation	Write up interaction between parts of the program Write up important data structures for the program and why they were chosen Compare program to design document and comment on any changes	Completed program, design document

Testing Phase (9/3/13 – 16/3/13):

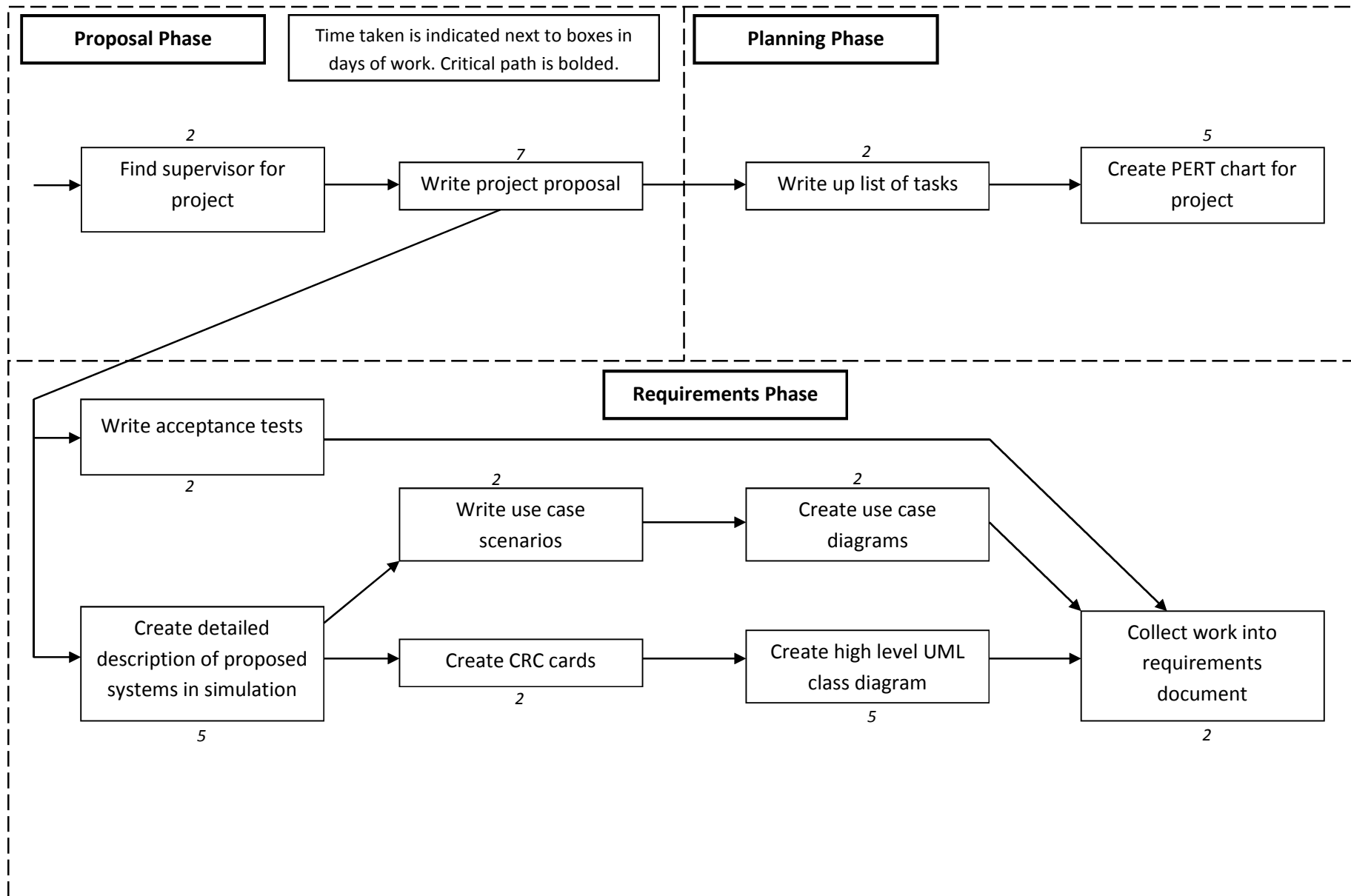
Task	Subtasks	Requires
Perform black box (acceptance) tests	Code tests Perform tests Rewrite parts of the program that fail tests	Completed program
Write up testing	Write up tests, including input, expected output, actual output and actions taken.	Completed white box and black box testing

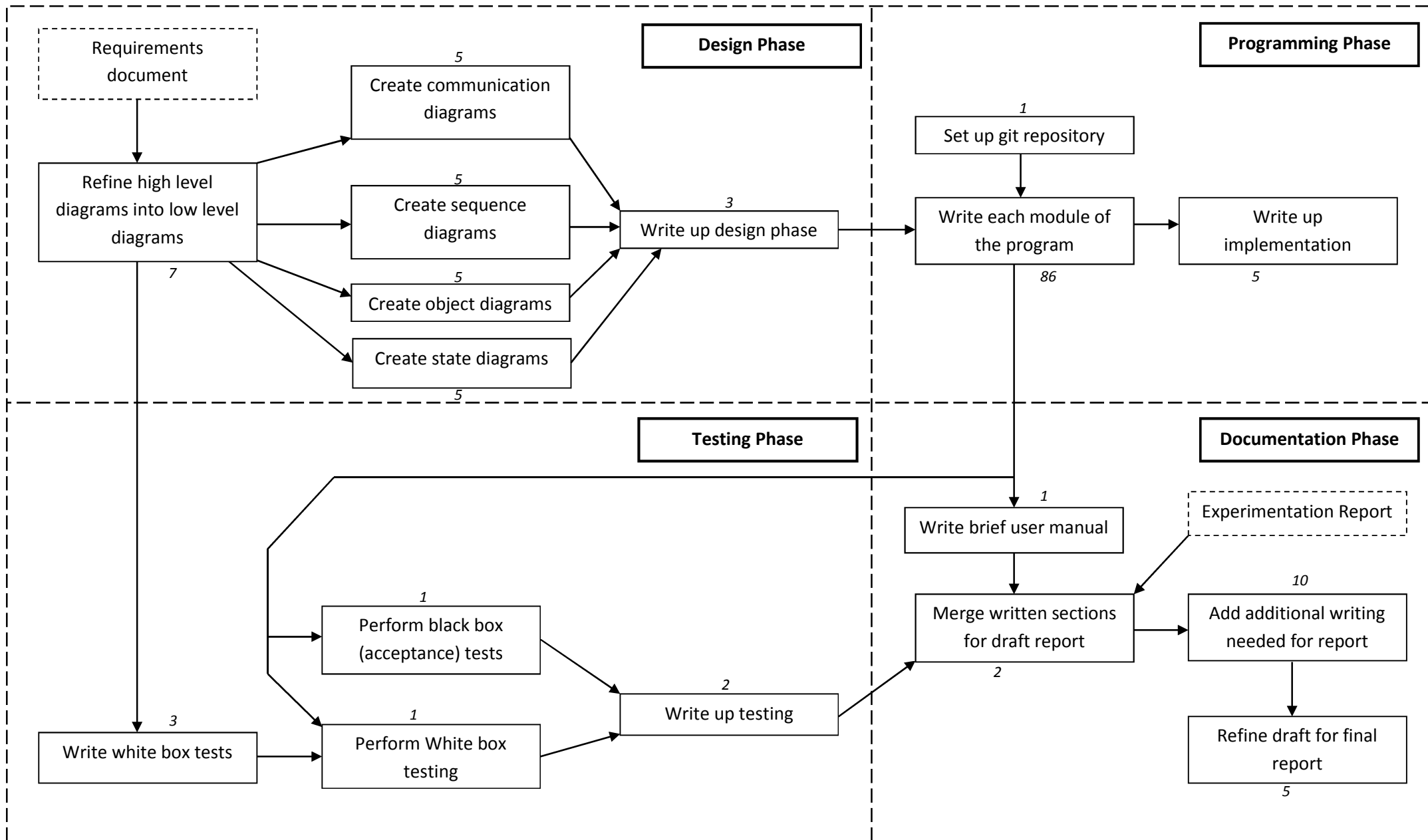
Demonstration Phase (17/3/13 – 20/3/13):

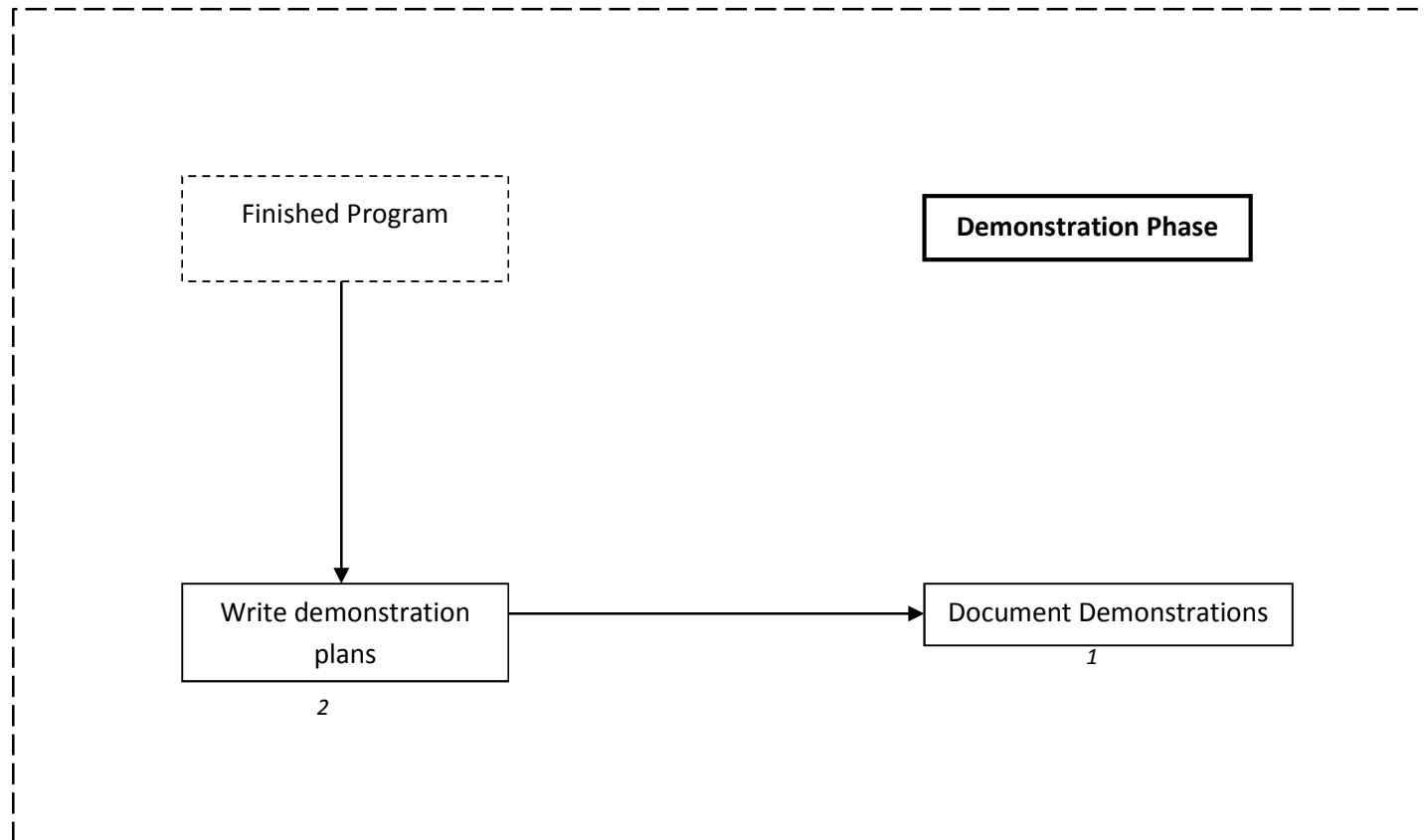
Task	Subtasks	Requires
Write demonstration plans	Consider what parameters will give most interesting results	Completed Program
Document Results	Run demonstrations Write up observations	Written demonstration plans

Documentation Phase (21/3/13 – 11/4/13):

Task	Subtasks	Requires
Write User Manual		Finished program
Merge written sections for draft report		Finished documentation for all phases User Manual
Add additional writing needed for draft report		Merged documents
Refine draft for final report		Finished draft report
Write a brief user manual for the program to be put in the appendices of the report		Completed program







5. Requirements:

5.1 Introduction:

The section deals with the requirements generation for this project. It includes a brief description of the desired behaviour of the system, including some brief examples of things that should be able to occur in the simulation. Following this is the project specification that was generated based on the desired behaviour, after which are a set of CRC cards, then UML class diagrams, inheritance diagrams and use case diagrams and use case scenarios. This section then finishes with a set of acceptance tests.

5.2 Desired Behaviour:

Creatures should move around the world. They should be able to eat plants and remains and be able to attack and kill other creatures to create remains. Creatures should also be able to hide or run away from other creatures. Creatures should not be able to move onto squares containing other objects. Plants should regrow food units and remains should decay until there is nothing left of them. After a number of moves are made the simulation should pause, remove all the old creatures from the world, breed them based on how successful they have been, reset the world and add the children back to it.

A complex genome:

Creatures should base their behaviour and abilities off of a complex genome represented as a set of cells. Each cell has a dominant and a recessive colour. The dominant colour is used for the majority of the actions the gene takes, while the recessive colour is present since the colour used when 2 genes are bred together is random so either can be selected.

Behaviour should be based on a set of Scenarios which map to certain responses, corresponding to actions the creature takes. The majority colours of different sub-sections of the gene should decide which response each creature uses. For example a creature with all of its cell in the top left red might use more “attack” responses than one with all blue cell.

Abilities should be based on pattern matching using small sets of coloured cell, including wildcard colours. Each time one of these sets is matched against a set of cells in the gene the abilities of the creature should be modified.

The use of this complex genome representation allows for a layer of abstraction between the genetics of the creature and its abilities and behaviours. Additionally the gene is both highly expressive, since it has 7^{100} different possible configurations, and very limited, since it will always be some arrangement of the same colours.

Using different methods to determine these different aspects of the creature is being done for two reasons. Firstly there is the hope that the interactions between these systems would produce some interesting effects, as well as causing some instability in the system which would lead to greater diversity. Secondly the different checks require different methods to run them, and since one of my

goals when starting this project was to familiarise myself with C# I see this as a way to gain more experience coding.

Example Sequences:

A creature manages to kill another creature, but when it dies a third carnivore comes out of hiding and kills the now weakened victor. The third creature is then free to enjoy both sets of remains.

Several herbivores cluster around and eat from a single plant, ignoring each other. Suddenly a carnivore charges into the group, some of the herbivores run, some hide, one of them fights and is quickly killed.

2 Creatures see each other and dance around, both too afraid of the other to go near the food in between them. Soon one of them, beginning to starve, changes its mind and runs for the food.

A creature sees some food on the other side of an obstacle. It moves around the obstacle to get to the food.

A creature sees the remains of another creature and begins to move towards them, only for the food to decay to nothingness before it reaches it.

A plant with very little food left on it sits next to one with a lot more food, 2 creatures appear, one moves towards the plant with little food, knowing this plant is more nutritious, the other prefers quantity over quality and goes for the more abundant source of food.

Several plants sit near one another. A creature moves between them and the plants it is not eating from regrow while it feeds off the others.

5.3 Specification:

This specification is loosely based off of the desired behaviour described above. Important concepts are underlined when they first appear and will have sections devoted to them in the specification.

The simulation consists of a number of creatures moving around in a world.

The World:

The world consists of a 1000 x 1000 set of tiles. At the start of the simulation all the objects that will be in the world are added in random clear tiles, including the initial set of creatures.

Updating the world:

Each tick the world updates all creatures, plants and remains within it.

Tiles:

Each tile is a single location in the world and can contain one object before it becomes impassable. Tile contents can be: plants, creatures, remains or obstacles. Each tile also has a colour represented as 3 numbers, one for each R, G and B.

Plants:

Plants contain a number of food units. After a certain number of updates the plant regrows a single piece of food if one is missing. Plants can be eaten from and herbivores get more energy from them. If a plant reaches 0 food units it remains in the world, but is ignored until it regrows.

Remains:

Remains contain a number of food units. Unlike plants they lose a food unit every n updates. Carnivores get more energy from eating remains. If a remains object reaches 0 food units it is removed from the world.

Remains are created at any spot in the world where a creature dies. They are more nutritious than plants by default to allow carnivores more leeway with finding them.

Obstacles:

Obstacles don't take any action or have any variables and just serve to block tiles from being passable.

Creatures:

Creatures are the main focus of the simulation. They can move around the world, eat, and interact with each other. Their behaviour and abilities are decided by an abstract genome.

Genome:

The genetic structure of the creatures is represented as a 10 x 10 grid of cells. Genes can be bred together and use a dominant and recessive system of mating. The genes are used to decide on both the stats of the creature and the behaviour of the creature.

Behaviour:

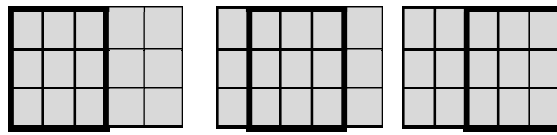
Creature behaviour is represented as a set of scenarios. Scenarios are decided by examining subsections of the genome and checking what colour is in the majority in that section. If there is a draw, the colour that is part of the draw that occurs first when going left to right row by row through the subsection being examined is the one chosen.

To avoid draws becoming very common we will avoid examining a too small subsection of the genome. To maximise the number of scenarios we can examine we will also avoid a too large subsection. This suggests that a 3x3 subsection would be acceptable:

- It has an odd number of cells, and thus is harder to draw in
- It is small enough to allow $8 \times 8 = 64$ overlapping subsections, with maximum overlap

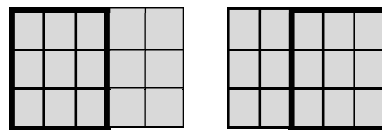
To allow for 64 different scenarios we would move the subsection we are examining over a cell at a time.

Areas to examine



However this level of overlap may stifle the ability of the creatures to adapt to specific scenarios, since any effect on one will more than likely have a large effect on another. The solution to this is to reduce the level of overlap, since that allows different subsections to evolve to have different majorities without it necessarily having too big an effect on the rest of the system.

Areas to examine



The semi overlapping nature of these subsections means that there is some 'bleeding' between scenarios. This effect is desirable since it allows value judgements to be placed on responses to specific scenarios, since we can see if the overlap on some sector is ignored completely to get a specific response to the scenario next to it. It also keeps the amount of scenarios supported to relatively large number, since you now have $4 \times 4 = 16$ different scenarios available.

The following is a list of the different actions the creature can take and what each colour means as it relates to the response to each scenario. The different colours are across the top, the scenarios are down the side and the responses are the response to that scenario if the colour above is the majority.

	1	2	3	4	5	6	7
Food	EAT	EAT	EAT	EAT	EAT_PREF	EAT_PREF	EAT_PREF
Food & Creature	EAT_PREF	EAT_PREF	ATTACK	HIDE	EVADE	ATTACK	EAT
Multiple Food	EAT_PREF	EAT_PREF	EAT_PREF	EAT_PREF	EAT_CLOSE	EAT_CLOSE	EAT_CLOSE
Multiple Creature	ATTACK_CLOSE	ATTACK_WEAK	EVADE	ATTACK_LEAST_DEF	EVADE	ATTACK_FAR	ATTACK_CLOSE
Creature	ATTACK	HIDE	EVADE	AMBUSH	STALK	DEFEND	IGNORE
Multiple food & Multiple creature	ATTACK_CLOSE	HIDE	EVADE	IGNORE	EAT_CLOSE	EAT_PREF	EAT_CLOSE
In combat	ATTACK	ATTACK	DEFEND	DEFEND	EVADE	EVADE	ATTACK
In combat & wounded	ATTACK	ATTACK	DEFEND	DEFEND	EVADE	EVADE	EVADE
In combat & see another creature	EVADE	EVADE	ATTACK_CLOSE	ATTACK_LEAST_DEF	ATTACK_WEAK	EVADE	ATTACK_CLOSE
Starving & food nearby	EAT	EAT	EAT	EAT	EAT_PREF	EAT_PREF	EAT
Starving & multiple food nearby	EAT_CLOSE	EAT_CLOSE	EAT_CLOSE	EAT_PREF	EAT_PREF	EAT_PREF	EAT_CLOSE
Starving & Creature nearby	EVADE	ATTACK	ATTACK	ATTACK	EVADE	HIDE	HIDE
Starving & creature & food nearby	EAT_CLOSE	ATTACK	HIDE	EAT	EAT	AMBUSH	EAT
Starving & multiple creature & food nearby	EAT_CLOSE	EAT_PREF	EAT_PREF	ATTACK_CLOSE	ATTACK_CLOSE	EAT_PREF	EAT_PREF

Responses:

EAT – The creature tries to eat the food source. If it is not next to this food source it will move towards it

EAT_PREF – The creature will eat the closest preferred food item, ignoring any non-preferred food sources.

EAT_CLOSE – The creature will eat the closest food item, regardless of preference.

ATTACK – The creature will initiate combat with the creature it is focussed on, or move towards if it is not adjacent.

ATTACK_CLOSE – The creature will try to attack the closest creature it sees

ATTACK_WEAK – The creature will try to attack the least strong creature it sees

ATTACK_LEAST_DEF – The creature will try to attack the creature with the lowest defence it sees

ATTACK_FAR – The creature will try to attack the least close creature it sees

HIDE – The creature will attempt to be stealthy and hide in place.

EVADE – The creature will try to run away from whatever dangers it sees

AMBUSH – The creature will try to use stealth to sneak up on and attack the nearest creature

STALK – The creature will try to follow the nearest creature stealthily and staying outside its sight range if possible

DEFEND – The creature will defend itself from attack but not attack back

IGNORE – The creature will function as if it has seen nothing and move randomly

Stats:

The abilities of the creatures are decided by stats. These stats are decided by patterns within a creature's genes.

The stats a creature has are:

- Strength – How strong the creature is. Used in combat
- Speed – Governs the order in which creatures take turns each tick, as well as how many tiles a creature can move per turn.
- Awareness – How far a creature can see
- Initial Health – The health the creature starts with, its health cannot increase above this value
- Initial Energy – The initial energy the creature has, energy can be increased above this value
- Stealth Value – How stealthy the creature is, used when hiding, stalking, ambushing
- Defence – Sets how well defended the creature is, used in combat

Patterns:

The patterns to find in the gene are stored as part of a .txt file that is read into the simulation and parsed during initialisation.

Patterns are small sets of cells set to specific configurations. They have a set of modifiers associated with them.

To match the pattern simple translation is used. The pattern is placed at the top left of the gene, and the cells are compared. If all the cells match then the modifiers of the pattern are added to the genome. The pattern is then moved across one and the process is repeated going left to right and row by row. With the multipliers of the pattern being added to the gene each time the shape is found.

Once the whole gene has been checked against one pattern it is checked against the next and the next. Each check for a pattern is independent of any other, and so patterns can overlap with each other and with themselves.

Here is an example pattern. Dashes indicate wildcards, cells in the shape that will match any colour in the gene.

1--

-2-

--3

Gives +STR, +DEF.

Other Stats:

Creatures also have other stats not decided by pattern matching. These are:

Stamina:

Stamina is set as 1/10 of the initial energy value a creature has. Actions which drain energy also drain stamina, and a creature cannot take an action unless it has both enough energy and enough stamina to do so. Stamina, unlike energy regenerates a percentage of itself per tick up to the initial maximum.

To stop creatures being able to immediately clear all the food off of any food source when they find it creatures can only eat when they have maximum stamina and the amount (but not the max) is halved each time they eat.

Diet:

Diet is a number from 1 to 0. When eating plants the nutritional amount is multiplied by 1 – diet. When eating remains the nutritional amount is multiplied by the diet. The diet is modified by the count of different colours in the gene. It starts at 0.5 and for every 1 2 or 3 it is reduced by 0.005, and for every 5, 6 or 7 it is increased by the same amount.

Colour:

The creature's colour is also found using the count of the colours. The colour of the creature is the mean average of all the colours found in the gene, found by averaging all the R G and B values of the colours in the gene.

Stealth:

Creatures that try to hide and are within view of another creature must make a check, and roll an 'exploding' random number (max 10)* higher than the value of (colour difference + (awareness radius - creature distance) - stealth)

While stealthy all energy costs are doubled, however if you attack another creature from stealth they take $\frac{1}{2}$ strength score + stealth value in damage.

Combat:

Combat costs 10 energy for the victor and 20 for the loser.

Attacking another creature means making a check of Attacker's attack + 'exploding 10' vs Defender's defence + 'exploding 10'. Whoever gets a higher number damages their opponent the amount they beat them by.

Once 2 creatures are in combat they have to make a check to escape it of speed + exploding 10 vs attack + exploding 10. If the evader fails they take damage equal to the other creature's attack – their defence. Regardless of the result they are able to move.

Movement:

Movement costs 40 energy / (speed/2) per tile moved. Creatures can move in any cardinal direction (north, south, north-west, etc.). Creatures can move a number of tiles equal to their speed per tick, though if they are wandering randomly they should only move one tile per tick to save energy.

Creature actions per tick:

Each tick the creature performs the following set of steps.

1. Rejuvenates health or stamina if they are less than the maximum
2. Scan the world in a radius determined by the awareness
3. Decide what scenario to use and get the response
4. Take some action

Cells:

Genes are made up of cells. Each cell can be one of 7 colours. The cell is given 2 colours upon creation and decides what colour to present as based on the dominant/recessive system. For simplicities sake the system can be explained like this: Each colour is dominant over the next 3 colours and dominated by the previous 3 colours. This wraps around the sequence so 7 for example is dominated by 4, 5 and 6 and dominant over 1, 2 and 3.

* An exploding random number is a number that is rolled once, and then if the maximum value is gotten is rerolled and added to the previous result. This is done recursively. So with a max of 10 you might get 10 + 10 + 3 = 23 as your value if the first 2 numbers you roll are 10s.

Breeding:

When breeding the genome takes a random part of each cell and mixes it with the random part of the same cell in another genome to get the new cell for the new genome. Child cells have the same location as parent cells.

For example (parts of each genome used are bolded):

11	23		11	42		11	22
33	34	+	11	22	→	31	32

Each part of the genome has a 0.2% chance to mutate when copied and become a random number from 1 to 7.

The colour chosen from the cell is the dominant colour 60% of the time and the non-dominant colour the other 40% of the time.

Judging:

Judging is done at the end of each generation, which occurs after a set number of updates. At the end of the generation all the living and dead creatures are gathered and judged.

Creatures are judged based on the level of health and energy they have at the end of the round, or, if they are dead, based on the order they died in. When two creatures have the same health and energy their stats will be compared to find which is better.

Typically the rating list should look like:

- Creatures with very high energy and health
- Creatures with less high energy and health
- Creatures that died near the end of the generation
- Creatures that died earlier

Once this list is created and sorted the bottom 25% is eliminated. The top 25% are bred together to create half of the next generation and the remaining creatures are bred together to make the rest of the generation.

After enough children are created to populate the next generation the world is reset, so all remains are cleared out and the plants are all returned to max food units, and the old creatures are all removed from the world. After this the children are placed randomly in the world.

5.4 User Interaction:

The ways the user can interact with the system are divided into several key areas:

Observation:

The user will be able to see the current state of the simulation, both a broad overview of the world, showing the locations of different objects and a more in depth look at specific objects in the world when they select them.

Modification:

The user will be able to modify parameters of the simulation so that they can set up experiments and view how different conditions change the evolution of the creatures. This modification should be done through an easy to use menu that uses text boxes, radio buttons, sliders and so forth.

Examples of variables that could be changed by the user are nutritional values of plant and remains, energy drain each tick, how heavily the simulation values various stats when judging creatures and so on.

Intervention:

The user should be able to take some actions directly in the world, for example the user should be able to clone or kill creatures they are viewing, or force plants to instantly regrow or remains to instantly decay. This should be available at least on an object by object basis, however some ability to affect ALL objects in the world simultaneously might also be nice.

The user should also be able to clone and destroy creates if they so choose, whether to attempt to create a certain change in the next generation or to examine the effects of severely reduced diversity on the evolution of creatures.

The user should also be able to affect the speed of the simulation, allowing them to view more interactions in smaller amounts of time.

If time permits the genome editor would be another way for the user to interact with the system, by using a simple, 'paint' like interface to create custom genes, which can then be imported directly into the world.

5.5 CRC Cards:

Design Patterns:

After considering the various systems and potential classes that might be necessary for the simulation to function I decided on several design patterns that I would use:

- Façade Pattern: The Façade pattern will be used to reduce coupling where possible. By making one class route all calls to classes under it I can reduce the need to modify the 'hidden' classes when modifying classes that use them.
- State pattern: The state pattern will be used in the display to allow multiple states without the use of long conditionals or switch statements, allowing easier upkeep and greater extensibility.
- Singleton pattern: Simulation will use the singleton pattern to avoid the errors that could occur if 2 different sets of parameters exist.

The decision to use these patterns is reflected in the CRC cards.

Cards:

Class: World	
Responsibilities: Function as a façade for the internal mechanisms of the world Update all the creatures each tick Store the state of the world Degrade remains and regrow plants Keep track of round lengths Rank performances of creatures Cull the weak and breed the strong creatures	Collaborators: Tile Creature Simulation Plant Remains

Class: Tile	
Responsibilities: Keep track of what is inside itself	Collaborators: Obstacle Plant Remains Creature

Class: Obstacle	
Responsibilities: Exist in a tile	Collaborators: None

Class: Plant	
Responsibilities: Be eaten Regrow Keep track of how much food is currently in the plant	Collaborators: None

Class: Remains	
Responsibilities: Be eaten Rot away Keep track of how much food remains on the body	Collaborators: None

Class: Creature	
Responsibilities: Act dependent on genome each tick Keep track of stats Keep track of location in world Don't move into tiles that should be impassable Store genome Breed with other creatures	Collaborators: Genome Creatures

Class: Genome	
Responsibilities: Keep track of what colour each cell is Provide methods to breed with other genomes Perform pattern matching to give creature stats Store list of shapes and what they mean for the creature	Collaborators: Creature (needs to be able to give it stats) Genome Shape

Class: Cell	
Responsibilities: Keep track of what colour it should appear as Keep track of both parts of the genome found in this cell (dominant and recessive) Provide parts of the cell randomly to the genome when needed for breeding.	Collaborators:

Class: Shape	
Responsibilities: Exist to allow looking up of different shapes by genome Store what effects it has on the creature if found	Collaborators: Cell

Class: GenomeEditor	
Responsibilities: Allow easy creation of genomes Allow saving and loading of genomes Allow adding of genomes to the world	Collaborators: Genome Simulation

Class: Simulation	
Responsibilities: Store user set parameters Keep track of which creature is currently being viewed/followed Keep track of generation number and target generation number Keep track of the state of the program Keep track of speed of the program	Collaborators: World Creature SimulationState Display

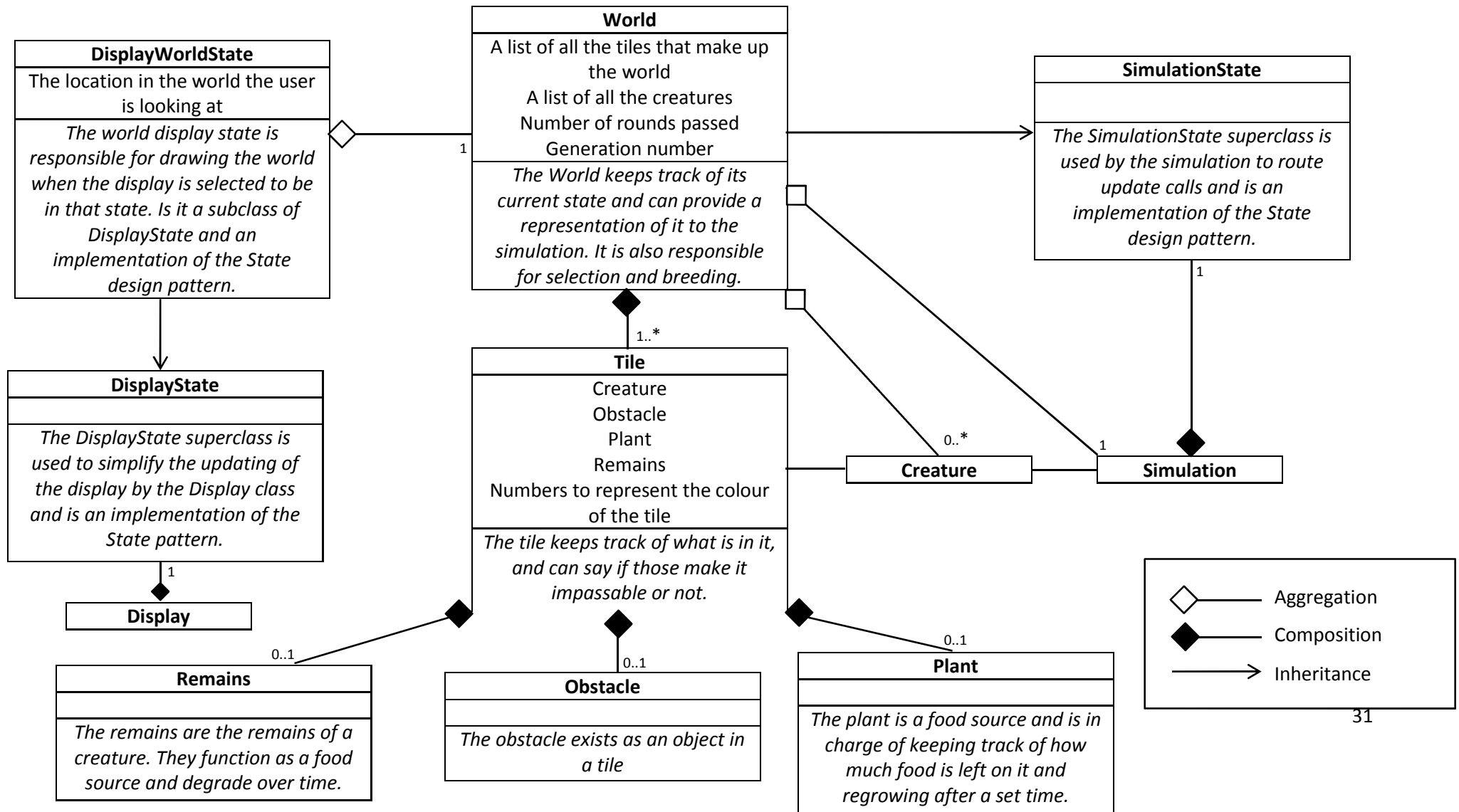
Class: Display	
Responsibilities: Store user set cosmetic options Tell the DisplayState to update	Collaborators: DisplayState
Class: DisplayState	
Responsibilities: Function as a superclass for the states the display can be in. Provide suitable abstract methods to be overridden by subclasses.	Collaborators: Display
Class: DisplayWorldState	
Responsibilities: Display the world Display viewing areas to allow more detailed viewing of creatures and genomes Allow user input and control of the simulation	Collaborators: World Display
Class: DisplayGenomeEditorState	
Responsibilities: Display the genome editor Take user input for the genome editor	Collaborators: GenomeEditor Display
Class: DisplayMainMenuState	
Responsibilities: Display the main menu Allow the user to select what option they wish to access on the main menu.	Collaborators: MainMenu Display
Class: MainMenu	
Responsibilities: Keeps track of what option the user has selected on the main menu, reacts to input.	Collaborators: Simulation
Class: SimulationState	
Responsibilities: Keep track of the state the simulation is in and make sure the correct modules are updating. An implementation of the State pattern, with this being the superclass and World, GenomeEditor etc. being subclasses.	Collaborators: Simulation

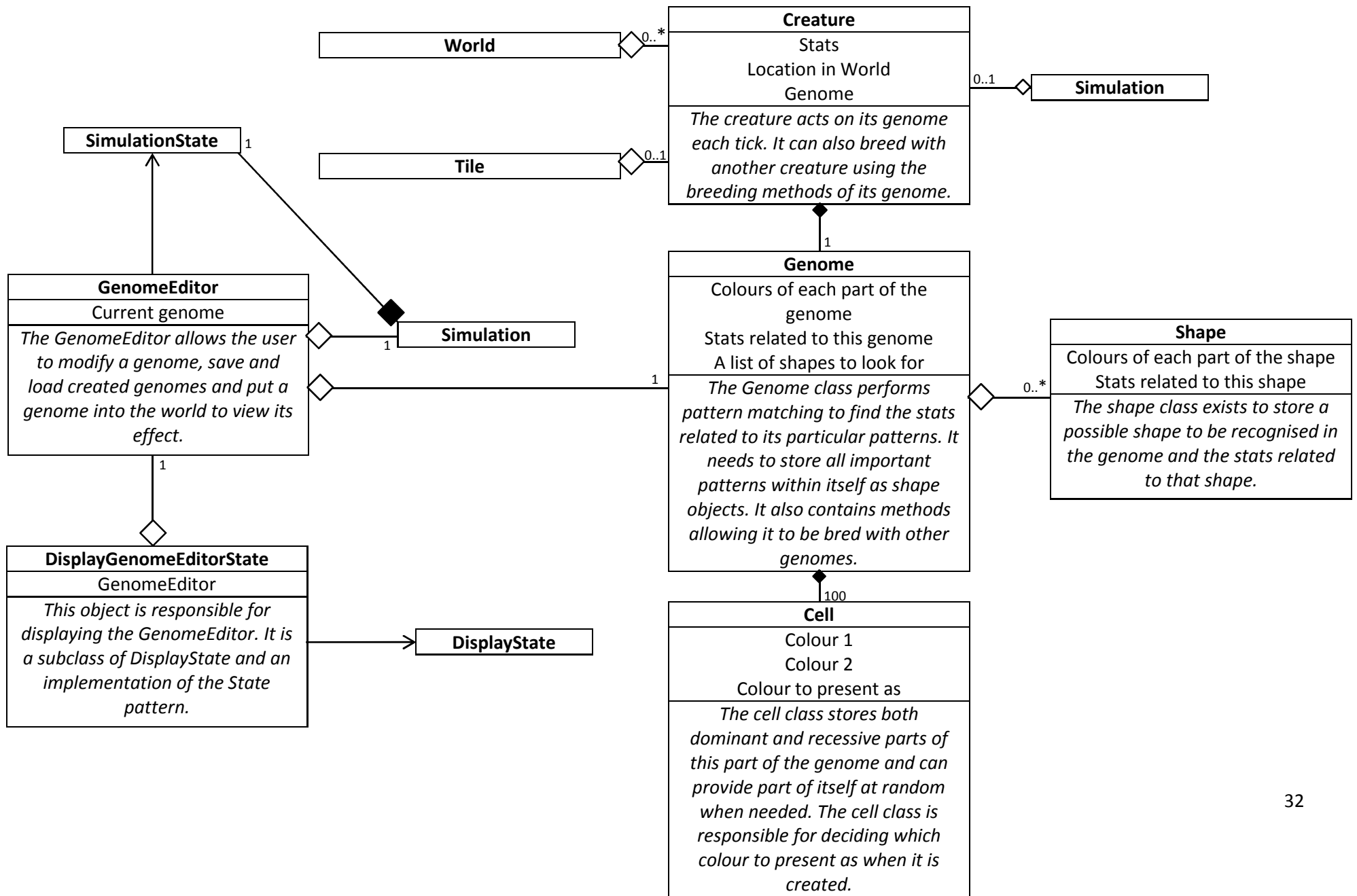
Class: OptionsMenu	
Responsibilities: Allow editing of program parameters, both technical and cosmetic.	Collaborators: Simulation Display

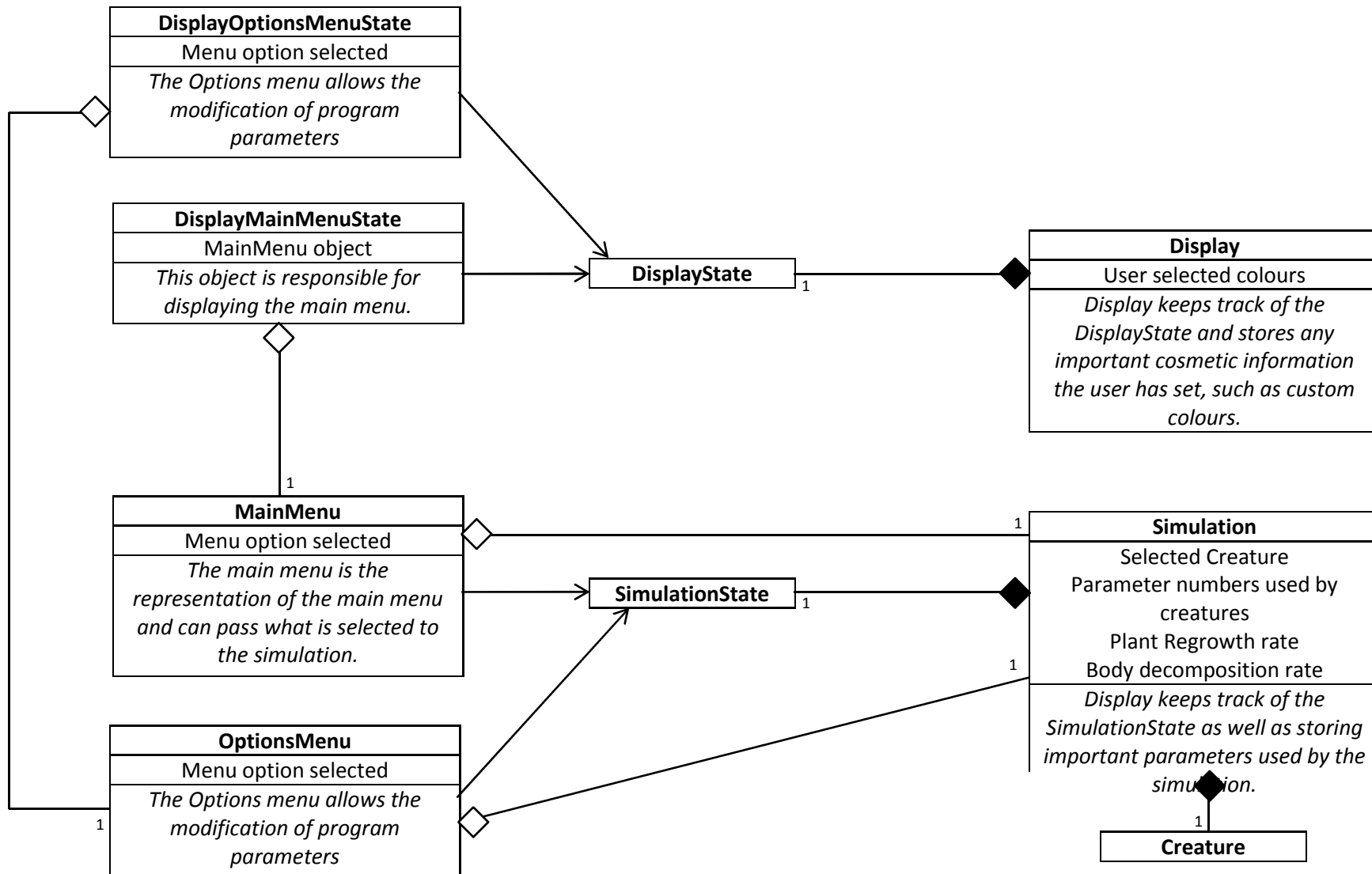
Class: DisplayOptionsMenuState	
Responsibilities: Display the options menu	Collaborators: Display

5.6 UML Class Diagrams:

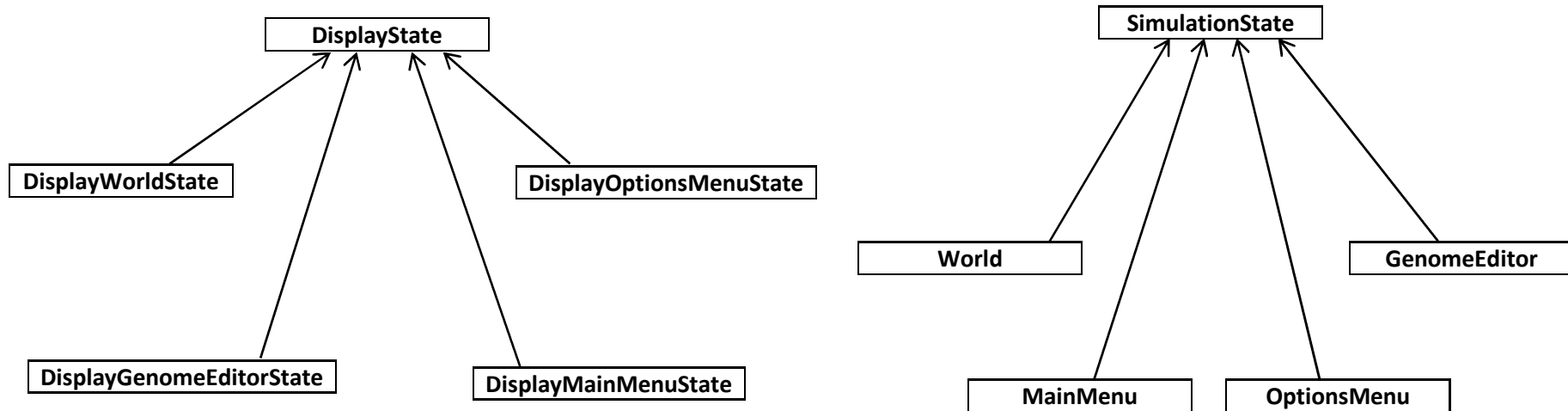
These are high level UML class diagrams that were created based on the CRC cards. Each class has the class name, important data the class will need and some of the responsibilities of the class. These will be expanded on in later class diagrams. Note that not all links are included on each page due to size constraints however I have indicated on any classes with additional links using a small box containing just the class name. Unless specified otherwise assume all numbers on the diamond side of a relationship are 1. These have been omitted to allow more compact diagrams.







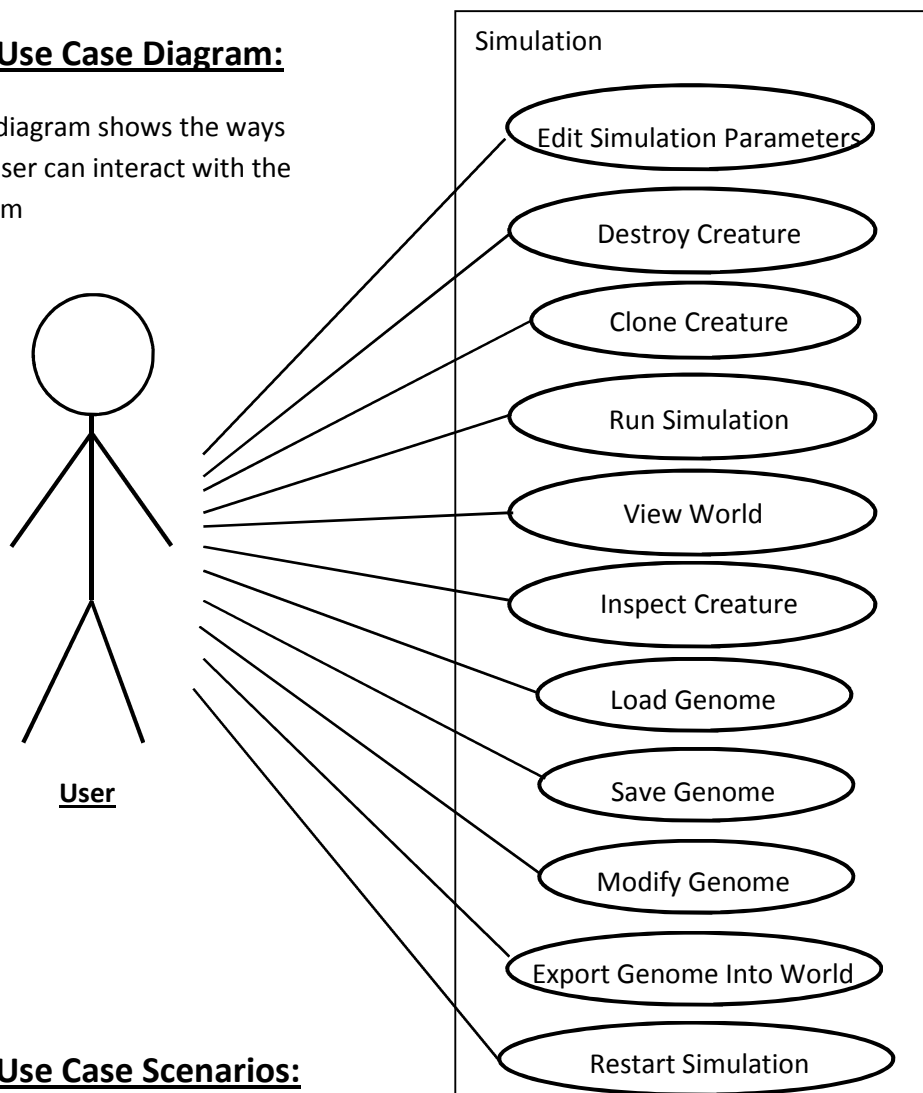
5.7 Inheritance Diagram:



These diagrams show the implementation of the State design pattern within the system. The superclasses will be abstract and will have an update or other such method that must be overridden. The Simulation and Display classes can then simply call the abstract method while having one of the subclasses as the active state in their code and the correct section will update automatically.

5.8 Use Case Diagram:

This diagram shows the ways the user can interact with the system



5.9 Use Case Scenarios:

Edit Simulation Parameters

The user navigates to the options menu where they are allowed to change a variety of simulation parameters and cosmetic details about the program. They are then returned to the simulation. The simulation is able to use the new parameters without restarting or causing errors.

Destroy Creature

The user is at the creature inspection window. They choose the option to destroy the creature. The creature is destroyed. No replacement creature is generated in the world until the next round of breeding. The simulation treats the creature as if it died naturally.

Clone Creature

The user is at the creature inspection window. They choose the option to clone the creature. An identical creature is spawned at a random location in the world. After a confirmation the user switches to be inspecting the new creature

Clone Creature Alt 1: The user chooses not to inspect the new creature. The creature that was cloned remains selected

Load Genome

The user is at the genome editing window. They decide to load a previously saved genome. The genome is loaded into the editor and once loaded it can be modified.

Save Genome

The user is at the genome editing window. They decide to save the genome currently being worked on. A location to save the genome is selected and it is saved.

Modify Genome

The user is in the genome editing window. They are able to put their choice of colour into various cells in a genome. The stats associated with that genome are displayed and updated when the user directs them to be.

Export Genome Into World

The user is in the genome editing window. They choose to export the current genome into the world. A new creature is created in the world with the provided genome. The program state is switched to viewing the world with the new creature centred in the viewing area.

Run Simulation

The user runs the simulation. The simulation breeds together creatures indefinitely using any custom parameters the user provided. The simulation runs at a user specified speed.

View World

The user is at the world viewing area. The user can move the viewing area around the world to get a better idea of what the world looks like and how different creatures are faring.

Inspect Creature

The user is at the world viewing area. The user selects a creature in the world. While the creature is selected the viewing area is centred on it, no matter where it moves. The user is presented with a view of the creatures stats and genome, as well as options to clone or destroy the creature.

Restart Simulation

The user chooses to restart the simulation. The simulation generates a new world and a new assortment of random creatures and begins running from generation 1.

5.10 Acceptance Tests:

These tests are based on the initial requirements detailed in the project proposal. Each requirement is assigned a series of tests to fully check that it is met. The tables indicate how each test will be carried out, the input that will be given to the test and the expected output. The tests will be carried out on several different machines, all will be running windows 7 64bit and the program will not be tested on other operating systems.

Requirement: Implement creatures having abilities that are affected by the patterns found in their associated genome

Test	Input	Expected Result
A simple genome will be tested using a single search shape to check that any shapes are recognised	A genome containing 3 instances of the search shape A shape definition to allow the shape to be recognised	The stats returned by the genome should reflect that there are 3 occurrences of the shape.
A single genome will be tested using 2 overlapping search shapes to check that overlapping shapes does not affect the patterns recognised	A genome containing 3 instances of one shape and 2 instances of the other. The instances should overlap in one case and not in the other. A shape definition to allow both shapes to be recognised	The stats returned by the genome should represent the number of occurrences of each shape regardless of overlap.
A single genome will be tested using 2 different colour shapes to check that having multiple colours functions correctly	A genome containing 2 instances of shapes of each colour side by side A shape definition to allow both shapes to be recognised	The stats returned by the genome should show 2 of each shape recognised
A single cell sized shape will be tested at the very bottom right of the genome to check that the whole thing is being scanned	A genome consisting of a single coloured cell in the bottom right of the genome A shape definition to allow the single cell to be recognised.	The stats returned by the genome should show the single cell as being recognised

Requirement: Implement creatures having behaviours that are affected by viewing sectors of their associated genome

Test	Input	Expected Result
Several creatures will be created with genomes of solely 1 colour and the responses they have stored compared to the responses expected	Several creatures with genomes of all 1 colour	The test creatures will exhibit the desired behaviour

Requirement: Implement creatures having basic needs and the drive to fulfil these needs

Test	Input	Expected Result
The simulation will be run and the actions of the creatures examined	The completed program	The creatures will attempt to eat the food in the world

Requirement: Implement genome affecting how and how well creatures can fulfil certain needs

Test	Input	Expected Result
2 Creatures, 1 carnivore and 1 herbivore will be created and manually given a list containing plants and remains to see which they will select as their preferred	2 Creatures, 1 carnivore and 1 herbivore, a list containing at least 1 plant and at least 1 remains	Each creature will choose its preferred food type
2 Creatures, 1 carnivore and 1 herbivore will be created and fed a plant and a remains each, and how much energy they get from each piece of food will be examined.	2 Creatures, 1 carnivore and 1 herbivore, a plant and a remains	The creatures should regain more energy from their preferred food source

Requirement: Implement a dominant and recessive system of mating

Test	Input	Expected Result
A genome cell will be created with the same primary and secondary values and the value the genome presents as will be compared to the value it should present as	A single genome cell with 2 of the same colours stored in it	The genome will represent itself as the colour that is stored in it.
A genome cell will be created with 2 different colour values and the colour the genome presents itself as will be compared to the colour it should present as	A single genome cell with 2 different colours stored in it	The genome will represent itself as the dominant colour

Requirement: Implement worlds with obstacles and food sources

Test	Input	Expected Result
A world will be generated randomly and inspected visually to confirm the presence of food sources and obstacles	A randomly generated world	The world will contain food sources and obstacles

Requirement: Implement round based breeding

Test	Input	Expected Result
The simulation will be run for the specified number of ticks and then the test will observe if the simulation changes state to begin breeding the new generation of creatures.	A working simulation	The simulation will breed the new generation after a set number of ticks

Requirement: Implement visualisation of the worlds, creatures and genomes

Test	Input	Expected Result
The tester will generate a random world, and use the debugging tool to compare the software viewer with the underlying model.	A randomly generated world	The visualisation of the world will match the model

Requirement: Implement simple modification of simulation rules

Test	Input	Expected Result
The tester will modify an easily visible rule and manually trigger some actions or calculations that use that rule	A working simulation	The actions should have different results with different rules.

6. Design Document:

6.1 Introduction:

This section deals with the more in depth design work based on the high level designs and the specification produced in the requirements section (Section 5). This section starts with an explanation of the language and framework I chose to use. After this there is a diagram showing the updated plan for the structure of the program. This is followed by a list of changes made from the earlier design and the reasoning behind these changes. Next, the more important methods of the program are discussed. The GUI design finishes this section.

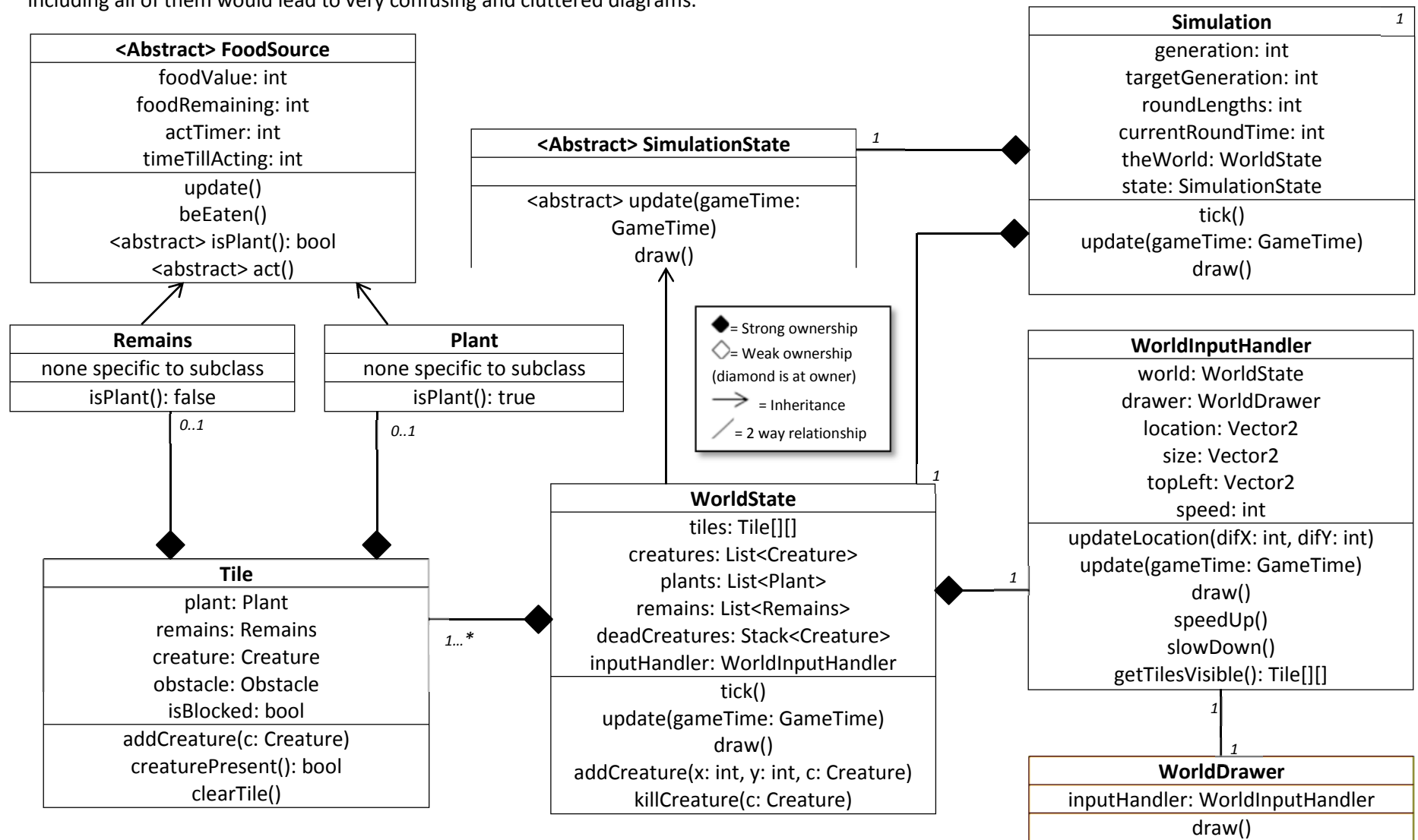
Deciding on a language:

I will implement this project in C# and Xna. I will use C# for a couple of reasons. Firstly it is a language I am fairly unfamiliar with, having done one small project in it before now. I hope to use the programming I will have to do for this project to better learn this language, and become more familiar with it. Another reason I have chosen C# is its similarity to java, which I have used quite often, and which will give me a good base to build from.

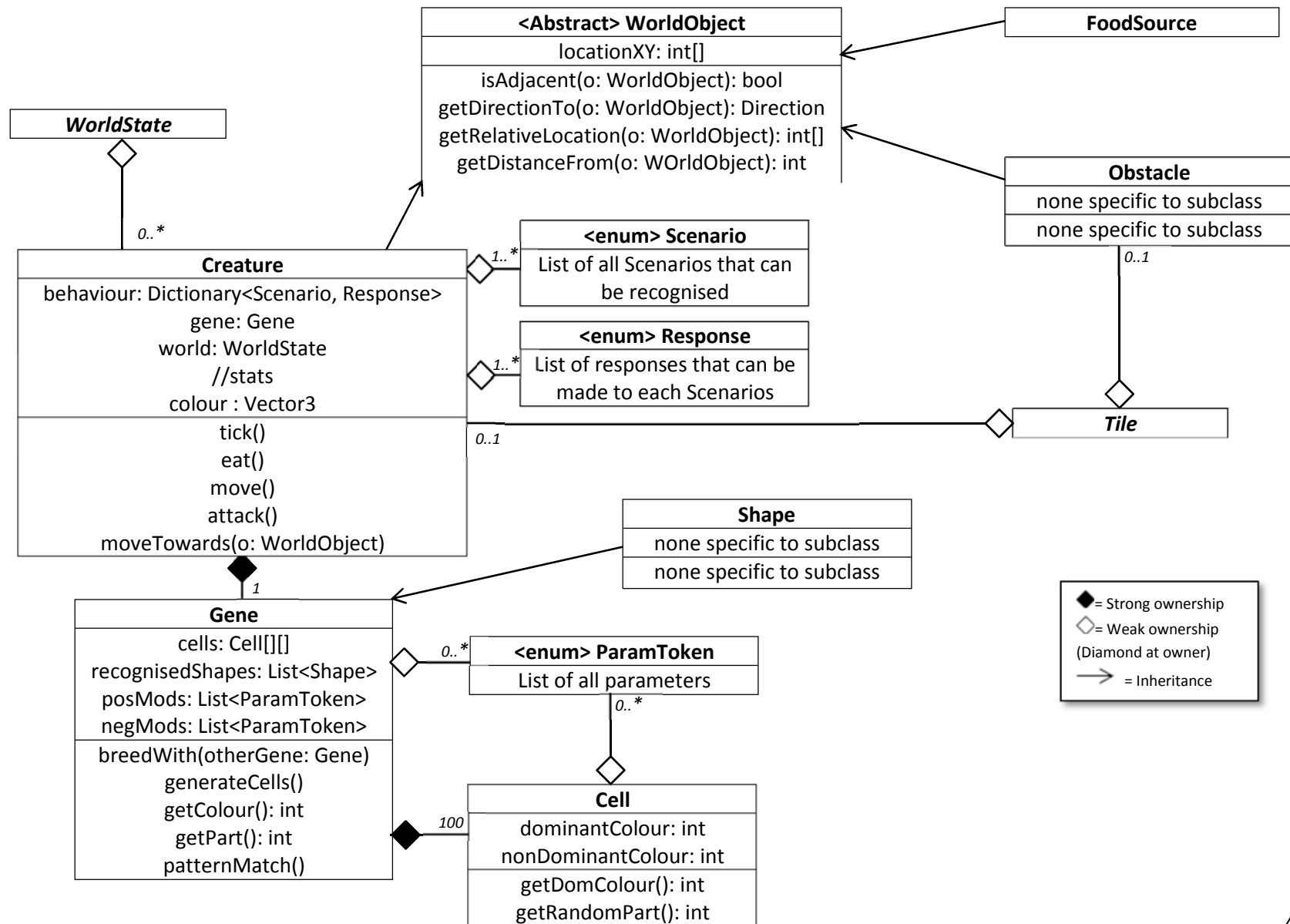
I will use Xna since I have used it in my previous C# project and was impressed with how easy it was to write front-end code using it. One of the other motivating factors behind my use of Xna is the fact that while it has ways of taking mouse and keyboard input it has no button or text field classes built in, and having to code these from scratch is something I think I should have some idea of how to do, even if I will probably avoid it in the future.

6.2 Updated Class Diagram:

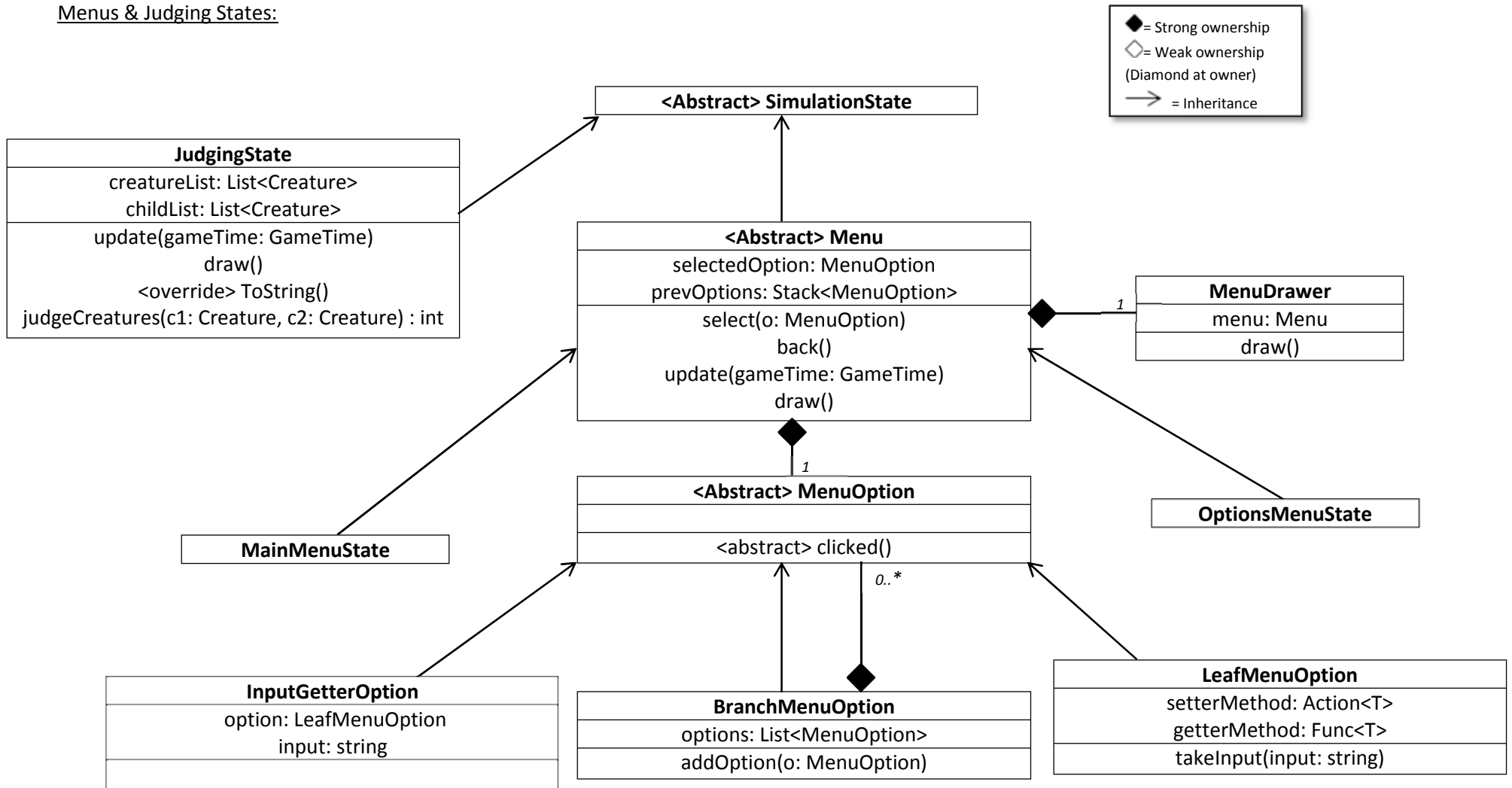
The following diagrams show the class structure the program will have. The explanations for each change made between the diagram in the requirements document and this can be found in the next subsection. Note that each class only includes its most important planned methods and variables since including all of them would lead to very confusing and cluttered diagrams.



World Objects:



Menus & Judging States:

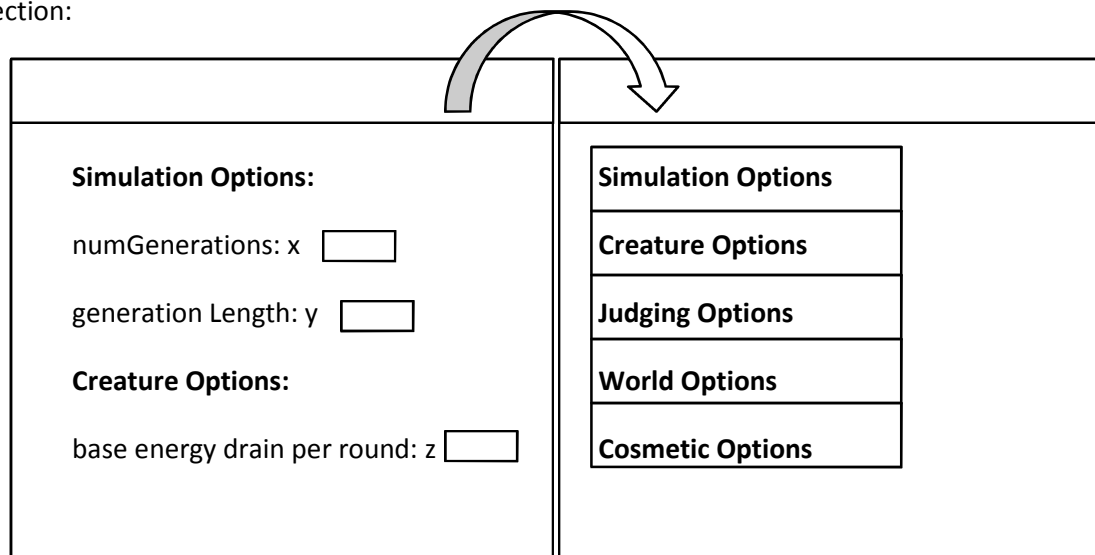


6.3 Changes:

Number	Classes	Change
1	OptionsMenu, MainMenu	Made them subclasses of the abstract Menu class
2	Menus	Made menu structure based off trees instead of lists
3	Display	Removed class entirely
4	DisplayState	Removed class entirely
5	Remains, Plant	Made subclasses of FoodSource abstract class
6	Remains, Obstacle, Plant, Creature	Made subclasses of WorldObject abstract class
7	Genome	Renamed to Gene
8	Shape	Made subclass of Gene
9	WorldState	Added InputHandler to deal with user input
10	WorldState	Removed direct relationship with front end class
11	WorldState	Removed responsibility to keep track of round & generation number
12	WorldState	Removed responsibility to keep track of speed

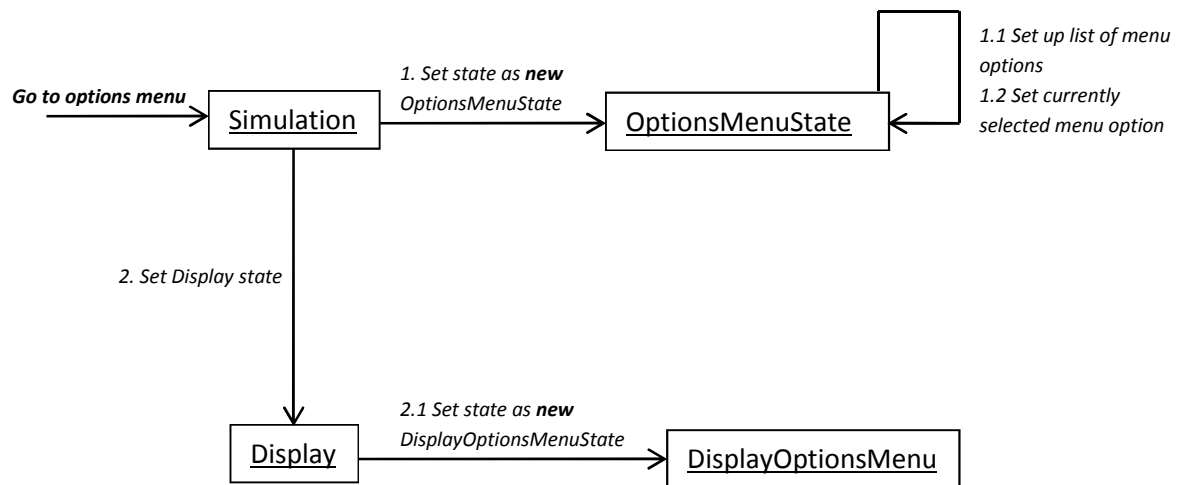
Reasons:

1. This change was done to better take advantage of inheritance and cut down on code reuse. By making the specific implementations of the menus simply implement the Menu class with a specific set of MenuOptions the creation of the different menus is made much easier.
2. When creating the GUI I found that the way I initially visualised the menus, as a single list with a series of headings to separate different types of option, was not very user friendly. This held true especially in the case of there being large numbers of parameters that could be changed, which was something I am hoping to implement. To solve this I changed the GUI design in order to add category selection:

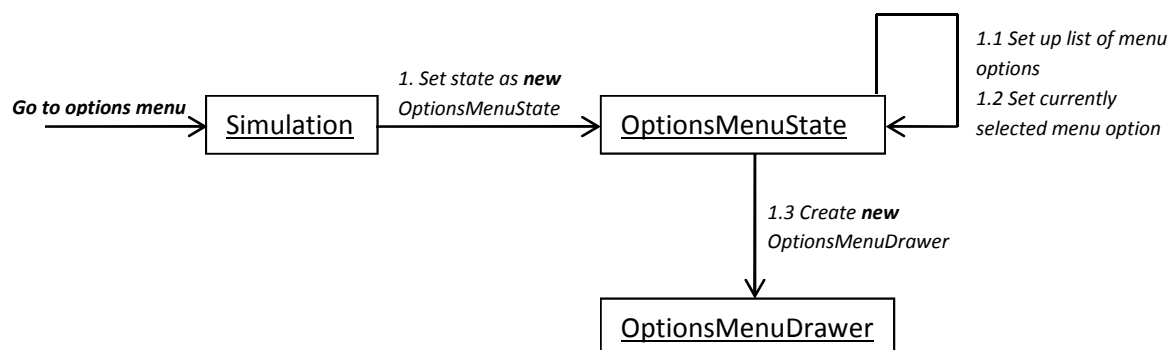


After this I realised I might also want sub-categories. Thus I decided to implement the menus as a tree structure instead of a list structure, requiring a change in the classes the simulation used.

3. In the requirements document the diagrams imply that the Display would store the current DisplayState class in charge of drawing the current SimulationState. This way of storing the different drawing states made changing states look like this:



This is a problem since it requires two different classes to be notified that the state has changed. This would make it very easy for errors to pop up if the SimulationState is changed without updating the Displaystate. Also since the DisplayStates will still need access to parts of the SimulationState anyway in order to know what to draw, this means the state will be stored in multiple places, which is redundant and could end up with the DisplayState trying to draw an old state if it is not correctly updated at all times. Both of these problems could easily cause run time. To solve this I removed the Display class, and made display (or drawing) classes part of the simulation states. Since not all SimulationStates need a drawing class I have not made it part of the superclass, instead it is up to each subclass to include or not include its own drawing state. This modification makes the state changes look more like this:



This is far more streamlined.

4. Once the Display class was removed and the various drawing classes tied more closely to the SimulationStates it seems there will no longer be any need for the superclass in order to exploit polymorphism, and no shared code since the different drawing classes will do very different things.

5. Since both types of food perform the same action, that is, both count down and then do something, and since both types of food need to have methods to handle being eaten, it makes sense to make both types subclasses.

6. All the objects in the world need to know where they are, since creature will need to know where they are in order to move towards them/away from them etc. Using inheritance here allows us to ensure that all the objects in the world can store this information, even if some of them choose not to.

7. This change is trivial.

8. Since shapes are just small Genes that can be matched against the larger ones, making them subclasses of gene makes sense.

9. Having the World class deal with the user input reduced class coherence, since it would be both in charge of being the model of the World and in charge of dealing with input, which are not very closely related functions. For this reason I have added the WorldInputHandler class.

10. Now that I have the InputHandler class I can make the drawing class get what to draw from that, creating a layer of abstraction between the model and the view, and turning these three classes into implementations of the Model-View-Controller design pattern. This is good since the drawer has no need to access the full model of the world, it only needs to know what to draw.

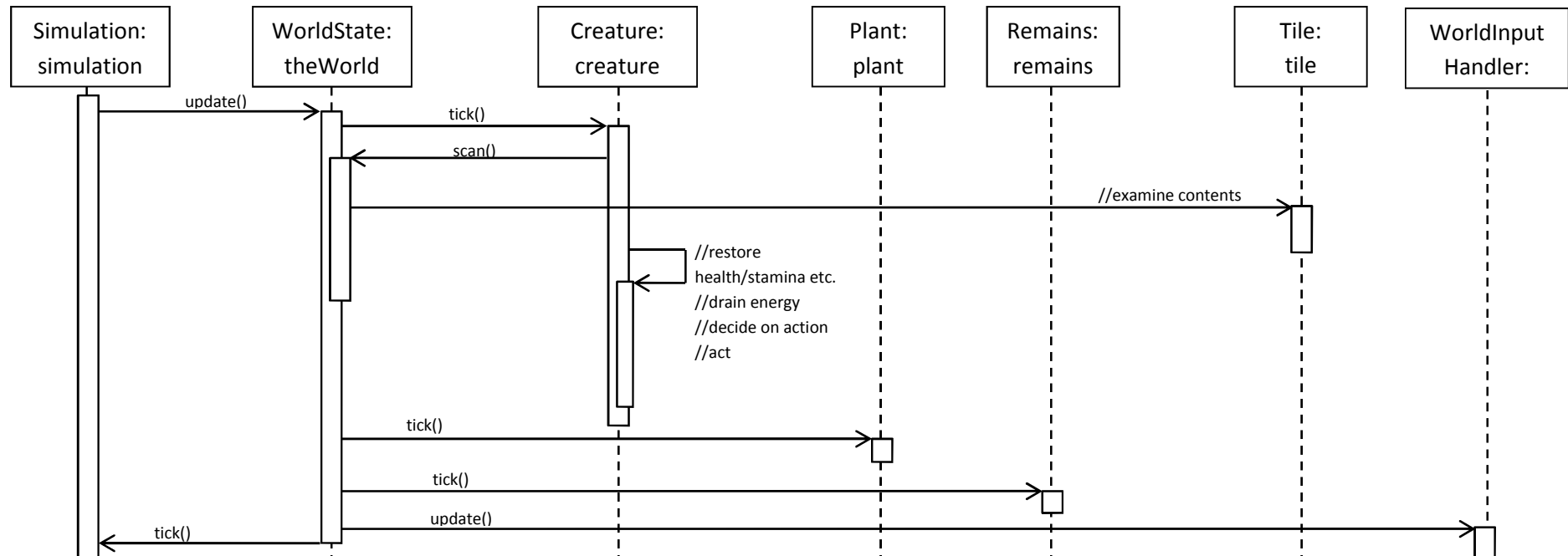
11. Removing the responsibility to keep track of round and generation number from world also makes the class more coherent. These tasks are now the job of the Simulation class, since they are related to the game rules, and require state changes when they reach set values.

12. Removing the responsibility to keep track of speed from the World does the same thing. The WorldInputHandler is a better place to store this since it is something that is related to user input, rather than the current state of the world.

6.4 Important Methods:

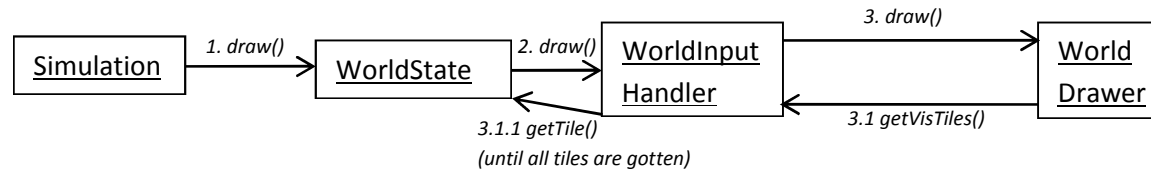
WorldState – update(GameTime gameTime):

Updating the world means updating all the creatures plants and remains in it, and then updating the WorldInputHandler.



WorldState – draw():

Drawing the world means telling the WorldInputHandler to draw the world, which tells the Drawer to draw the world and gives it the tiles that it needs to know about.



JudgingState – judgeCreatures(Creature c1, Creature c2):

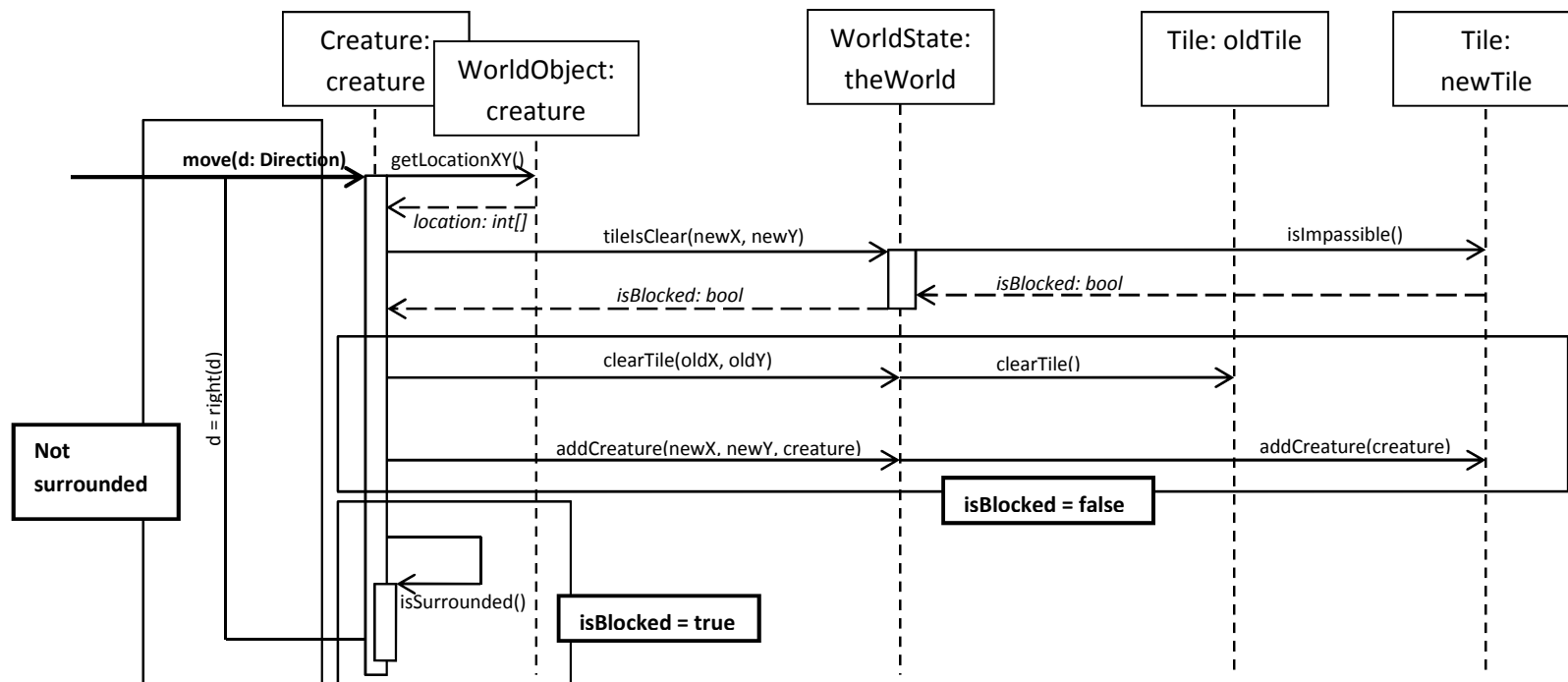
The judgeCreatures method is the fitness function applied to two creatures in order to find the better of the two. The function used is: $f = xH + yE$ where H is the creature's health, E is the creature's energy and x & y are constants that are specified by the user. The creature that has the higher value result from the fitness function is better. If $f(c1) = f(c2)$ then an additional function $g = strength + speed + awareness + defence + maxHealth + initial\ energy + stealth\ value$ is used to further differentiate between the creatures. If $g(c1) = g(c2)$ then the creatures are sorted in an arbitrary order.

Menu – select(MenuOption o):

Selecting a MenuOption pushes the currently selected MenuOption onto the stack of previous options and sets o as the selected menu option. This means that the back button can go back through the options in reverse order, instead of just being able to go back to the last selected option.

Creature – move(Direction dir):

When the creature moves it tells the world to remove it from the tile it is currently in and add it to the new tile it should be in. The direction is calculated with positive y being south and positive x being east.



If the creature cannot move in the selected direction it checks if it is surrounded, if it is not then it tries to move in the direction one to the right of the one that was originally provided to it.

Creature moveTowards(WorldObject o):

When the creature needs to move towards something it uses the `getDirectionTo` method to get the direction it needs to move towards something and then uses the `move` method with the direction that this method returns. This method is called whenever a creature tries to act on something it is not adjacent to, so that it will move towards the object in question and hopefully be in range to act on it next tick.

WorldObject – getDirectionTo(WorldObject o):

This method gets the cardinal direction that the other object o is from the object the method is called on. It works like this:

1. Get the relative location of the object o (x, y)
2. Divide both x and y by the maximum value out of x and y.
3. Use these tables to find the direction:

	y >= 0.5	y <= -0.5	any other y
x = 1	NORTHEAST	SOUTHEAST	EAST
x = -1	NORTHWEST	SOUTHWEST	WEST

	x >= 0.5	x <= -0.5	any other x
y = 1	SOUTHEAST	SOUTHWEST	SOUTH
y = -1	NORTHEAST	NORTHWEST	NORTH

WorldObject – adjacentTo(WorldObject o):

This method checks if the WorldObject o is adjacent to this WorldObject. Something is adjacent to this object with if its relative location is (x, y) and $|x| < 1$ and $|y| < 1$.

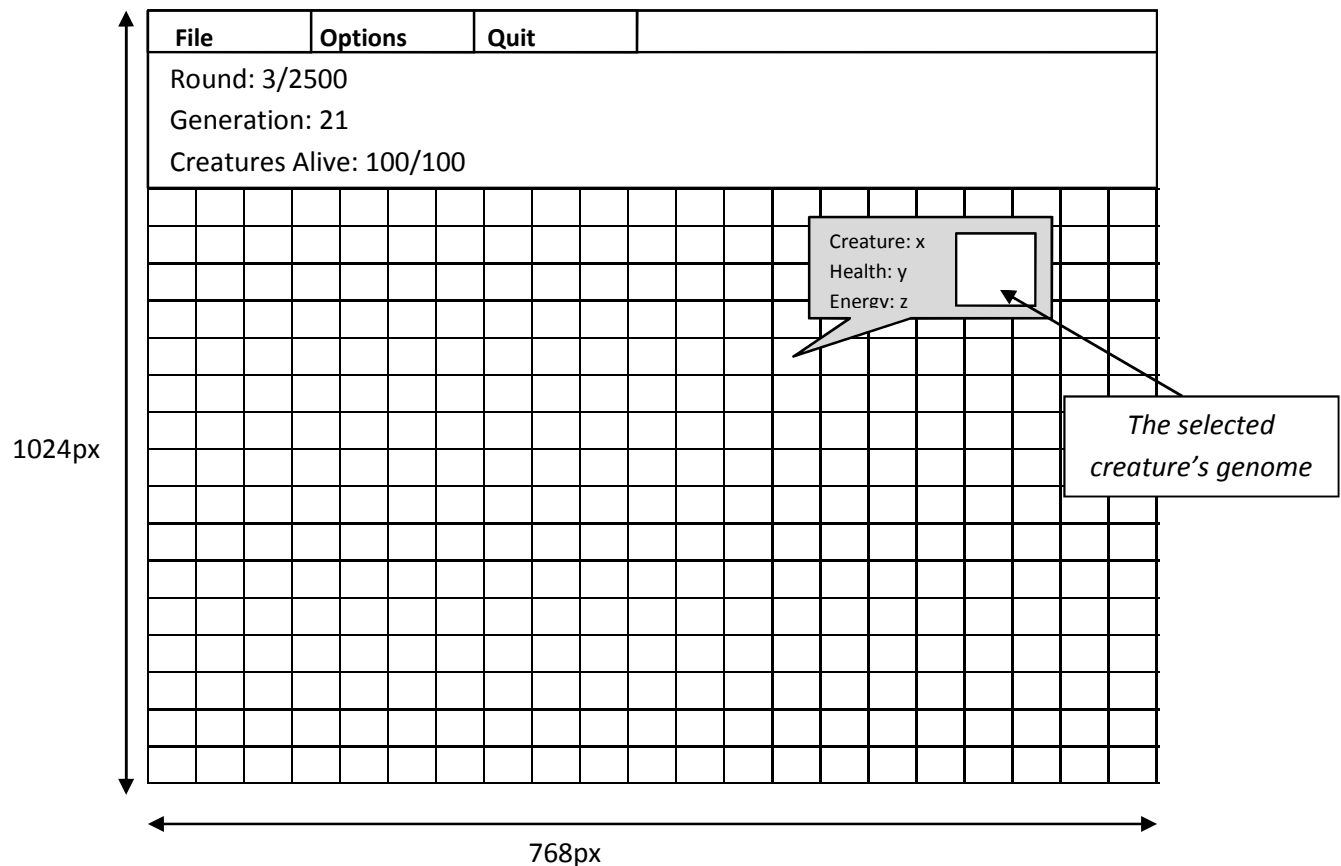
($|x|$ is the absolute value of x)

6.5 GUI Design:

The GUI design takes into account the purpose of each section of the program. The two states I have designed the GUI for are the running simulation, since that is what the user spends most of their time looking at, and the options menu, since it needs to be intuitive.

Since these are just designs they are not as detailed as the final product will be. I have included after each design some of what was cut out of the design.

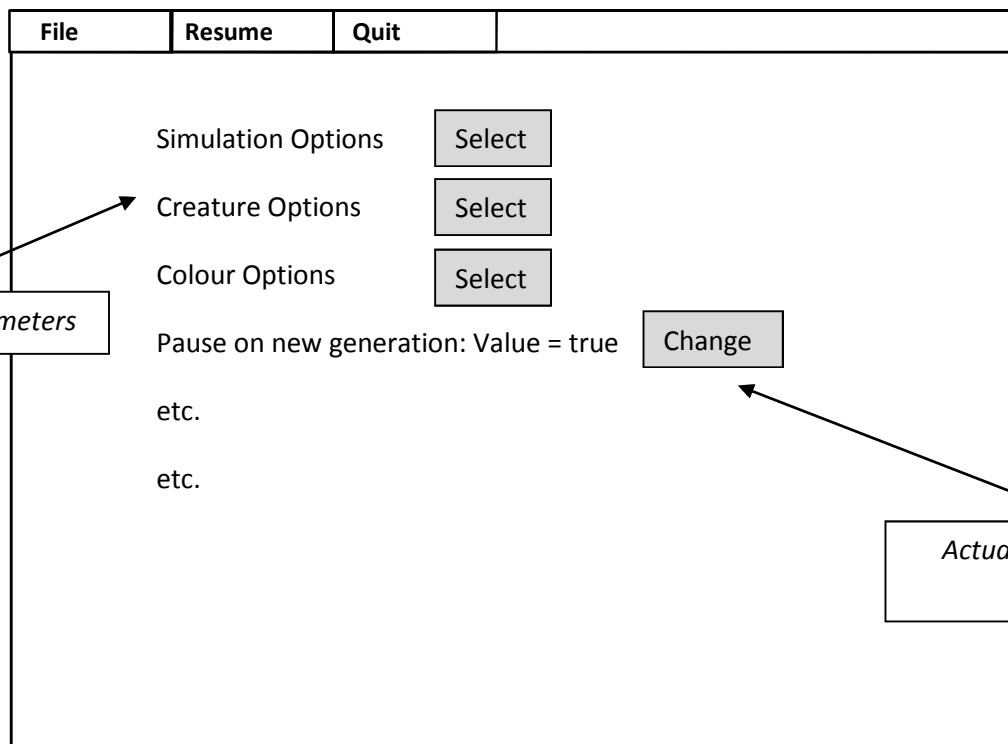
Simulation Running:



Not Pictured:

- Each tile in the world will have a different image depending on what its contents are.
- Tiles and creatures will be coloured the same colour as they are.

Options Menu:



Not Pictured:

- The buttons will use gradients to appear more aesthetically pleasing and to make them more visible to users.

7. Implementation:

7.1 Introduction:

This section of the report deals with the issues that were run into during the implementation of the project. It features discussions about deviations from the design and why these changes were made, as well as data structures used in the project, examining what functions each part of the final product needed to fulfil and how the data structures chosen help meet these needs.

7.2 Changes From Design:

Menu Leaf Nodes:

In the design document, all types of leaf nodes are handled by the same class, with an arbitrary type supplied at runtime. However during implementation I realised that the variables to be modified were only ints and bools and that I wanted bools to just toggle when selected in the menu instead of going to an input screen. To this end, I wrote the leaves as 2 separate classes, one for dealing with ints and one for dealing with bools. This also simplified the writing of the InputGetterOption class since it allowed the class to disregard any input that was not numbers or number related (operations etc.)

Menu Structure:

The menus were planned to be handled by the same set of classes, with the both menus being a specialised implementation of these classes. While I was writing the main menu, I decided to use a simpler design and make the menu simply a list of 4 buttons. This allowed me to create the main menu without needing the more complex set of classes that the options menu required. Since the button class had already been created at that point I only needed to create simple subclasses for all the buttons necessary, something that took almost no time at all. By creating this class separately from the options menu I was also able to make the menu options more specialised to the options menu and in the finished product they have a different appearance to the other buttons used in the product.

Shape class:

The shape class was initially a subclass of Gene since they seemed like they were closely enough related, however I had not accounted for the fact that the Genome class needed to do pattern matching when created, something Shape had no use for. To solve this I removed Shape as a subclass of Gene, making it so they did not have to perform the checks Gene did. To try and avoid some code reuse I did however reuse the Cell class from Gene to represent the cells in Shape.

How the world is updated:

One of the most important changes from the design is in the WorldState class. In the design the WorldState was supposed to update all the creatures, plants and remains in it each update. While this was acceptable for small numbers of creatures it did not scale well as more and more were added. Having more than about 20 creatures led to slowdown, stuttering and input lag, and

obviously I could not account for future cases when the user set the population to a very high number. To solve this I changed the way the WorldState updated.

The new update method in WorldState uses a pointer value to indicate the next creature on the list to update. It now updates the next creature in the list, increments the pointer value and checks if it has reached the end of the list of creatures. If it has then it updates the plants and remains, ticks the simulation and then resets the pointer to 0. The reason that updating creatures is the only thing split up like this is because it is by far the most resource intensive update method and so takes the longest to carry out.

This loop based paradigm was also used for the generation of creatures, both in the initialisation state and in the judging state, since initialising creatures can also take a variable amount of time based on the number of shapes that the creature has to be checked against, and the number of creatures that need to be created.

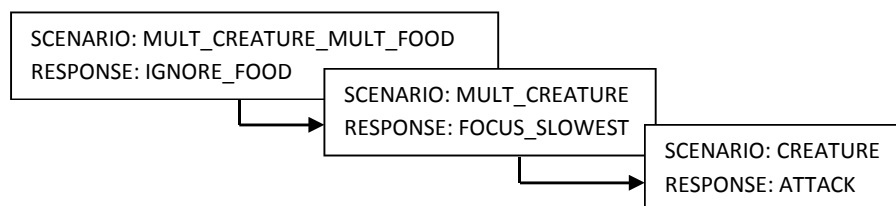
More drawing classes:

The change to a loop based paradigm meant that many of the classes I had assumed would execute in a single update needed some display class in order to show a progress bar or other status indicator. To do this I wrote a single class: SingleStringDrawer that just drew a single string in the centre of the screen and then made it so that any class that just needed a progress bar had a suitable method with a string output to use that drawing class.

Creature behaviours:

An important part of my design of the project was to have creatures capable of making distinctions between being able to see one food source or creature and multiple food sources of creatures and for creatures to be able to make different decisions based on what they saw. Initially I had behaviours for viewing multiple objects as things like “ambush closest”, “attack weakest” or “ignore all”. The problem with this way of dealing with multiple objects was that it limited what the creatures could possibly do when confronted with multiple objects, and the limits are different to what they can do if they see a single creature.

The solution to this problem was to examine the different actions creatures could take and deciding on a set of ‘atomic’ actions, actions that take a single object and act on it alone, and then have other responses to simply select which objects to focus on or ignore, applying an atomic schema to a single object once it is the only one the creature has not stripped out of its perception. For example if a creature runs across multiple food sources and creatures it could have this chain of decisions:



This way of dealing with multiple objects allows more interesting interactions between different parts of the genome, for example a creature that will wants to attack other creatures might evolve

to focus on the most wounded creatures, whereas something with a lot of defence might evolve to ignore other creatures if there is food nearby they can eat.

More detailed information on this new behaviour model can be found in Appendix D.

Inclusion of Display class:

While I was coding the project I decided that contrary to my thoughts during the design phase there was enough shared code between the different drawing classes to warrant including a class to collect it all. Instead of making it a superclass however I made it a class with several static methods and variables, allowing it to store the important variables for the drawing classes in a way that avoided problems of duplicates appearing.

Stat handling:

One of the other modifications to the design is how stats are handled. Some stats are modified so that the effects from the pattern matching are either increased or decreased. For example energy is modified so that any positive or negative changes add 100 energy units instead of just 1, while awareness is modified so that it takes a large number of positive or negative patterns to be found before the sight radius is increased or decreased by 1.

7.3 Data structures:

This section examines the data structures chosen for the project. It looks at the most important needs of several parts of the product and how the data structures used helped meet those needs.

7.3.1 WorldState:

The world state needed to be able to efficiently check for objects in the world at specific location, easily update the creatures, plants and remains that are present in the world, making sure to update the creatures in order of speed, with fastest updating first. It also needed to keep track of the creatures that died, including when they died, since the fitness test for dead creatures is based off of this.

Checking for objects at locations in the world:

To check for objects in the world the WorldState stored the state of the world as a 2D array of Tile objects. This allowed the lookup of locations in the world by using the co-ordinates as the indices of the array.

The array was stored in the format `array[row][column]` rather than `array[x][y]`. This was done to allow easier writing of the world to the console or a file if such functionality was needed since it allowed line by line output. To ensure that the lookup of the tiles found the location in x and y the `getTile(int x, int y)` method was used for all access to the array and that method reversed the inputs so it looked up `array[y][x]`.

The array used was also a jagged array, meaning different subarrays within the 2D array could have different lengths (eg. `tile[0]` can have length 3 while `tile[1]` can have length 4), this functionality was not used, and there are other methods of creating 2D arrays within C# that automatically have all the sub arrays with the same length. The reason a jagged array was used in this project is because this is a structure I am more familiar with from programming in java, and was thus something I was more comfortable with using, and more confident I would not make a mistake when using.

Updating Creatures, Plants and Remains:

In order to update the Creature, Plants and Remains present in the world the WorldState stores lists containing all the objects belonging to each type of WorldObject.

While it would be possible to update all these world objects by scanning through the world tile by tile and updating each as it is found this approach presents several problems. Firstly it is very inefficient, requiring up to check all the empty tiles to see if they contain an updatable object instead of just going to the objects we do want to update. Secondly it could cause issues with creatures moving when they are updated into a tile that will be updated in the future, which could lead to them being updated multiple times per full tick of the world. Finally it presents problems with updating the Creatures by speed since the whole world would need to be scanned through to find the next fastest creature to update each time.

The use of lists solves all these problems. The efficiency and multiple updates per tick are solved since it allows the world iterate through the list and update each object that way. Updating the creatures in order of speed is also handled by this data structure, through use of the List's built in sort method. Using sort allows the writing of a simple comparison method, comparing two Creatures by speed, which can then be used to sort the entire list. This sorting method is based on the quicksort algorithm and has average efficiency of $O(n \log n)$ (MSDN Library). The downside of the list sorting method, that it is not a stable sort, is not a concern when running the world since the sort is only performed once during initialisation.

Another benefit of using lists is that they have variable length, allowing creatures created during the running of the simulation (through the user's ability to clone creatures) to be added to the list of creatures without fear of running out of space in an array.

Keeping Track Of Dead Creatures:

The world also needs to keep hold of the creatures that have died, including the order in which they died, since they are included in the breeding pool for the next generation, and the fitness function for dead creatures is based off of time alive.

A stack was used to track this information. It was used since it stores the order in which the creatures died as part of its structure, creatures are added as they died and the FILO structure ensures that the top of the stack is the most recently dead creature. A typed stack was used to avoid having to cast back to creatures when getting output.

7.3.2 Creature:

The creatures need some way to keep track of the gene associated with them, their location in the world and the behaviour and stats they have based on the gene associated with them.

Gene:

The gene associated with the creature is stored as a Gene object which is discussed in more detail in the next section. The storage of the gene as its own separate object was done since it allows the pattern matching and breeding methods to be part of the gene class, preserving the cohesion of the creature class by making its methods concerned solely with the behaviour and current status of the creature.

Location:

The location of the creature in the world needed to be stored so that it can call methods in World to check nearby tiles and move. The location is stored as a Vector2 in the WorldObject superclass which will be looked at below.

Behaviour & Stats:

The behaviour of the creatures was stored as a Dictionary mapping a Scenario to a Response, this Dictionary is set up as part of the initialisation of the Creature objects. Using a Dictionary for this purpose allowed me to split the behaviour code into 2 sections, one to decide what scenario applied and one to call the correct method for the response. This allows a small amount of extensibility since what responses are possible for each Scenario can be modified in the extensions of the Scenario enum.

The stats of the Creature are stored as ints and doubles. The stats start at an initial value, are modified based on the Gene and then stored. This way of setting up the stats allows easy modification of the base values of the Creature's stats, which made it very easy for me to tweak them during debugging and testing. This also means that a further extension to the project, allowing the user to modify these values during the running of the simulation would be quite simple to implement.

The way Gene passes the modifications it makes to stats is dealt with in the next section.

7.3.3 Gene:

The gene needs a simple way to store the modifications it makes to stats to the creature, as well as a way to keep track of the dominant and recessive colours in each cell of the Gene.

Modifications to Stats:

The gene stores the various modifications it makes to the stats in the form of 2 lists of ParamTokens, with each entry in the positive list being a positive point to add to one of the stats, and each entry in the negative lists being a negative point to add to one of the stats. This method of storing the tokens was chosen since it is simple to set up and easy for the creature to act on by simply iterating through each list in turn.

Colours:

Colours of each part of the gene are stored in cells. Colours are stored as int values. The use of int values allows the colours each int is displayed as to be modified by simply changing the colour map stored in the Display, as well as making it easy to check if 2 colours matched since comparing 2 primitives like ints is far simpler than comparing 2 objects like the Color objects that were the other option.

Using Cells allowed the gene to store the dominant and recessive colours in a simple package that also contains methods to allow getting either the dominant or recessive colour at random.

7.3.4 WorldObject:

The most important function that the WorldObject has is to keep track of where the object is. This information is stored as a Vector2. Vector2s were used for this for 2 reasons. Firstly the Vector2 has which part of it is the X co-ordinate and which part is the Y co-ordinate built in, whereas a 2 element array can have some uncertainty if it is using [row, column] or [x, y]. Secondly Vector2s were used since I had initially thought I would need the normalisation method for some of the things I wanted the WorldObjects to be able to do. When I realised I did not need this method I had already created the class using Vector2s.

7.3.5 Tile:

The tiles are part of the world and need to be able to tell the world what type of WorldObject is in them, if any. Tiles store each potential object that could be in them in a separate variable, that is, they have a variable for a Creature, one for a Plant, one for a Remains object etc. This was done since it allows easy typing of the item in the Tile, as well as easy checking if the Tile is empty since it is just a matter of checking if all the variables are null.

7.3.6 OptionsMenuState:

The options menu needs to be able to split the different variables available for viewing into categories and allow the user to move between these categories, view the current values of the variables and modify them. The menu also needs to be able to go 'up' one level, returning to the previous list of options.

The options menu uses a tree like structure with the top level OptionsMenuState serving to point to the current node, keep track of previous nodes and pass any communication between options. The tree structure was the best choice for this menu since it is highly flexible and extensible, allowing me to easily modify it and add further options as I made more of the simulation parameters modifiable.

A Stack was used to store the previously selected options, since a stack allows the menu to pop the top element to move back one step, and allows the back button to be pressed several times without just moving back and forth between the 2 most recent nodes.

Getting the values for and modifying variables was done by assigning each leaf menu option a getter and setter method that would access and mutate the variable that they were concerned with. This allowed the options to interact with the various parameters of the simulation without making it necessary to make the variables public. Getting the input from the user is done by a special InputGetterOption that stores the input it has currently gotten as a string. This way of getting and passing input was chosen since it is easy to concatenate numbers onto the end of the string and the string itself is very easy to parse to an int.

7.3.7 JudgingState:

The JudgingState needs some way to sort the collections of creatures it is given into order based on the fitness criteria. A list is used to store and sort the creatures. A list was chosen because of the provided sort method. As in the WorldState class the sort method of the list allowed me to easily specify what should put any creature before or after the other in the list and allowed me to sort them all based on that.

8. Testing:

8.1 Introduction:

This section of the report deals with the testing carried out on the finished program. It includes a description of the testing procedure, and then descriptions of each test performed, including any errors found by these tests and the results of rerunning the tests after the fixing of any failed test. The majority of these tests are based on the acceptance tests in the requirements document (*Section 5*).

8.2 Testing procedure:

Most of the tests are carried out using pre-processor directives to access code that is placed in the `update()` method of the `Simulation` class and that terminates the program after it is run. This code outputs information to the console which is then checked manually to find the results of each test. While this is not as efficient as automated testing I do not feel confident enough in my C# knowledge to write such tests and while it may be more open to error this testing method still provides the repeatability of an automated test. If I had more time I would definitely try to spend some of it learning fully how to automate testing in C#, however as it is I am far more confident with the results of tests done this way.

One of the most important parts of the testing was to try and 'build up' the testing, that is, to verify that the lower levels of something were correct before testing the higher levels. To this end, the following tests are in an order that ensures the constituent parts of an object are tested before the object itself is.

Another important part of the testing was to focus on different extremes as input, for example I have ensured in the testing of Cell numbers that the input used numbers as far apart as possible.

Tests are divided into phases and sections, each phase includes a number of tests, and a new testing phase occurs only when some code has been rewritten. In a new phase all previous tests are repeated, to check they get the same result, though this is only shown if that test comes back with a different result.

In some of these test sections you will see `#define xxx` in `Simulation`. This is merely stating the name of the pre-processor variable to define in the code if you wish to run the tests in that section. Note that some tests will require you to also `#define DEBUG` in other classes to access debug code.

8.3 Genome Cell Testing (#define CELLTEST in Simulation):

Phase 1:

Test: A Cell will be created and given 2 different colours as its dominant and recessive colours, the colours and the one chosen as dominant will be written to the console and checked. This will be repeated to check for colours that require the checker to loop round or not and for inputting the colours in any order.

Input	Expected Output	Actual Output	Action
Colours 1 , 7	DOM: 7	Expected	None
Colours 7 ,1	DOM: 7	Expected	None
Colours 1 , 3	DOM: 1	Expected	None
Colours 3 , 1	DOM: 1	Expected	None

Test: A Cell will be created and given 2 of the same colour as its dominant and recessive colours, the colours and the dominant colour will be written to the console and checked.

Input	Expected Output	Actual Output	Action
Colours 0 , 0	DOM: 0	Expected	None
Colours 7 ,7	DOM: 7	Expected	None

8.4 Genome Pattern Testing (#define PATTERNTEST in Simulation):

Phase 1:

Test: 3 3x3 Genomes with a single positive shape, a single negative shape or no shape will be provided to the simulation and the pattern matching algorithm run, the results will be written to the console and compared manually to the expected results.

Input	Expected Output	Actual Output	Action
Genome of the shape: 100 010 001 Default shapes.txt file	POSITIVES SPEED NEGATIVES	POSITIVES SPEED NEGATIVES STRENGTH SPEED AWARE STEALTHVAL DEFENCE ENERGY HEALTH	Problem with pattern matching, further testing required
Genome of the shape: 100 010 000 Default shapes.txt file	POSITIVES NEGATIVES	POSITIVES SPEED NEGATIVES STRENGTH SPEED AWARE STEALTHVAL DEFENCE ENERGY HEALTH	Seems like pattern matching does not examine the whole object correctly, further testing necessary
Genome of the shape: 001 010 100 Default shapes.txt file	POSITIVES NEGATIVES SPEED	POSITIVES NEGATIVES STRENGTH SPEED AWARE STEALTHVAL DEFENCE ENERGY HEALTH	Seems problem is twofold: 1. Pattern matching does not read the full shape 2. Negative matching will always return true (Could be a symptom of 1) Step through pattern matching code to look for errors

Error found: The for loops in the cellCheck method in genome (which handles pattern matching) were written in such a way that the program would not iterate the row and column being checked so only the top left element of each shape was being looked at. The negative shapes were always returning true since they had a wildcard as their top leftmost element.

Phase 2:

Input	Expected Output	Actual Output	Action
Genome of the shape: 100 010 001 Default shapes.txt file	POSITIVES SPEED NEGATIVES	Expected	None
Genome of the shape: 100 010 000 Default shapes.txt file	POSITIVES NEGATIVES	Expected	None
Genome of the shape: 001 010 100 Default shapes.txt file	POSITIVES NEGATIVES SPEED	Expected	None

Test: A 3x3 genome with 2 overlapping shapes will be provided to the simulation and the pattern matching algorithm run, the results will be written to the console and compared manually to the expected results.

Input	Expected Output	Actual Output	Action
Genome of the shape: 101 010 101 Default shapes.txt file	POSITIVES SPEED NEGATIVES SPEED	Expected	None

Test: A 4x3 genome with 2 shapes of different colours will be provided to the simulation and the pattern matching algorithm run, the results will be written to the console and compared manually to the expected results.

Input	Expected Output	Actual Output	Action
Genome of the shape: 0611 1061 1106 Default shapes.txt file	POSITIVES STRENGTH ENERGY NEGATIVES	Expected	None

Test: A 3x3 genome with a single colour in the bottom right will be provided as well as a custom set of recognised shapes to make only that single colour cell a shape, this will check that the whole of the genome is being checked.

Input	Expected Output	Actual Output	Action
Genome of the shape: 000 000 001 Custom recognised shapes list containing only the shape 1, associated with strength.	POSITIVES STRENGTH NEGATIVES	Expected	None

8.5 Genome Polling Testing (#define POLLTEST in Simulation):

Phase 1:

Test: Genomes with all cells of the same colour will be provided to creatures and the behaviour deciding algorithm will be run, the behaviour of the creature will then be written to the console and the behaviour when facing a creature will be compared to the expected output (creature was used since it was different for each colour).

Input	Expected Output	Actual Output	Action
Genome with all cells as 0	ATTACK	Expected	None
Genome with all cells as 1	IGNORE	Expected	None
Genome with all cells as 2	HIDE	Expected	None
Genome with all cells as 3	AMBUSH	Expected	None
Genome with all cells as 4	DEFEND	Expected	None
Genome with all cells as 5	EVADE	Expected	None
Genome with all cells as 6	STALK	Expected	None

Test: A genome with all but the elements needed for the creature poll set to 0 will be provided to the creature and the behaviour deciding algorithm run, the results will then be output and the creature behaviour will be compared to what was expected.

Input	Expected Output	Actual Output	Action
Testing normal majority: 000 000 001	Response 0: ATTACK	Expected	None
Testing draws: 000 021 111	Response 0: ATTACK	Expected	None
Testing draws: 100 021 110	Response 1: IGNORE	Response 0: ATTACK	Examine polling method in Gene

Error Found: Due to rewriting poll at some point the for loops were going from 0 to 1 and subtracting 1 inside the loop instead of simply going -1 to 1. This meant polling was only checking a 2x2 area in the top left to find the majority.

Phase 2:

Input	Expected Output	Actual Output	Action
Testing normal majority: 000 000 001	Response 0: ATTACK	Expected	None
Testing draws: 000 021 111	Response 0: ATTACK	Expected	None
Testing draws: 100 021 110	Response 1: IGNORE	Expected	None

8.6 Need Fulfilment Testing:

Phase 1:

Test: A world will be generated with at least 1 creature in it, the creature will be observed acting to check if it moves towards food once it is within view of it.

Input	Expected Output	Actual Output	Action
Working simulation	Creature moves towards visible food, if it has that behaviour	Expected	None

8.7 Genome Efficiency Testing (#define DIETTEST in Simulation):

Phase 1:

Test: 2 Creatures, 1 fully a herbivore, the other fully a carnivore will be created, A plant and a remains will also be created, the creatures will be made to check how much energy they would get from each and this value will be compared to the expected value the food should have based on that creature's diet.

Input	Expected Output	Actual Output	Action
Herbivore creature Plant Remains	Plant : 1000 Remains : 0	Plant : 1000 Remains : 1000	Further testing required
Carnivore creature Plant Remains	Remains : 1000 Plant : 0	Plant : 1000 Remains : 1000	Examine diet values and checking of food against these values

Error found: A cast to int within the code to decide the diets would always set it to 1, furthermore the checker of values a for loop had the wrong check so the decrease as a herbivore was never happening.

Error found: A faulty check within the code to deal with all of colour 3 (which is the normalising colour and will move the diet value back towards 0.5 set diet to always be 0.5 if the creature was a herbivore)

Phase 2:

Input	Expected Output	Actual Output	Action
Herbivore creature Plant Remains	Plant : 1000 Remains : 0	Plant : 0 Remains : 0	
Carnivore creature Plant Remains	Remains : 1000 Plant : 0	Plant : 1000 Remains : 1000	Examine calculation of nutritional value to see where a mistake has been made

Error found: Plant does not override isPlant() and so is not identified as a plant by the creature, to solve this isPlant is now abstract in FoodSource and overridden in both Plant and Remains.

Phase 3:

Input	Expected Output	Actual Output	Action
Herbivore creature Plant Remains	Plant : 1000 Remains : 0	Expected	None
Carnivore creature Plant Remains	Remains : 1000 Plant : 0	Expected	None

Test: 2 Creatures, 1 fully a herbivore, the other fully a carnivore will be created, A plant and a remains will also be created, the creatures will be given the 2 foodsources in a list and be made to pick the most nourishing.

Input	Expected Output	Actual Output	Action
Herbivore creature Plant Remains	Picks plant	Expected	None
Carnivore creature Plant Remains	Picks remains	Expected	None

8.8 Breeding testing (#define BREEDTEST in Simulation(For test 2)):

Phase 1:

Test: The simulation will be run for the specified number of ticks and then the test will observe if the simulation changes state to begin breeding the new generation of creatures.

Input	Expected Result	Actual Result	Action
A working simulation	The simulation will breed the new generation after a set number of ticks	Expected	None

Test: 1 Genomes with all cells of a single colour will be mated with itself repeatedly and the results checked, eventually one of the children should contain a cell of a different colour.

Input	Expected Result	Actual Result	Action
A single genome with all cells set to 0, 0, mutation chance set to 1 in 100	One or two cells on each child might be mutated and contain a non 0 value	Expected	None

8.9 Visualisation testing:

Phase 1:

Test: The tester will load up the program, and using the debugging system examine the underlying data of the world, checking that at 5 visible entities in the visualisation match where they appear in the underlying data structure.

Input	Expected Result	Actual Result	Action
A working simulation	The underlying data will mirror the actual appearance of the world	Expected	None

8.10 Simulation Rule Modification Testing:

Test: Using the in-simulation menu the energy drain per tick for creatures was modified to be both higher and lower, a creature was then visually inspected to see if the change worked.

Input	Expected Result	Actual Result	Action
A working simulation	The higher energy drain per tick will be reflected in the creature's energy loss each tick	Expected	None

9. Demonstration:

9.1 Normal Execution:

Parameters Modified: This run through of the program was executed using the default parameters.

Tracking: I noted down the number of creatures left alive at the end of generations 1, 10, 30 and 100. If the evolutionary algorithm does provide a gradual improvement in creature fitness this number should increase.

Observations:

Generation Number	Creatures Surviving
1	17
10	29
30	31

These results show that survivability increases incredibly quickly when transitioning from random creatures to evolved creatures, and then increases more slowly.

9.2 Carnivore's Paradise:

Parameters Modified: For this run through I increased the food value of remains to 5000, increased the number of food units per remains to 5, decreased the food value of plants to 250, increased the creature population to 500 and decreased round lengths to 5000 ticks.

Tracking: I examined a random sample of 10 creatures at the start of generation 31 to see what diet they favoured. I would expect that the creatures will be more carnivorous with these parameters.

Observations: Of the creatures I looked at, all had a diet value at or below 0.5, making them herbivores. This was an unexpected result. This result may have been caused how high I set the value of meat to, while keeping plants just as numerous with only a lower energy value, possibly causing creatures to attempt to wring as much energy as possible out of the more abundant plants, while remains were so good regardless that efficiency with them didn't matter as much. While I recognise that this sample is not ideal I do not have any easy way of getting a larger sample, and in fact a way to get more information on the simulation results is one of my recommended extensions (*Section 10*)

9.3 Trees instead of shrubs:

Parameters Modified: The Plant objects in the default world have high population, low amount of food on them, and middling nutritional value. This makes the creatures grazers. For this run through of the simulation I modified the plants to have 100 food units of 250 nutrition each, and to have 1/10 of the default population. I also made the creature population 500, raised the starving energy level to 5000 and reduce energy drain to 1 per tick.

Tracking: I will examine random samples of creatures at generations 10, 25 and 50 and look for interesting patterns in the genes. I will also watch for interesting events while the simulation runs.

Observations:

There are several interesting things to be observed when running the simulation with this set of parameters. Firstly the creatures are far more aggressive, since food is more rare to find they are more likely to attack and eat each other, and this was seen in the genes I looked at with a gradual shift towards carnivorism.

Secondly I saw several of the example events I specified during the desired behaviour section of the requirements (*Section 5*) actually occurring. Specifically I saw behaviour changes due to starvation, as well as creatures running from food when others appeared nearby.

Something else I saw that was interesting was a creature that was hiding and watching another creature eat food. The other creature was not hostile, but the hiding creature would not go towards it and eat. This changed when another creature approached and ate at the tree, and soon all 3 creatures were eating from the plant. The reason for this was that the creature had a hide response to seeing a single creature and a ignore food response to seeing a single creature and food, however multiple creatures and food triggered an ignore creatures response. But what was more interesting was that this behaviour was actually quite intelligent. The hiding creature had no idea if the eating creature would attack if approached so it waited for another creature to see what happened. While this led to it losing out on food it could have prevented it from being killed.

One thing I also found was that the creature's rarely had high stats. This is likely because the simulation was not run long enough or that the shapes to recognise were too complex to evolve into by chance, and thus could not be selected for. For a future demonstration I could edit the shapes to be 2x2 rather than 3x3 and see if that produces more of an effect.

9.4 Conclusion:

The interesting observations, not to mention the scenarios from the desired behaviours section actually occurring shows that the simulation is able to produce complex creature behaviours and that the interactions between the different scenarios can lead to interesting behaviours.

More importantly, the fact that I could edit all these parameters in order to produce several vastly different instances of the simulation shows that this tool could serve as a good base for further research.

10. Conclusion:

10.1 Goals:

10.1.1 Achieved:

The product testing that was carried out in section 9, and the brief demonstration in section 10 confirms that the product meets all of the requirements specified for the product in the requirements documentation. Since the requirements were based off of the goals specified in the proposal I can confidently say that those goals were met as well, with some exceptions which will be discussed in 11.1.2.

The personal goals I had for this project were fairly simple. Primarily I wanted to use this project as a way to learn C# and XNA. While I do think I have a far better idea of the options and limitations of both of these environments now I recognise that there is still very much left to learn and I hope to continue developing my skills with these in the future.

Another personal goal I had was to get better at estimating how long it would take me to perform various tasks necessary for development. I feel that I failed in meeting this goal, since my plan had me finishing the project much earlier than I actually managed to. Some of this was due to the rearranged year structure, since I had to spend a lot of time I had thought to do the project in on revision, and some of it was just due to over optimistic estimates, possibly because I was not very familiar with what I was working with. In the future I need to modify plans to leave more slack time for unplanned breaks in work.

Finally I also wanted to make sure I wrote the program in a way that was both easily extensible and simple to maintain. I feel that I have partially met this goal. Some parts of the program are very easy to modify, for example the list of shapes the genes use for pattern matching, whereas some part would be very difficult to modify or extend, such as the way the creature selected the scenario and response to use. While programming the project I thought of several ways to improve extensibility in these areas, these alternatives are discussed below.

10.1.2 Not Achieved:

While the most crucial systems were implemented fully there was one thing in the specification that was not implemented fully and that was the colour system that would have been used by the stealth system. The colour system was one of the less crucial systems, since it was only used for stealth checks, which rarely happened. Since it was a low priority goal it was left until later in the development and time unfortunately ran out.

The stretch goals were also not achieved. They had been planned for if the product was completed far earlier than was planned, however since the implementation took more time than expected the stretch goals were not attempted. These would serve as a good starting point for an extension to the program should more work be done on it in the future.

10.2 Alternate Implementations:

Scenarios & Responses:

The current failing of the scenario system is twofold. Firstly it is difficult to add scenarios to the current batch and have them properly detected. Secondly the way responses are handled requires lots of code reuse and redundancy within the Creature class.

The first step to solving this would be to make Scenario a class. Scenarios would need to have a list of conditions and a list of responses and some int to identify the scenario. The list of conditions could be of any length, but the list of responses would need to be 7 elements long.

Conditions would be stored as a list of representations of world objects, possibly as an enum, with the possibility of modifiers similar to regular expressions. When a creature wants to decide what behaviour to use it would then need to scan through the list of scenarios to find one that best matches what it sees, with the match being as detailed as possible, for example if it saw 3 plants it would prefer to use (plant plant plant) over (foodsource*) or (plant*).

This would solve one problem, since it would make it far easier to add further scenarios to the simulation and the scenarios would be better able to deal with more conditions in a better way.

For responses they would need to be divided into two groups, selectors and actions. Selectors would simply contain what type they are and a pointer to part of the conditions of the scenario. They would just tell the creature to ignore part of what they see and rerun the decision making test. Actions on the other hand would be more similar to what we have now and would include attack, eat etc.

Scenarios would need to be stored as a static variable in the Creature class since they would not be initialised multiple times. Creatures would keep a list of ints based on polling each part of their gene when they are created. When they need to decide what to do they find the best fit they get the int value stored it, use that to look up the int they've got in the response array and then pass both the response number and themselves to the Scenario which looks up the correct response and calls the correct methods in creature to make it happen.

Adding Shapes:

While it is quite easy to add and modify shapes for the genes to detect there are several ways to make the process simpler. The best way would be to allow the genome editor module, if it was completed to modify and save shapes. Regardless of that however it would also be better to store the current shapes not as a text file but as an XML file since that allows easier parsing and is in general more readable.

10.3 Possible extensions:

This section is a list of possible areas to extend the program, in order to add additional or alternative functionality.

Editors:

An obvious extension to the program would be implementing the stretch goals set in the proposal and formalised during the requirements phase. These include things like being able to modify genes in a simply painter program (by which I mean you can paint cells different colours with the mouse) and then be able to export these genes into the world.

Save / Load Functionality:

Another extension would be the ability to save and load the current state of the Simulation onto the hard drive and reload it later. This would require both saving the current state of the world, such as the locations of all the creatures and objects, as well as the seed and current status of the random number generator so that reloading will give the same results as just continuing to run the simulation, since if the random generator is just reinitialised it will begin the sequence of random numbers over again.

Statistic Viewer:

One of the things I feel is missing from the program in its current state is an easy way to get information out. You can input lots of information and modify lots of parameters but there is no easy way to view the results of these modifications. The solution to this would be to implement some way to output number values and current genes to a file, including average values of various things like diet and all the various stats. Better still would be to be able to graph these things from within the program and be able to see whether or not the evolutionary process is actually improving creature lifespans by viewing the number of creatures alive at the end of each round so far for instance.

Additional Stat Effects:

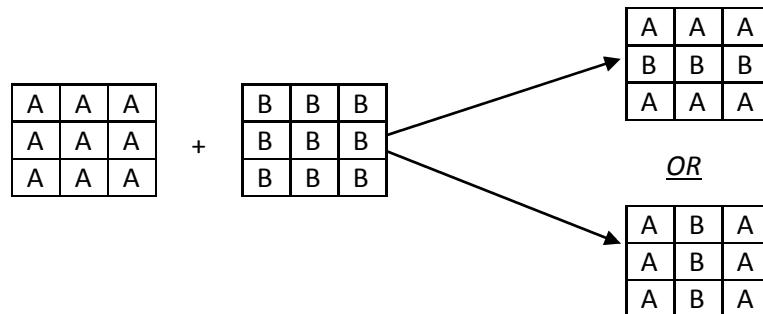
Another good extension would be to give stats further effects. Currently they all have certain effects that make sense for what they are. Some however, particularly speed, don't have enough effect on creature lifestyle. Something that it might be useful to implement is have faster creatures take more actions per tick, or just move further, giving better benefits for speed.

Further Parameter Modification:

Being able to modify even more parameters of the simulation would also be a good thing to add, for example being able to modify what the base values of each stat are and how much they are divided by as they are set would be good, since it would allow the user to specify if they want, for example, patterns that give strength in the genes to be very inefficient, requiring multiples to add or subtract one point, compared to points of awareness.

More Types Of Breeding:

Adding more types of breeding is something else that was specified as a stretch goal in the proposal, and would be an interesting thing to compare with creatures with the exact same state in every other sense. Other forms of breeding could be strip based, like so:



Or chance based where each cell is either parent As or parent Bs based on a coin flip.

Another way to change breeding would be to change *when* the breeding is done. As it is, the creatures are selected at the end of the round and are all sorted, eliminated, bred etc. An alternate way to breed the creatures would be to do the 'best of 3' type breeding, which works by selecting 3 creatures from the population. It would then compare these, eliminate the worst performing and breed together the other two. This method of breeding would entirely remove the discrete generations, instead having a gradual change every n ticks. This method would cause a problem with the creatures moving in order of speed, since every time one is added the list needs to be resorted. It would also need to have a delay before a creature is eligible to breed, to allow the creature to make some decisions. Finally there would need to be a way to repopulate the world if creature populations started to get very low.

Yet another way to implement breeding would be to have creatures above a certain energy level seek out other creatures to breed with them. The act of breeding would have to drain a large amount of energy and the energy level that creatures seek breeding at would have to be above the base initial energy level to avoid creatures all breeding at the start of the round, and creatures would need some way to seek out a successful mate, perhaps one that allows them to scan a larger area just for this purpose to avoid chance having too much of an effect of the evolution. Finally as before there would need to be a way to repopulate the world if the population gets too low, this could be either creatures giving birth to more creatures per breeding encounter when population is low, or just regenerating the world when all creatures die.

10.4 Experimentation:

This project was designed to be highly extensible and flexible to allow people more scientifically minded than myself to pick it up and perform experiments, either as it is or with additional functionality. The ability to change so many of the parameters used by the Simulation, and the ease with which it is possible to make even more parameters modifiable should allow a wide range of experiments to be performed using this program.

11. References:

- British Computing Society, 2011. *BCS Code of Conduct*. [online] Available at: <<http://www.bcs.org/category/6030>> [Accessed 5 November 2012]
- British Computing Society, 2004. *The British Computer Society Code of Good Practice*. [pdf] Available at: <<http://www.bcs.org/upload/pdf/cop.pdf>> [Accessed 5 November 2012]
- Desutter-Grandcolas L. and Robillard T., 2004 *Acoustic evolution in crickets: need for phylogenetic study and a reappraisal of signal effectiveness* [online] Available at: <http://www.scielo.br/scielo.php?pid=S0001-37652004000200019&script=sci_arttext> [Accessed 6 November 2012]
- Floreano D., Husbands P. and Nolfi S., 2008 Evolutionary Robotics In: Siciliano B. and Khatib O. eds. 2008 *Handbook of Robotics* Berlin: Springer Verlag Chapter 61
- Pichler, P. and Canamero, L. 2007. An Evolving Ecosystems Approach to Generating Complex Agent Behaviour. In: IEEE (Institute of Electrical and Electronics Engineers), *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*. Honolulu, Hawaii 1-5 April 2007
- Ray, T.S., 1993 An Evolutionary Approach to Synthetic Biology, Zen and the Art of Creating Life. *Artificial Life* 1(1)
- MSDN Library, *How to: Read Text from a File*. [online] Available at: <<http://msdn.microsoft.com/en-GB/library/db5x7c0d.aspx>> [Accessed 10/03/2013]
- keshtath, 2007 [.net] [XNA] *Getting Text from keyboard*. [Forum post] Available at: <<http://www.gamedev.net/topic/457783-xna-getting-text-from-keyboard/#entry4038836>> [Accessed 17/03/2013]
- MSDN Library, *List<T>.Sort Method*. [online] Available at: <<http://msdn.microsoft.com/en-gb/library/b0zbh7b6.aspx>> [Accessed 05/04/2013]
- Canas, J. M., & Matellán, V., 2002. *Dynamic schema hierarchies for an autonomous robot*. In: *Advances in Artificial Intelligence—IBERAMIA 2002* (pp. 903-912). Springer Berlin Heidelberg.
- Harvey, I., Husbands, P., Cliff, D., Thompson, A., & Jakobi, N., 1997. Evolutionary robotics: the Sussex approach. *Robotics and autonomous systems*, 20(2), 205-224.
- Source Making, *Design Patterns*. [online] Available at: <http://sourcemaking.com/design_patterns> [Accessed 17/04/2013]
- Ventrella, A., *GenePool: Exploring the Interaction between Natural Selection and Sexual Selection*. [pdf] Available at: <www.ventrella.com/Alife/GenePool.pdf> [Accessed 17/04/2013]
- TamuECE, 2009, *Predator Prey Autonomous Robots*. [video online] Available at: <http://www.youtube.com/watch?v=RvB6_FQuDhg> [Accessed 17/04/2013]

Appendix A - User manual:

User Documentation for the Genome Project:

Installing the program:

Simply run the ClickOnce install package provided to install all necessary assets and environments.

Running the program:

Once the program is installed it can be run by double clicking Genome.exe or using the start menu option.

How to use the program:

Start-up:

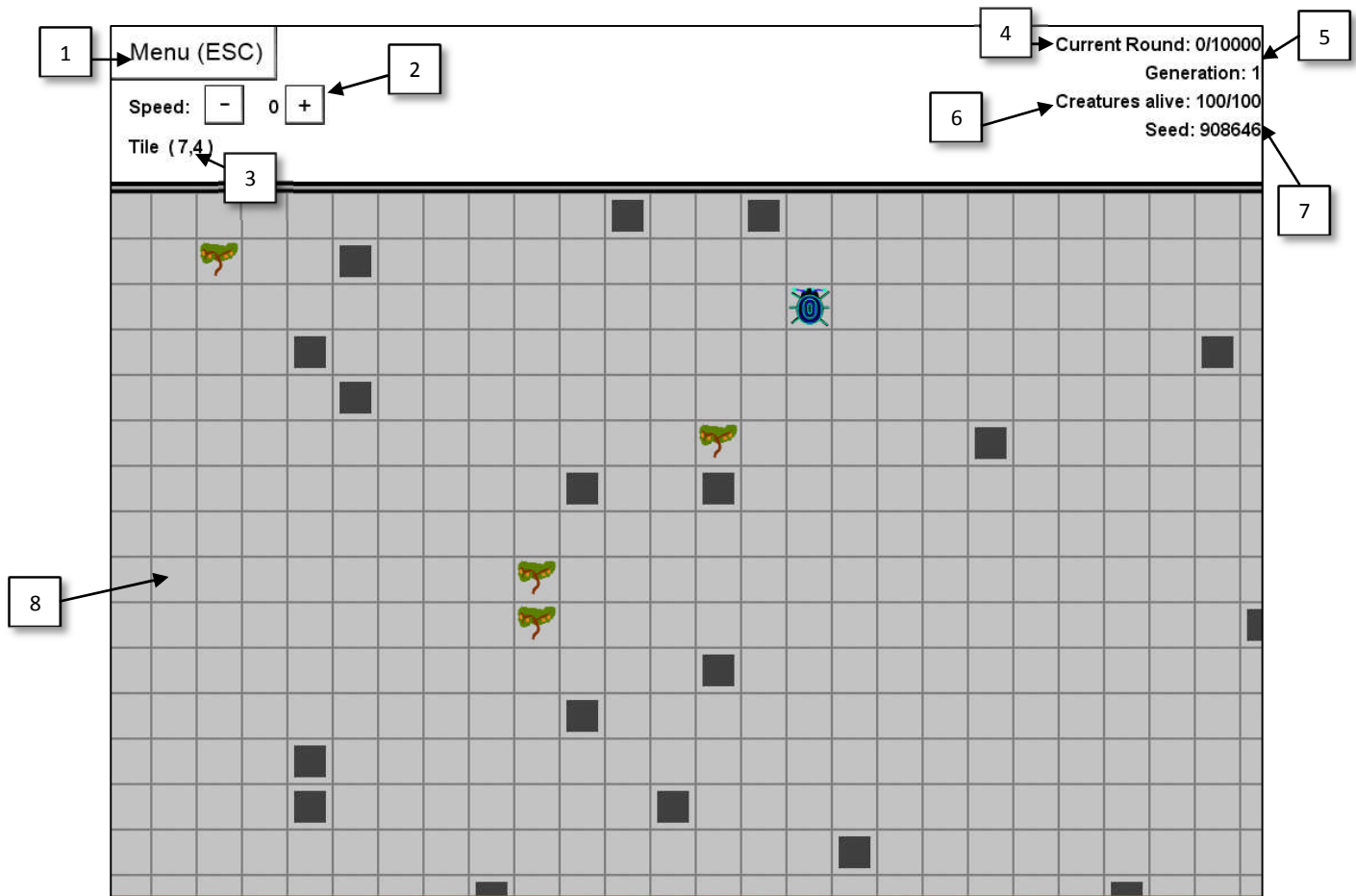
Once you have started the program you will see the following screen:



This screen shows the progress of the simulation in generating the initial generation of creatures, for this first set the default values of everything will be used, don't worry! If you want to change any settings it is easy to do.

The World Viewer:

Once initialisation has been finished you should see the following screen:



Here is what each part of this screen means:

1. Menu Button – Press this or press ESC to go to the main menu, where you can access options, quit or reset the simulation
2. Speed Controls – The number displayed is the speed of the Simulation, the buttons provided will increase or decrease it
3. Tile number – This is the location of the tile the mouse is over in the form (X,Y)
4. Current Round: Generations are split into a set number of rounds or ticks, during which each creature acts once, this number shows the current round and the number of rounds in each generation
5. Generation number: This shows how many generations of creatures we have gone through
6. Creatures alive: Shows the number of creatures alive in the world in relation to initial population
7. Seed: Shows the seed used when creating the world
8. The world view, displaying the current state of the world as a series of tiles. To move click and drag in the direction you want to go (more below)

Interacting with the world:

Click and drag within the world space to move the view around the world. Click on any tile containing a creature, plant or remains object to view that object. The view will be centred on that object, and in the case of creatures the view will follow them around.

Tiles:

The current state of the world is displayed as a series of images, here is what each image means:

Blank Tile:



This tile is blank, and can be moved into by any creature

Creature:



This tile contains a creature, you can click on the creature to bring up a more detailed view

Plant:



This tile contains a plant, if the plant appears as the tile on the left it has some food left on it, if it appears as the tile on the right it is depleted and barren of food, click on the plant to get a more detailed view

Remains:



This tile contains the remains of a dead creature, click on it to get more information

Obstacle:



This tile contains an obstacle that blocks the path of creatures

Options:

If you go to the main menu and click on the options button you can change a variety of different settings related to the running of the simulation, the function of each of these is explained in more detail when they are clicked on.

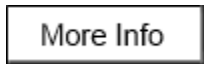
IMPORTANT: FOR SOME SETTINGS IT WILL BE NECESSARY TO RESTART THE SIMULATION FOR THEM TO TAKE EFFECT, TO DO THIS RETURN TO THE MAIN MENU AND HIT THE RESTART BUTTON.

Further options:

The shapes recognised by the simulation when pattern matching can also be modified, to do this go into the .txt file found in the Content folder under the base genome.exe add and change the contents, make sure you back this file up before making any changes. If you hear a beep when starting up the program then the file is not correctly formatted. An explanation of how to format new shapes can be found within the file

The Viewer:

If you click on a creature, plant or remains you will see a viewing window that displays more detailed information about whatever it is you clicked in, you also get a number of different options depending on the type of object you clicked on.

 More information: Clicking this button will cycle through a number of different screens of information about the selected item.



Clone: Clicking this button will clone the selected creature and place the copy at a random location in the world



Kill: Clicking this button will kill the selected creature and remove it from the world

Known Issues:

The program will close due to an ObjectDisposedException if your computer goes into sleep mode while it is running.

The program will run slower if it is not the active window.

Appendix B: Project Proposal

Candidate Name: Owen Cox

Supervisor: Phil Husbands

Working Title: Parameter based a-life with an abstract genome

Aims and Objectives:

Motivation:

Examining the patterns produced by an evolutionary algorithm with this form of abstract genome will definitely give interesting results, especially if the user is allowed to modify the simulation and run it under different conditions.

Primary:

Creatures:

- Implement simple creature objects with an associated genome
- Implement creatures having abilities that are affected by the patterns found in their associated genome
 - Strength, Sight range, Speed
- Implement creatures having behaviours that are affected by viewing sectors of their associated genome
 - On seeing hostile creature ...
 - On seeing weak creature ...
- Implement creatures having basic needs and the drive to fulfil these needs
 - Food, Rest, Health, etc.
- Implement genome affecting how and how well creatures can fulfil certain needs
 - Carnivore vs. Herbivore vs. Omnivore
- Implement creature to creature combat

Genes:

- Implement genes as 100 x 100 grids of coloured cells (7 Colours)
- Implement a pattern recogniser that can find patterns of cells within the genome
 - Eg: Red|Red|Red
- Implement a dominant and recessive system of mating
- Implement pseudo-randomly generating genes.

World:

- Implement worlds with obstacles and food sources
- Implement pseudo-randomly generating worlds

Simulation:

- Implement round based breeding
 - Need some measure of fitness and to eliminate the unfit
- Implement visualisation of the worlds, creatures and genomes
- Implement time step control of the speed of the program
- Implement simple modification of simulation rules

Stretch:

Allow the user to greater control over the simulation

Create editors to allow the user to create and use their own genome designs and worlds.

Add more methods of mixing genes to the program and allow the user to choose between them

Implement more breeding methods and ways of mixing genes in the simulation

Implement species in the program, restricting who creatures can breed with.

Implement actively friendly behaviours between creatures governed by the genome

Implement further, more complex behaviours for creatures

Relevance:

Being able to write algorithms for mixing genomes and program sensible finite state machines to govern creature behaviour is important for computer scientists. Additionally the software engineering methods used during planning and implementation of this complex system is taught by this course.

Resources Required:

Computer with:

- Microsoft visual studio
- XNA
- C#

installed, as well as standard operating system etc.

Access to university library and materials found in bibliography

Personal Timetable:

	Mon	Tues	Wed	Thurs	Fri	Sat
11:00			Human Computer Interaction			
12:00						
13:00						
14:00		Web Computing		Web Computing		
15:00		Web Computing	Project Work	Comparative Programming		
16:00	Human Computer Interaction			Comparative Programming	Comparative Programming	
17:00					Project Work	
18:00						Project Work
19:00	Project Work	Project Work	Project Work	Project Work	Project Work	Project Work

Reading List:

- Animals to Animats Volumes 1 to 12
- ECAL Papers
- Handbook of Robotics Chapter 61: Evolutionary Robotics
- Better Living Through Chemistry: Evolving GasNets for Robot Control
- Adaptive Behaviour: How Not to Murder Your Neighbor: Using Synthetic Behavioral Ecology to Study Aggressive Signaling
- An Evolving Ecosystems Approach to Generating Complex Agent Behaviour
- An Evolutionary Approach to Synthetic Biology, Zen and the Art of Creating Life
- O'Reilly Learning XNA 4.0 – Aaron Reed
- Pattern Recognition – Sergio Theodoridis, Konstantinos Koutroumbas

Interim Log:

28/9/12 –Reading: O'Reilly Learning XNA 4.0

1/10/12 – Arranged meeting with supervisor, asked about reading materials, other students doing similar work.

3/10/12 – Reading: Better Living Through Chemistry: Evolving GasNets for Robot Control

9/10/12 – Sent off Proposal

Appendix C: Project Log:

28/9/12 – Reading: O'Reilly Learning XNA 4.0

1/10/12 – Arranged meeting with supervisor, asked about reading materials, other students doing similar work.

3/10/12 – Reading: Better Living Through Chemistry: Evolving GasNets for Robot Control

9/10/12 – Sent off Proposal

22/10/12 – Sent Project Plan for feedback

25/10/12 – Reading: An Evolutionary Approach to Synthetic Biology, Zen and the Art of Creating Life

29/10/12 – Reading: An Evolving Ecosystems Approach to Generating Complex Agent Behaviour

2/11/12 – Sent Interim Report draft for feedback

5/11/12 – Updated Interim Report based on feedback

6/11/12 – Go more feedback on updated interim report

6/11/12 – Updated interim report based on feedback

6/11/12 – Gave in interim report

3/12/12 – Got Marker's feedback on interim report

4/4/13 – Asked about implementation discussion

7/4/13 – Sent project conclusion to supervisor for feedback

16/4/13 – Discussed Design document

17/4/13 – Got feedback on report draft

17/4/13 – Updated report based on feedback

Appendix D: Scenarios / Responses Used:

This appendix includes the details of the new scenarios and the possible responses to them.

I have not bothered explaining each response piece by piece since the names of the responses are fairly self-explanatory, and in the cases where they are not they were explained in the project specification in the requirements section. A more detailed description of each scenario and response can also be found in the Scenarios enum in the code.

	0	1	2	3	4	5	6
IN_COMBAT	ATTACK	DEFEND	EVADE	ATTACK	DEFEND	EVADE	ATTACK
IN_COMBAT_CREATURE	ATTACK	EVADE	FOCUS_CLOSEST	FOCUS_WOUNDED	DEFEND	ATTACK	DEFEND
IN_COMBAT_WOUNDED	ATTACK	DEFEND	EVADE	ATTACK	EVADE	EVADE	ATTACK
CREATURE	ATTACK	IGNORE	HIDE	AMBUSH	DEFEND	EVADE	STALK
FOOD	EAT	EAT	EAT	EAT_PREF	EAT_PREF	EAT_PREF	EAT_PREF
STARVING_FOOD	EAT	EAT	EAT	EAT	EAT	EAT_PREF	EAT_PREF
CREATURE_FOOD	ATTACK	EVADE	HIDE	IGNORE_FOOD_NON_PREF	IGNORE_CREATURE	IGNORE_FOOD	IGNORE_FOOD_NON_PREF
STARVING_CREATURE_FOOD	ATTACK	IGNORE_CREATURE	IGNORE_CREATURE	IGNORE_FOOD	IGNORE_FOOD_NON_PREF	HIDE	IGNORE_CREATURE
MULT_CREATURE	FOCUS_WEAKEST	FOCUS_SLOWEST	FOCUS_WOUNDED	FOCUS_LEAST_DEFENDED	FOCUS_CLOSEST	FOCUS_MOST_HUNGRY	EVADE
MULT_CREATURE_FOOD	IGNORE_FOOD_NON_PREF	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_CREATURE	IGNORE_FOOD
STARVING_MULT_CREATURE_FOOD	IGNORE_FOOD_NON_PREF	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_CREATURE	IGNORE_FOOD
CREATURE_MULT_FOOD	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_FOOD_NON_PREF	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_CREATURE	IGNORE_CREATURE
MULT_CREATURE_MULT_FOOD	IGNORE_FOOD	IGNORE_CREATURE	IGNORE_FOOD_NON_PREF	IGNORE_FOOD	IGNORE_FOOD_NON_PREF	IGNORE_CREATURE	IGNORE_CREATURE
MULT_FOOD	EAT_CLOSEST	EAT_CLOSEST_PREF	EAT_LEAST_DANGEROUS	EAT_MOST EFFICIENT	EAT_MOST_REMAINING	EAT_MOST_NOURISHING	EAT_CLOSEST_PREFERRED