Design

Contents

1 High-Level

The high-level design serves as a transition between the requirements and the full low-level design. Below are designs used to identify key components of the system and how they shall interact, both with each other and with the user.

1.1 Context Model

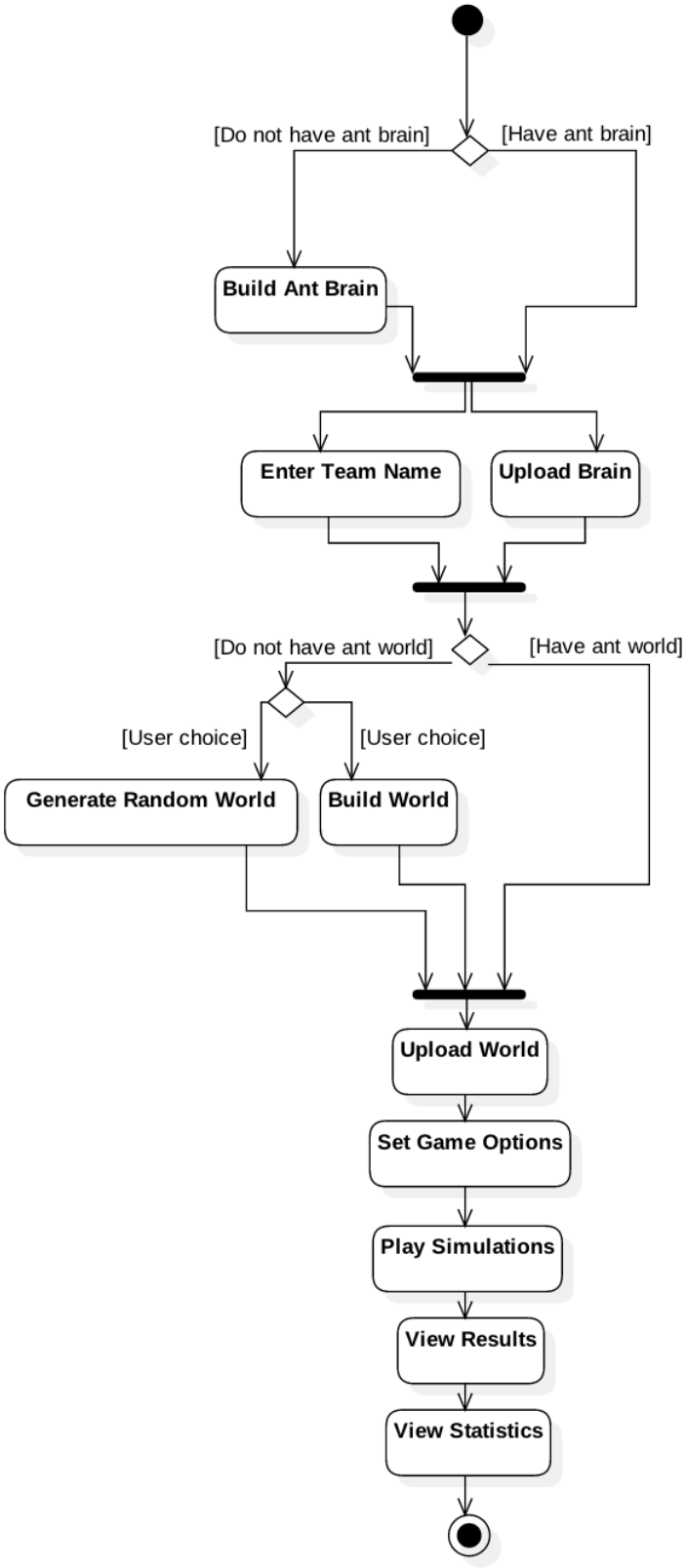
As with many computer games, this system is mostly 'self-enclosed' - that is, it does not require a large amount of interaction with separately-developed external systems. Thus, the context of the Ant Game system is minimal. However, unlike many other computer games, the style of the game can require the user to provide some external 'data' before playing (an ant brain, and sometimes an ant-world). This means that there is a possibility for external programs to be developed to aid the user in writing a custom ant brain/world. This would mean that there would be an interaction between the Ant Game program and two other systems, as below:

However, as per the functional requirements document (`Game/4` and `World/1`) the program shall parse ant brains and worlds from a *file* (i.e. only indirectly received from outside software), so this interaction with external programs does not need to be accounted for in development.

1.2 Process Model

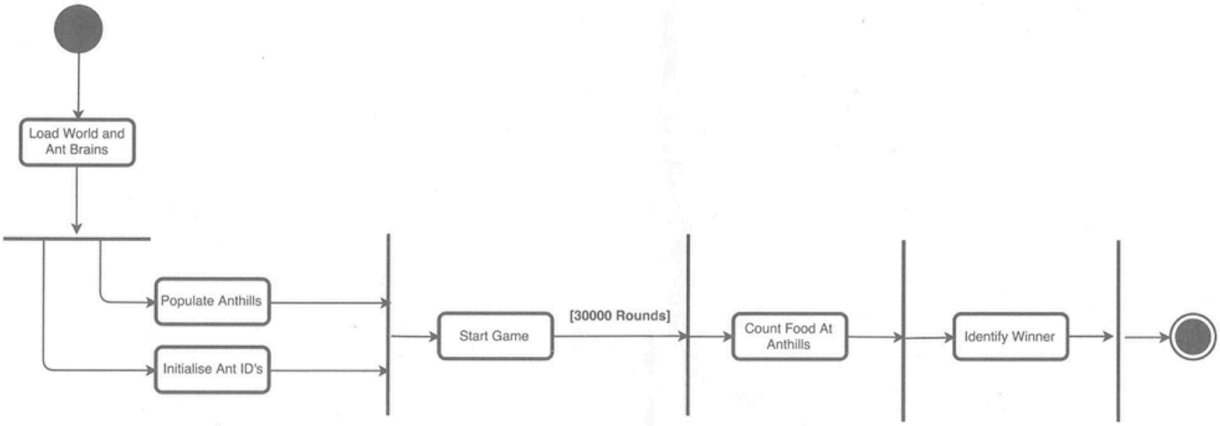
1.2.1 User Process

The process model on the following page describes a player interacting with the ant game system. Note that the only two actions where the user is not directly interacting with the program is the `Build Ant Brain` action and `Build World` action - both of these actions are done independently of this system by the user (in a text editor or otherwise). This process model is abstract enough that it applies to both the two-player game and the tournament. It is assumed that at least two users are following this process model at the same time, and interacting with the same instance of the program.


[Do not have ant brain]
[Have ant brain]
Build Ant Brain
Enter Team Name
Upload Brain
[Do not have ant world]
[Have ant world]
[User choice]
[User choice]
Generate Random World
Build World
Upload World
Set Game Options
Play Simulations
View Results
View Statistics

1.2.2 Game Process

The following diagram gives a high-level view of the flow of the system when simulating a game between two ant colonies on a particular world. Ant brains and a single world are loaded by two actors (the players). The system then initialises ants on the two anthills for each colony. Each ants identifiers are also initialised, as per the order in the functional requirements. The game is then started, lasting for 300,000 rounds. Once the round counter is at its maximum, the food at each anthill is counted; the team with the most food at their brains anthill is declared the winner.



1.3 Use Cases

The process model makes it clearer as to how the user will interact with the system. From the process model and the requirements, the primary use cases of the system have been extracted, as below. The only (human) actors in the following use cases are the players of the game.

1.3.1 Upload Ant Brain

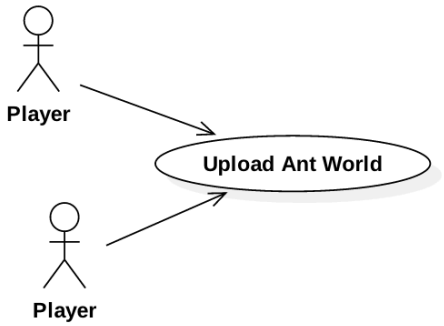
	Ant Game: Upload Ant Brain
Actors	Each player in a two-player game or a tournament. That is, although not interacting with each user simultaneously, the system will be interacting with multiple users in turn.

Description	In order to simulate a game, the players must upload their custom ant brains. These must be parsed from file into memory.
Data	The location of a file on disk.
Stimulus	Triggered by user clicking on a file-chooser button, navigating to the path of the file, and clicking 'Parse' (on the game setup screen of the system's interface).
Response	Whether the file represented a valid ant brain as per the functional requirements for parsing given in `Parsing Specifications/Brain`. If invalid, the system should inform the user of what went wrong and on which line (if relevant).

1.3.2 Generate Random World

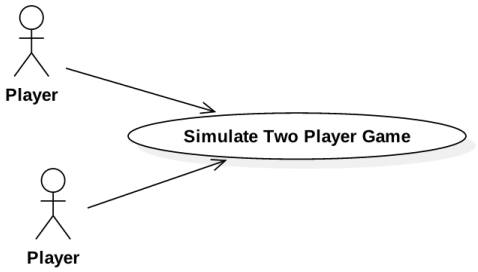
	Ant Game: Generate Random World
Actors	A player, before a game or tournament. Filesystem.
Description	A tournament requires a world that conforms to the specifications laid out in the functional requirements (see requirement `World/2`). This component of the system can be interacted with by the user to write a random world to file which conforms to these specifications.
Data	-
Stimulus	Triggered by the user by interacting with the GUI, selecting the option to generate a random ant world.
Response	Generates a random ant world. Converts the ant world to a textual description, as per functional requirement `Parsing Specifications/World`. Asks the user where to save to on disk, and then writes to a text file at that location.

1.3.3 Upload Ant World



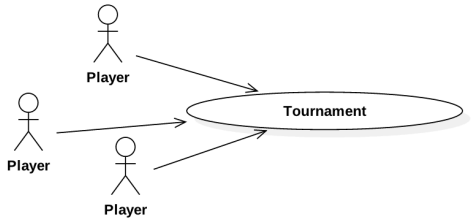
	Ant Game: Upload Ant World
Actors	Two or more players who agree to upload a particular ant world to compete on.
Description	In order to simulate a game or tournament, there must be at least one ant world for the ants to compete on. These must be parsed from file into memory.
Data	The location of a file on disk.
Stimulus	On the game and tournament setup screens, there will be a button to upload an ant world. Once the user clicks on this, chooses a file to parse, and clicks 'Parse', this use case will be invoked.
Response	Whether the file represented a valid ant world as per the functional requirements for parsing given in `Parsing Specifications/World`. In addition, if this use case is invoked in the context of a tournament setup, the additional criteria for a contest world shall be checked (see requirement `World/2`).

1.3.4 Simulate Two Player Game



	Ant Game: Simulate Two Player Game
Actors	Two players.
Description	Once the game setup is complete (with players setting the ant brains and ant world) this use case comes next, with a competition between the ants on the world being simulated.
Data	The contextual data from other use cases will be in memory - the world and brains. Some other relevant data shall be provided by the user, such as the team names and options for game viewing (detailed in the interface design section).
Stimulus	The 'Play' button on the game setup screen.
Response	The system will simulate a game between the two ant colonies as described in the functional requirements (`Game/1`). It will present a simulation of the world as the ants carry out their actions. At the end of the simulation, it will display the winner of the game to the players, as well as relevant game statistics.

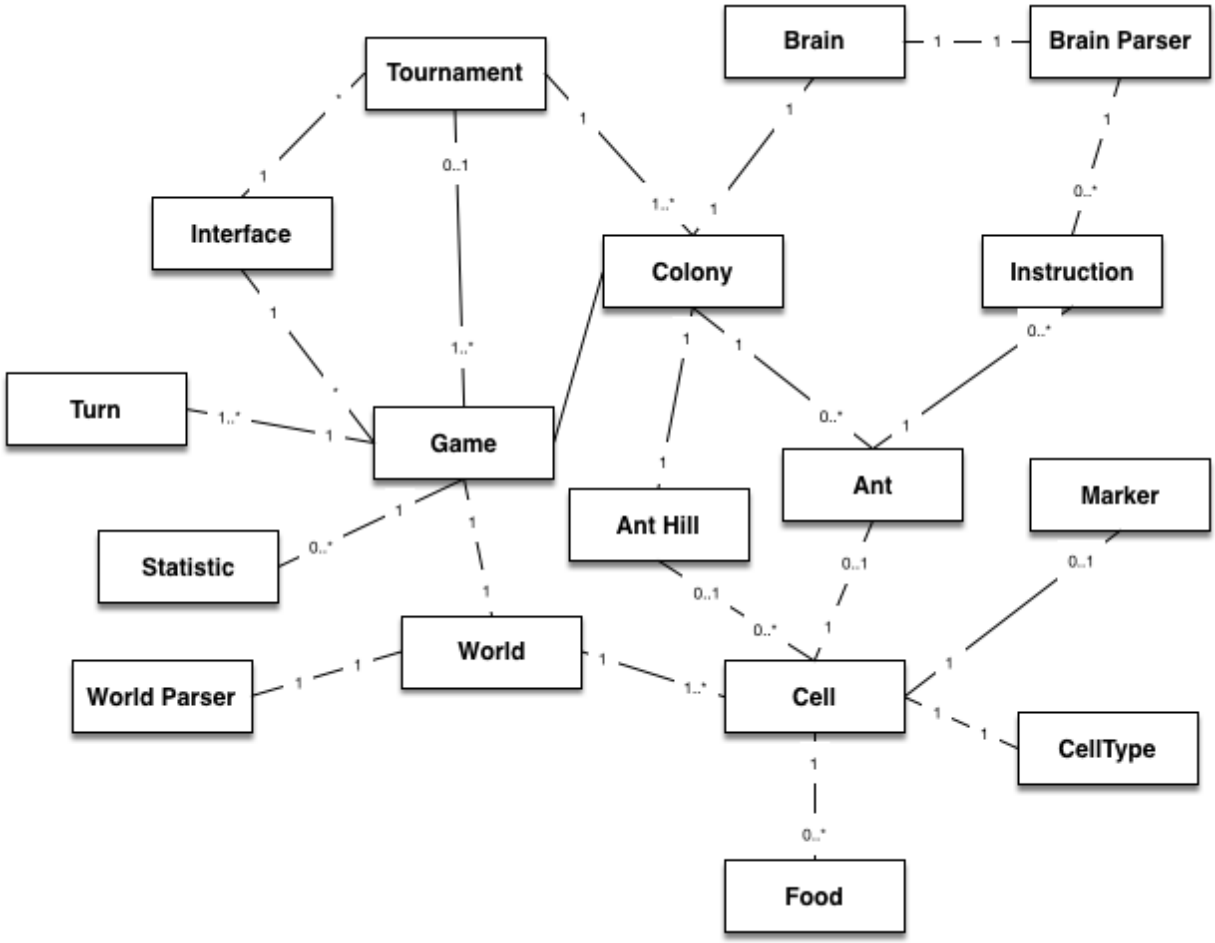
1.3.5 Tournament



	Ant Game: Tournament
Actors	Two or more players.
Description	This use case can be considered a 'superset' of the use Two Player Game use case. Once each of the players have uploaded the ant brains and ant worlds, they can start a tournament to compete amongst multiple ant brains on multiple ant worlds, to see which player is the overall winner.
Data	The contextual data from other use cases will be in memory - the worlds (at least one, possibly several) and brains. As above, some other relevant data shall be provided by the user, such as the team names and options for game viewing.
Stimulus	The 'Play' button on the tournament setup screen.
Response	The system simulates an ant game between every possible combination of teams, worlds, and colours (red or black). Then it returns the final results of the tournament to the user (as per the functional requirement `Game/3`) as well as statistics on the game.

1.4 System Components

From the functional requirements, the process model and use cases, the main system components and their interactions have been modelled below. Note also the multiplicity between the components.



The system components diagram is extremely useful in terms of developing a full design, and is the fundamental building block of the low-level design section.

2 Low-Level

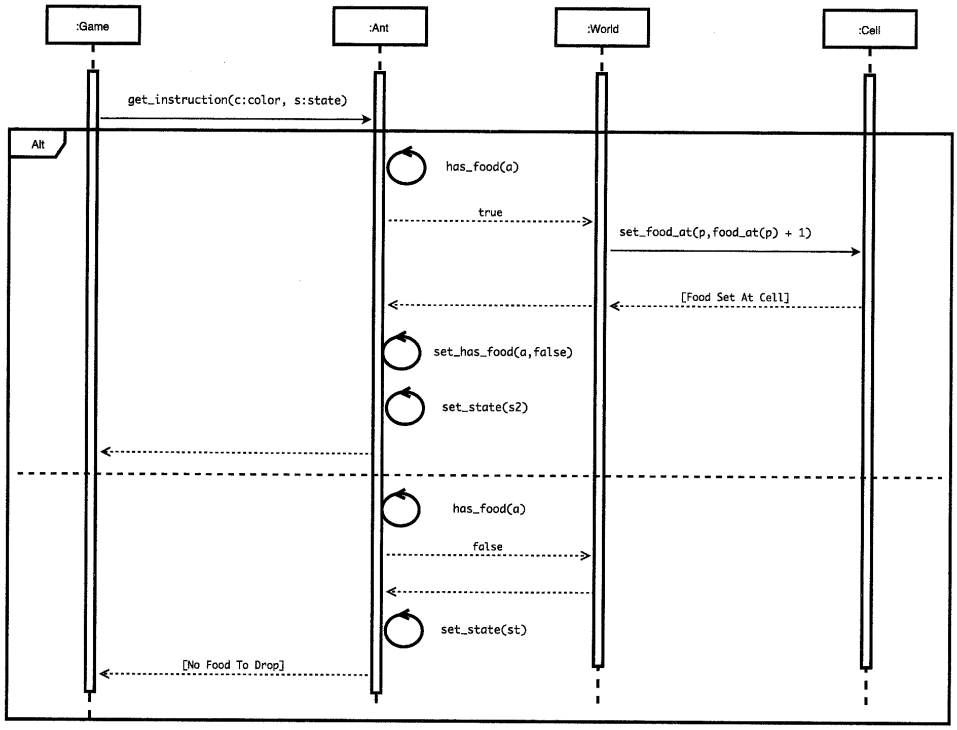
The low-level section of the design goes into further detail of how the system is to be implemented. The lower level design has been done is a slightly non-conventional order — instead of beginning from the system component diagram to develop detailed class diagrams, the initial point of the low-level design has started from the use cases to construct detailed sequence diagrams. From the sequence diagrams, class diagrams have been constructed to validate the design.

2.1 Sequence Diagrams

The following sequence diagrams detail, in particular, the behaviour of an ant as it interacts with the world in a game simulation. A single 'operation' is described in each sequence, and clearly exposes the relevant components and behaviours associated with the operation.

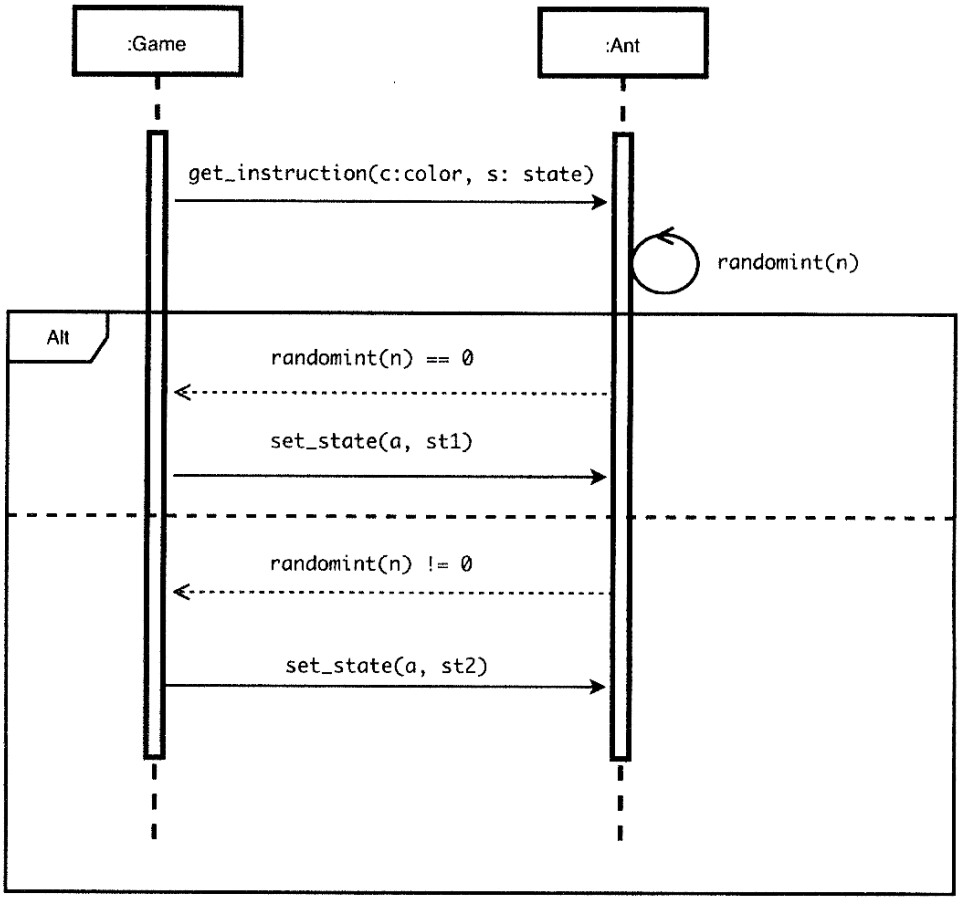
2.1.1 Ant - Drop

This sequence diagram describes the actions of an ant when performing a *drop* instruction.



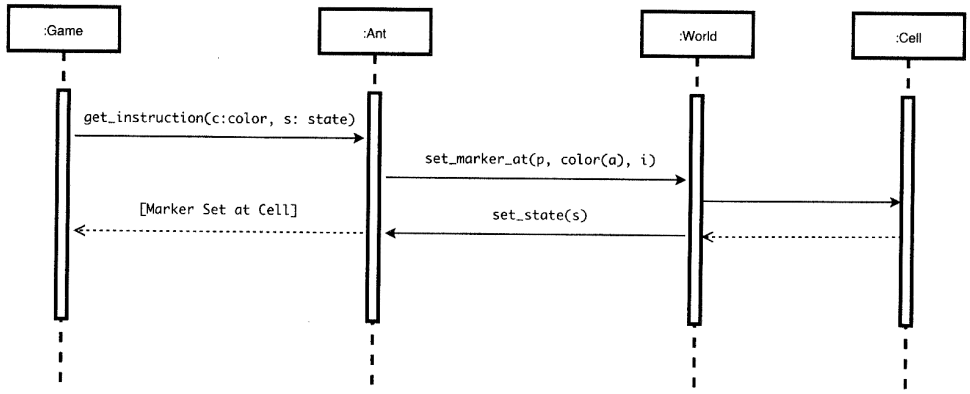
2.1.2 Ant - Flip

This sequence diagram describes the actions of an ant when performing a *flip* instruction.



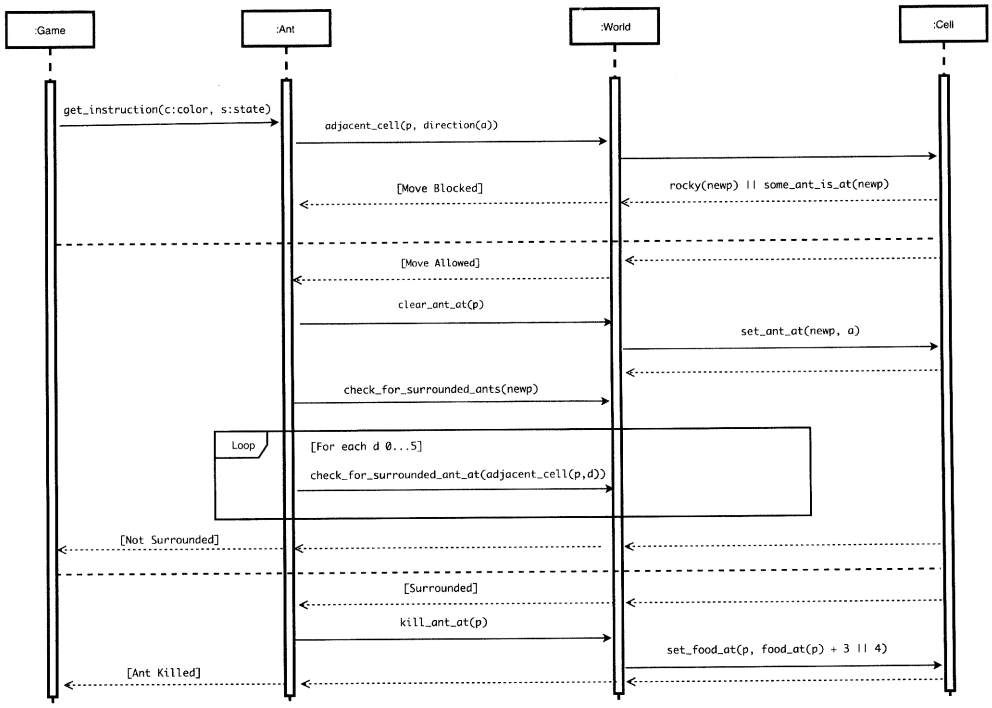
2.1.3 Ant - Mark

This sequence diagram describes the actions of an ant when performing a *mark* instruction.



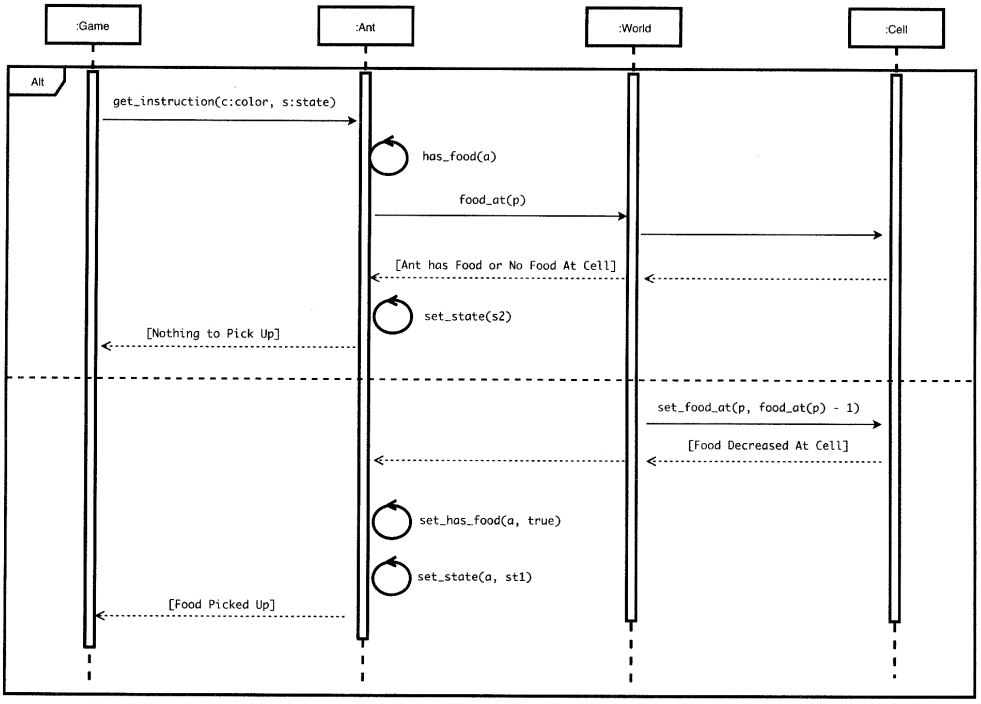
2.1.4 Ant - Move

This sequence diagram describes the actions of an ant when performing a *move* instruction.



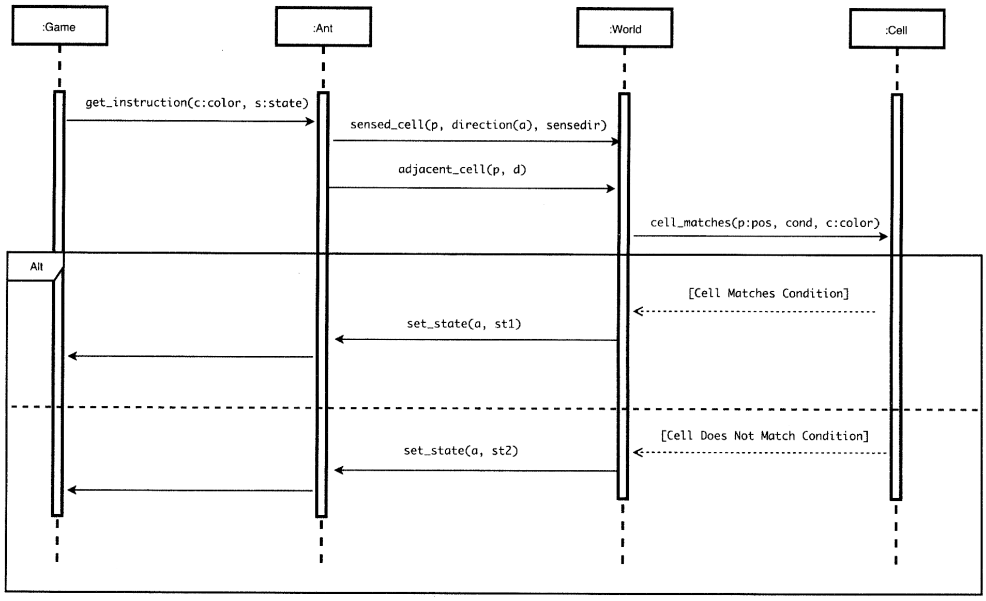
2.1.5 Ant - Pick-Up

This sequence diagram describes the actions of an ant when performing a *pick-up* instruction.



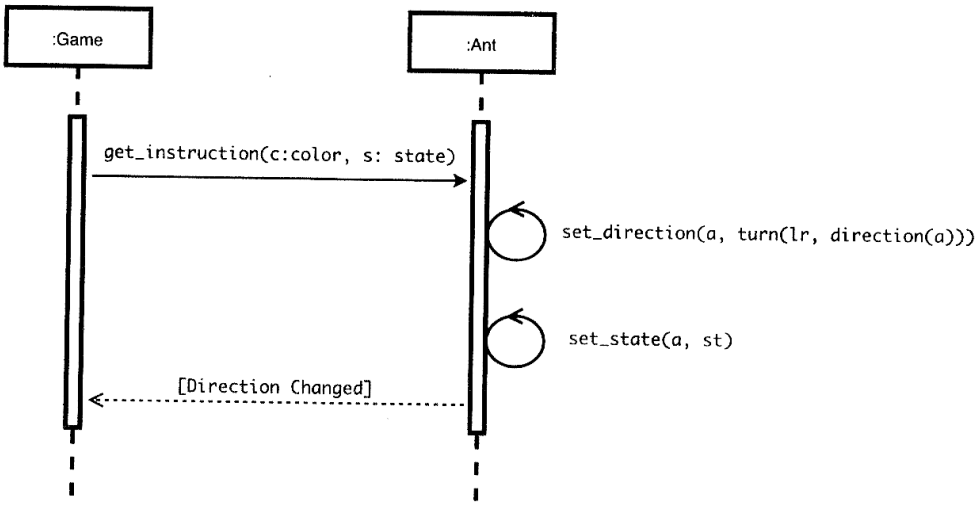
2.1.6 Ant - Sense

This sequence diagram describes the actions of an ant when performing a *sense* instruction.



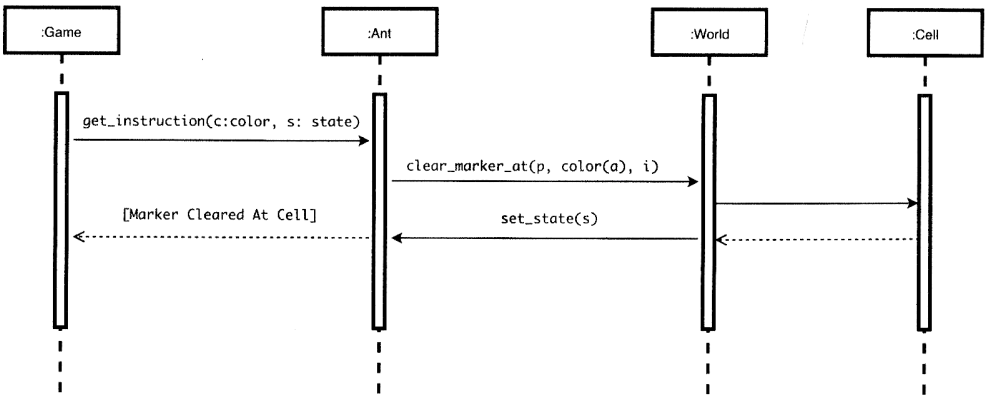
2.1.7 Ant - Turn

This sequence diagram describes the actions of an ant when performing a *turn* instruction.



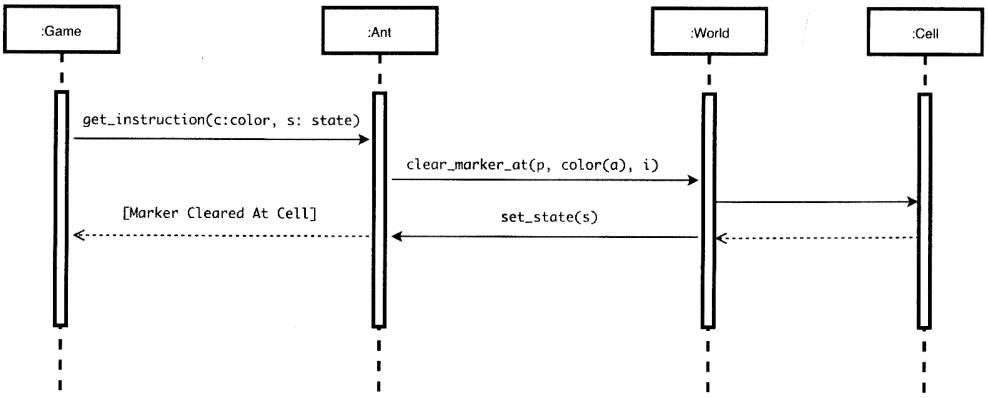
2.1.8 Ant - Unmark

This sequence diagram describes the actions of an ant when performing an *unmark* instruction.



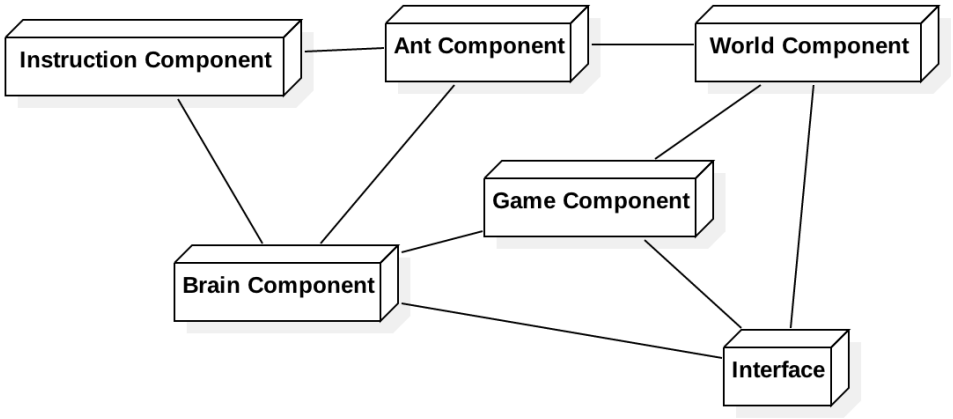
2.1.9 World - Execute Instruction

This sequence diagram describes the actions of the world when interacting with the ant to execute an instruction.



2.2 Class Diagrams

The below diagram is an overview of the components which are described in much more detail in this section. Each of the components below is a 'module of functionality'. The classes inside each module are highly related, whereas the interaction between the classes of separate modules have minimal coupling.

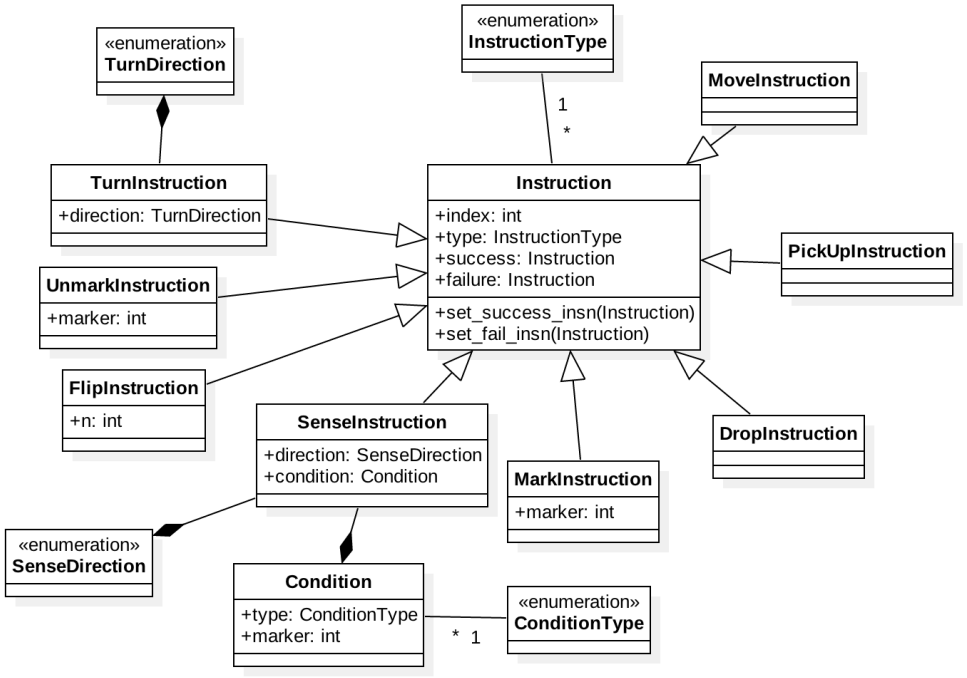


The class diagrams in this section have omitted getters and setters for brevity. After each diagram is a brief description of the purpose of each class involved.

2.2.1 Instructions

The following diagram abstracts the different type of *instructions* that an ant brain can carry out. This component of the system follows a linked-list style paradigm - an `Instruction` has two self-

referential attributes `success` and `failure`, which are the states that the ant brain will transition to depending on whether the ant's action was a success or a failure. Observe that the majority of classes below have little to no operations, only attributes — this is because the majority of state changes will be handled by other components, whereas this component is primarily for holding data. All classes are immutable, apart from setting the transition instructions.

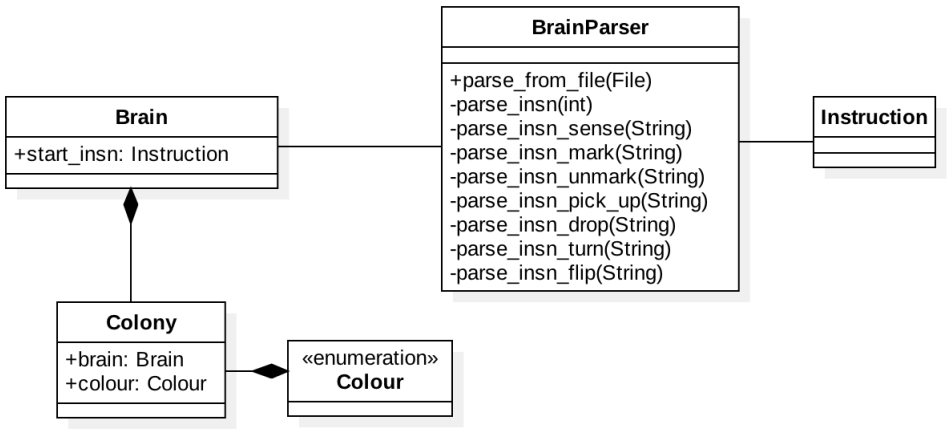


	Ant Game: Instructions
`TurnDirection`	An `enum` that describes the directions available for an ant to turn in - `left` or `right` as per functional requirement `Ant/6`.
`TurnInstruction`	Represents an instruction that orders an ant to turn in a particular direction. The transition success state and fail state are both the same state.
`UnmarkInstruction`	Represents an instruction that orders an ant to unmark a cell that holds a marker (of the same colony) of a given integer identifier. The transition success state and fail state are both the same state.
`FlipInstruction`	Represents an instruction to generate a random number in the range `0..n`.
`SenseDirection`	An `enum` that describes the directions available for an ant to sense in - `left-ahead`, `ahead`, and `right-ahead`.
`Condition`	A class which represents a sense condition for an ant to validate on a cell. A condition has a type, and if it is a sense condition, also a marker identifier.
`ConditionType`	An `enum` that represents possible conditions for an ant to check on a cell - see requirement `Ant/1`.

`SenseInstruction`	A class which represents an order for an ant to check a condition on a cell in a specific direction.
`MarkInstruction`	A class which represents an order for an ant to mark a cell. **Note**: the design specification for the `marker` field has been slightly modified - instead of an integer, the `MarkInstruction` class will use an instance of a `Marker` class, which holds an integer marker and *also* an associated colony colour (red or black).
`DropInstruction`	A class which represents an order for an ant to drop its food. The success and failure states for this class are the same state.
`PickUpInstruction`	A class which represents an order for an ant to pick up the food on its current cell.
`MoveInstruction`	A class which represents an order for an ant to step forward in its current direction.
`InstructionType`	An enumeration which holds all possible instruction types - e.g. `DropInstruction`.
`Instruction`	Finally, the abstract base class of all above instruction types — this class stores information which is shared amongst its subclasses. Each instruction has an `index` (the line number, 0-based), a type (holding the type of subclass), and two recursive instances of `Instruction` which hold the state to transition to if the ant's evaluation of this instruction is a success, and the state to transition to if the ant's evaluation of this instruction fails. For several subclasses, these are the same state.

2.2.2 Brain

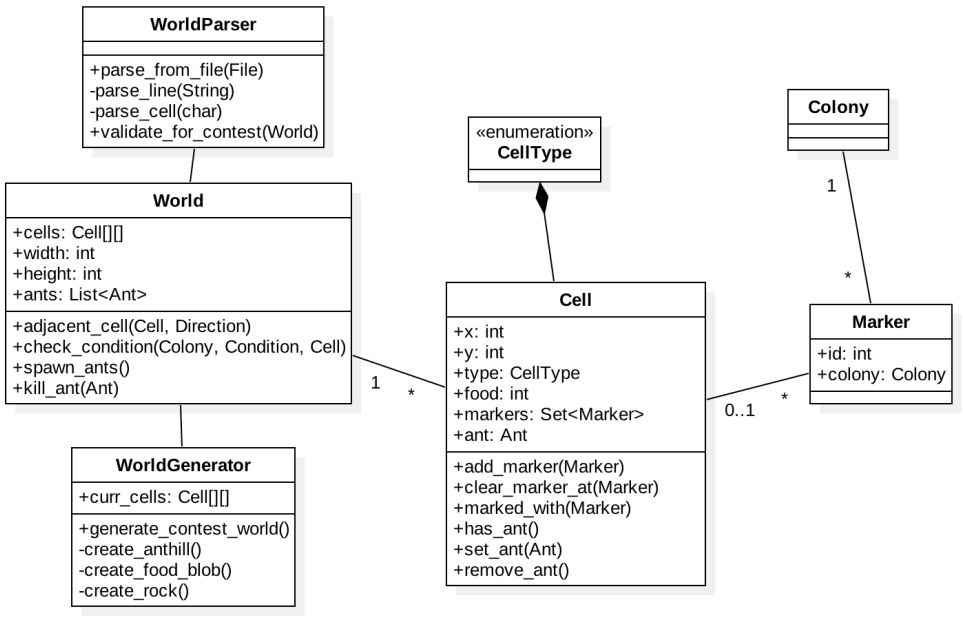
Generating the above instruction data is done by the *Brain Parser* component. As can be seen in the brain class diagram, a colony is composed of a colour and a brain - **note**: a colony of ants shares the same brain - an ant will, upon instantiation, get the start-instruction from the brain, and then keep track of its own state from then on. The brain parser exposes only one public method (to parse an ant brain file). The rest of the methods are to convert a string representation of the given type into the programmatic representation as per the instruction class diagram.



	Ant Game: Brain
`Instruction`	This is a reference to the `Instruction` class in the previous diagram.
`BrainParser`	This class has the purpose operation of translating a string-based representation of a brain graph constructed from the subclasses of `Instruction`. The `parse_from_file(File)` method should also validate the syntax of the brain specification. The internal design of the class is *deliberately left unspecified*: it is left up to the programmers - either by stepping through the file line by line and connecting up the graph afterwards, or recursively parsing instructions based on the transition states that each line references.
`Brain`	This class represents a unified brain of a colony. All ants share the same brain, but they do not share the same brain state - i.e. they keep track of the current state based on the return value of `Instruction#success()`/`Instruction#failure()`.
`Colony`	As mentioned directly above, ants of the same colony share the same brain. So an instance of the brain is contained in the colony representation. A colony also has an associated colour, as below.
`Colour`	Holds the possible colours of a colony - either red or black.

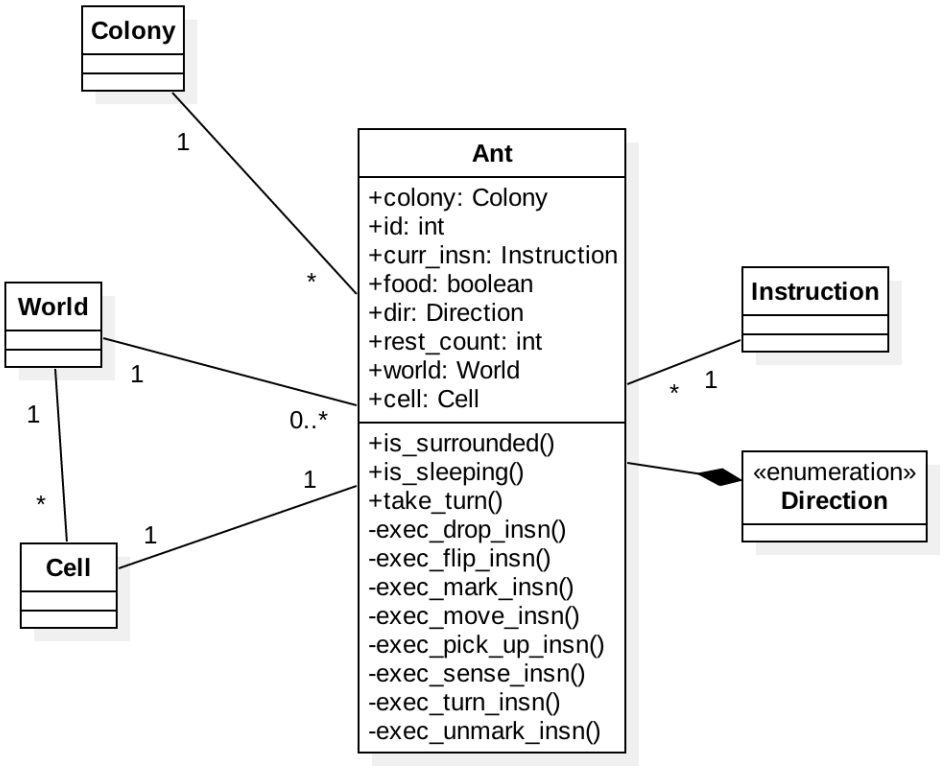
2.2.3 World

The next component of the system is the world-related content. The world parser reads a world specification from a file and produces a world instance. A world-generator instance can create a world instance via adding random contest-valid cells. A world is made up of an array of cells, which can have markers added and removed from them — notice that a marker instance has an associated colony. In addition a cell can also hold an ant, which the world initialises via spawning the ants at the start of a game.



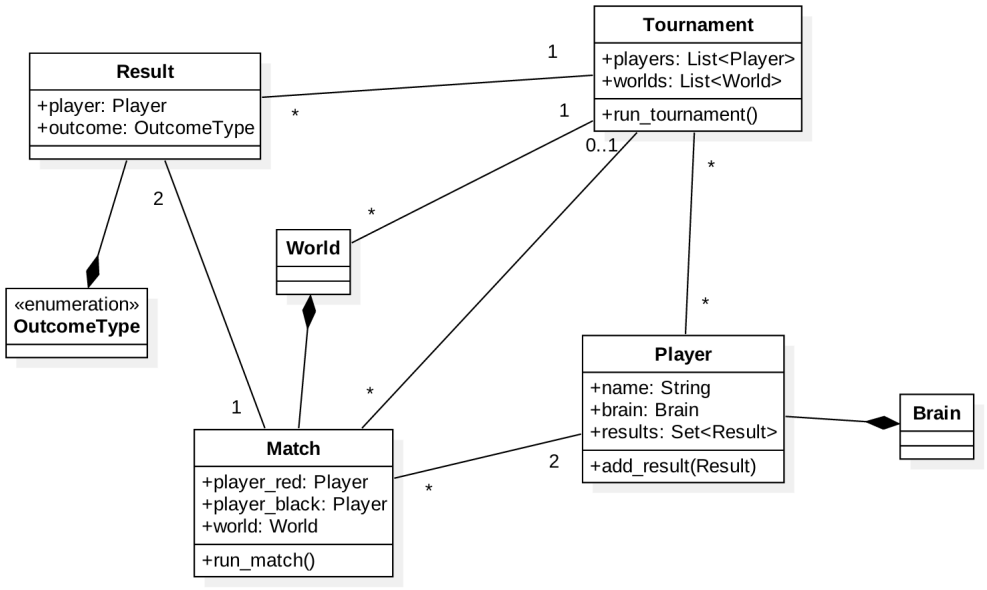
	Ant Game: World
`WorldParser`	Has the purpose of reading a world specification from a file. **Note**: additional operation requirement — this class needs to expose a method `validate_for_contest(World)` which checks that a given world conforms to the tournament world specifications.
`World`	Represents a game world and its associated state - that is, the world stores all cells and what they contain (food, markers, ants, rocks, ...) at any point. Exposes functionality to spawn and kill ants, check conditions on cells, find the adjacent hexagonal cell given a cell and direction.
`WorldGenerator`	Used for generating random game worlds which conform to the tournament world specifications.
`Cell`	The world is made up by a two-dimensional array of cells. Each cell contains information about what it holds - markers, ants, food particles. A cell instance exposes functionality for changing its state by adding and removing markers, ants, food.
`CellType`	An enumerated type which holds all the different possible cells as per requirements - rock, clear, red anthill, black anthill.
`Marker`	Notice that each marker holds not only the integer marker identifier, but also the associated colony - this is because cells store each colony's markers separately.
`Colony`	As per previous table.

2.2.4 Ant



	Ant Game: Ant
`Ant`	This is where the main logic for changing the state of the game world happens. An ant stores its own state (meaning the next instruction to execute on the world, as well as the unique ant identifier, the colony of the ant, whether the ant is holding food, the current direction in which it is facing, how many turns it is resting for — 0 if not resting). It also has access to its current cell and also the world. The ant exposes functionality to check if it is surrounded by 5 or more foe ants, to test whether it is currently resting. Most importantly, it exposes the `take_turn()` method, which executes an instruction. This method delegates to the relevant private method based on the instruction type. Each of these methods use the local state of the world to change the world's state, as well as the ants own state.
`Direction`	This enumeration holds all possible hexagonal directions, as per the relevant section of the functional requirements.

2.2.5 Game

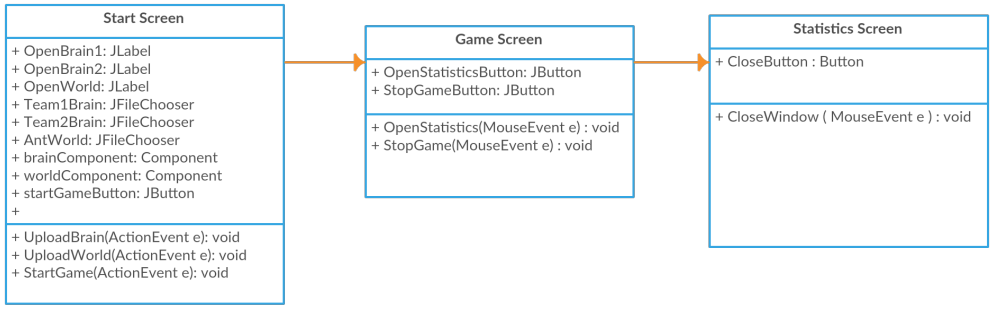


	Ant Game: Game
`Result`	A result is the outcome *for a particular player* in a match. That is, for each result, there are three possible sets of two results: {(player 1 win, player 2 loss), (player 2 win, player 1 loss), (player 1 draw, player 2 draw)}.
`OutcomeType`	The possible outcomes for a player in a match - win, loss, draw.
`Match`	A match describes a pairing between two players on a world. One player is tied to the red colony of ants, and another player is tied to the black colony. The result of a match (`run_match()`) is a set of two outcomes, one for each player. The job of the `run_match()` method is to iterate 300,000 times over all the ants, making them take steps and then also checking to see whether the ants die from being surrounded. The ant colony with the most food in the anthill is then declared the winner.
`Player`	A player essentially a programmatic model of an actual user. A user uploads an ant brain from the interface, and specifies a team name. They then can compete in either a tournament or a match. A player maintains a set of results, possibly from a single match or a tournament.
`Tournament`	A tournament can be treated as a collection of all possible matches as per functional requirements (see `Game/3`). The tournament simply creates matches between the players, and then executes them. Score counting is done as per requirement `Game/3`, and so is team elimination if drawing occurs.

Note: further to the above, an additional integration of the game with the world must be added to the current design. As per requirement `Game/2`, statistics must be produced for a game. This involves:

- integrating statistics tracking into the `Match#run_match()` method — by examining the state changes occurring to the world after each match iteration, increment relevant local variables to track the statistics specified in `Game/2`.
- the tournament statistics should track the *total* statistics over all games.
- these statistics should be accessible (by return value) to the interface.

2.2.6 Interface



Start Screen: the start screen is where the program initialises.