# EECS3311-W20 — Project Report

Submitted electronically by:

| **Team members** | **Name** | **Prism Login** | **Signature** |
|---|---|---|---|
| Member 1: | Joseph Emanuele | joeycs | J. E |
| Member 2: | Tamal Alexander | tamal | T. A |
| *Submitted under Prism account: | | joeycs | J. E |

**Contents**

## 1. Requirements for Project - Simodyssey

Our clients supplied us with the following specifications for Sim Odyssey: the desired product is a space exploration game which will allow the player to choose between two playing modes. The first mode is referred to as “test”. This mode provides a more detailed description of the current state of the galaxy and entities which are generated within it. The proportions of each type of entity generated are determined by thresholds specified by the user. A higher threshold corresponding to a given entity type implies this type of entity has a higher probability of appearing in the galaxy. The second mode is referred to as “play”. This mode does not generate the additional output of test mode as described above, and the thresholds are fixed at default values.

When either mode is chosen, random numbers are generated based on a predetermined seed, guaranteeing identical outputs from sequences of commands which are repeated from one game to another. After a game has started, a 5-by-5 galaxy is created. In the galaxy, an explorer entity at quadrant (1, 1) and a blackhole entity at quadrant (3, 3) are placed. The placement of non-explorer entities, i.e. stationary entities and computer controlled movable entities, are again decided by a random number generator. Specifically, these entities are placed in sectors within the galaxy; each sector has four quadrants and thus can contain a maximum of four entities.

After the galaxy is created, the player can issue commands to move the explorer throughout the galaxy. Some of these commands cause a turn to occur, resulting in the movement of movable entities. In some cases which are dependent on proximity to other types of entities, these entities perform secondary actions such as attacking other entities or reproducing. However, other commands do not advance the game state such as the command which displays the current status of the explorer, including its life and fuel levels. The game can end in a few ways. The player can lose the game if the explorer runs out of fuel, life, or is consumed by a blackhole. The player can win the game by landing on a planet which supports life. The player can also simply abort the game, neither winning nor losing. Once a game has ended, a new game can be started.

The appendix of this report will provide further details on the user interface grammar of Sim Odyssey. Also, the acceptance tests later outlined will provide further details on the software’s expected behaviour based on varying user input.

## 2. BON class diagram overview (architecture of the design)

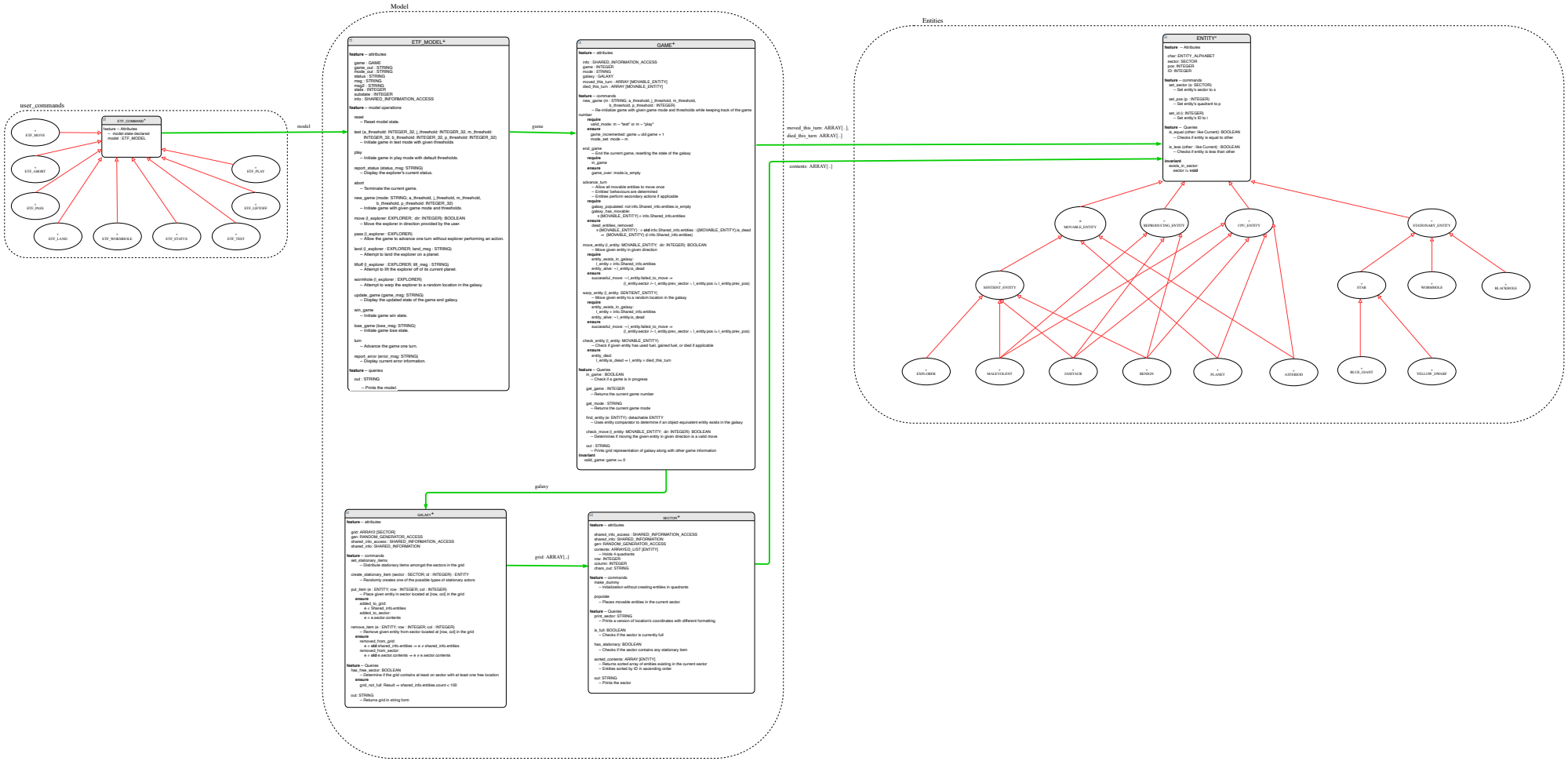


***Figure 1.*** *The BON diagram representing the Sim Odyssey software design*

At a high level of abstraction, The design of Sim Odyssey consists of two main components: the model cluster and the entities cluster. The model cluster contains classes pertinent to the game itself. These classes manipulate the state of the galaxy, the grid in which the game is facilitated, and produce output which provides useful visual feedback to the player regarding the state of the game. Specifically, ETF_MODEL uses STRING attributes to update the game's grid representation and messages while accepting a variety of user commands, GAME keeps track of the behaviour of entities and outputs the state of the galaxy, and the GALAXY and SECTOR classes organize entities into data structures in order to inform the GAME class about where entities should be displayed and how they should interact with each other.

The entities cluster contains a hierarchy of classes related by inheritance which classes in the model cluster instantiate as objects to compose the contents of the galaxy and its sectors. The purpose of this hierarchical organization of classes is threefold: firstly, the design takes advantage of code reuse by providing different types of entities, which can be grouped according to their required functionalities, with common features and attributes existing in deferred parent classes. In this way, the single choice principle is satisfied. Second, the design grants simplicity when implementing features in which it is required that only particular types of entities be manipulated. With the use of dynamic binding, it is possible to check if a context entity inherits from a particular parent class, providing access to features and attributes specific to this type and its children. This functionality allows for elegant implementations of features which can be applied to any child of the entity class, preventing the need for additional classes or features in the model cluster which are specifically implemented to operate on different types of entities. Finally, by taking advantage of multiple inheritance, the hierarchy is expandable, allowing for the addition of entity subclasses inheriting from a combination of existing or new classes depending on the new entity's desired behaviour.

To justify the design decisions discussed above, the following scenario will be considered: a programmer wants to create a new type of entity called SWARM, a fleet of insects who reproduce at a high rate and destroy any forms of life in their path. This new class could simply inherit from REPRODUCING_ENTITY to make use of code which creates clones, CPU_ENTITY to be considered in features which use random number generation to generate behaviour, and MOVABLE_ENTITY to allow it access to features and attributes used for movement. Now, all the programmer must do is implement any deferred features such as set_behaviour, inherited from CPU_ENTITY, and their new entity should act as desired. The programmer can take advantage of existing logic structures in the model cluster to generate SWARM objects' placement in sectors, movement throughout the galaxy, and interaction with other entities.

## 3. Table of modules — responsibilities and information hiding

Module one (1) consists of the ETF_MODEL, GALAXY, SECTOR and GAME classes:

| | | | |
|---|---|---|---|
| 1 | ETF_MODEL | **Responsibility**: Handles all the user commands for the game | **Alternative**: Implement several classes responsible for handling different user commands; breaks single choice principle by removing the possibility of code reuse |
| | Concrete | **Secret**: Implemented by initializing each attribute to a default value | |

| | | | |
|---|---|---|---|
| 1.1 | GAME | **Responsibility**: see ETF_MODEL | **Alternative**: Separate GAME into multiple classes responsible for different entity behaviours |
| | Concrete | **Secret:** Implemented via two arrays: entities that moved this turn and entities that died this turn | |

| | | | |
|---|---|---|---|
| 1.1.1 | GALAXY | **Responsibility**: see GAME | **Alternative**: Make class iterable to allow simpler traversal of sectors within the galaxy via across loops |
| | Concrete | **Secret:** Implemented via two-dimensional array which contains the galaxy's sectors | |

| | | | |
|---|---|---|---|
| 1.1.1.1 | SECTOR | **Responsibility**: see GALAXY | **Alternative**: Implement via a hash table where each entity in the sector is mapped to its quadrant, rather than each entity holding a quadrant attribute |
| | Concrete | **Secret:** Implemented via an arrayed list which contains the sector's contents | |

Module two (2) consists of the SHARED INFORMATION, SHARED_INFORMATION_ACCESS and ENTITY COMPARATOR classes:

| | | | |
|---|---|---|---|
| 2 | SHARED_INFORMATION | **Responsibility**: Contains attributes which are used to generate stationary and movable entities | **Alternative**: Implement throughout different classes; violates the design principle of cohesion |
| | Concrete | **Secret:** Implemented via linked list that stores entities which exist in the galaxy | |

| | | | |
|---|---|---|---|
| 2.1 | SHARED_INFORMATION _ACCESS | **Responsibility**: Provides singleton access to shared information | **Alternative**: Provide multiple instance access to shared information; violates single choice principle |
| | Abstract | **Secret:** none | |

| | | | |
|---|---|---|---|
| 2.2 | ENTITY_COMPARATOR | **Responsibility**: Compares entities based on their id | **Alternative**: Implement via insertion sort instead of quick sort; more space efficient, less time efficient |
| | Concrete | **Secret**: none | |

***Figure 2.*** *The modules which Sim Odyssey's software is composed of*

## 4. Expanded description of design decisions

The module which will be discussed in detail is the GAME class. It is the most important module in the design of Sim Odyssey as it is responsible for orchestrating the components of the GALAXY class, SECTOR class, and entities hierarchy to produce the software's gameplay.

The interface of the GAME class consists of features such as new_game, end_game, in_game, get_game, and get_mode which are useful for checking and manipulating the state of the current game at a highly abstract level. These features are only concerned with information including the number of games played so far, the current game mode, and whether a game is currently in progress by using INTEGERS and STRINGS to maintain basic data.

The next group of features found in GAME are responsible for the state of the current game at a lower level and include advance_turn, move_entity, warp_entity, check_entity, find_entity, and the hidden feature move_helper. These features create and mutate the galaxy (specifically, its grid and the entities contained within it), and arrays used for keeping track of which entities have exhibited certain behaviour in a given turn.

The feature advance_turn iterates across an array of entities which currently exist in the grid, using an algorithm which encompasses all movable entities to determine their behaviour during a given turn. This feature also uses type-checking to retrieve and mutate attributes specific to certain types of entities to dictate the behaviour of, for instance, an entity which is movable but also reproducing. An entity with these characteristics utilizes the same code used by all other movable entities to move, as well as accesses the same code all other reproducing entities use to reproduce. The design of this feature provides maintainability such that the addition of a new entity class which inherits from a combination of existing entity types requires little to no expansion of its content. This characteristic also makes the feature reliable, as the reuse of code used for deciding the behaviour of entities drastically reduces the requirement of debugging newly introduced code. Moreover, the single choice principle is satisfied because the alteration of, for example, the number or frequency of clones made by REPRODUCING_ENTITIES, requires only a single change in the deferred parent class' implementation.

Similarly to advance_turn, check_entity employs a strategy of determining the dynamic type of an entity object which is statically parameterized as the deferred parent class MOVABLE_ENTITY. It performs these checks to set the death and fuel statuses of the parameterized entity whenever applicable. The features move_entity, warp_entity, and find_entity also take advantage of the entity classes' hierarchical inheritance to cause entities

to move by manipulating the common attributes which determine their position in their sector and the galaxy, or simply return the entity object if it is found in the galaxy, in the case of find_entity. All features listed above are accessed by the advance_turn feature, hiding them from clients of the GAME class in many cases, as a single call to advance_turn results in all calls to these features which are necessary to produce the outcomes of one game turn. Finally, the hidden move_helper feature allows move_entity and warp_entity to reuse code for moving an entity from one location in the galaxy to another, whether it is in a specified direction in the case of move_entity or to a random location in the case of warp_entity. Additionally, the single choice principle is satisfied, since a change to the implementation of entity movement in the case of regular movement or warping only requires a change in the move_helper feature.

An alternative design of the GAME module which was previously suggested involves compartmentalizing some of the features discussed in the previous paragraph into classes separate from the GAME class. While the motivation behind this potential design decision was partly sound in that creating classes dedicated to moving entities, checking entities, etc. would benefit cohesion in the overall design of the software, it results in too large of tradeoffs in the form of simplicity and maintainability. This design would remove the possibility of utilizing the full potential of the entity inheritance hierarchy. Specifically, the arrays required for maintaining entities who have moved in a given turn and died in a given turn would need to be accessible by all of these classes. This functionality could only be achieved by either passing the arrays from class to class, which convolutes code for both designers and clients of the GAME class, or placing them in the SHARED_INFORMATION module, breaking cohesion in a different way by keeping data pertaining specifically to the occurrences of a game turn outside of the class which keeps all other data related to a game turn.

## 5. Significant Contracts (Correctness)

Contracts included in the GAME class and their significance will be described. This class contains many features vital to the functionality of Sim Odyssey; their contracts are equally important, as they ensure GAME is being used correctly by clients and producing the correct output.

The advance_turn feature contains two preconditions. The galaxy_populated precondition requires that the galaxy contains at least one entity upon the call of advance_turn. The purpose of this precondition is to prevent misuse of advance_turn through attempting to advance the game a turn before the game has properly started, i.e. the galaxy and its sectors have been created and populated. Similarly, the galaxy_has_movable precondition, once it has been determined that some entity exists in the galaxy, requires that one of said entities is a MOVABLE_ENTITY. Since advance_turn only operates on movable entities, this feature would simply do nothing if one did not exist in the galaxy. Without a precondition violation and thus a lack of feedback, a client of GAME who doesn't know the implementation of advance_turn would not know why this feature, who promises to progress the game, did not change the state of the game. The advance_turn feature also has a single postcondition, dead_entities_removed. This postcondition ensures that entities who have died in the turn just passed are removed from the galaxy, and thus are not printed to the galaxy or list of entities when GAME's out feature is called next.

The feature check_entity has a single postcondition called entity_died. This postcondition ensures if an entity has died during the latest turn, then it has been added to the died_this_turn array and had its death message set. Therefore, the purpose of dead_entities_removed is to detect discrepancies between the set of entities which have died in a turn, and the set of entities displayed to the player in the "Deaths This Turn" section; these two sets are intended to be equivalent.

The feature move_entity contains two preconditions, entity_exists_in_galaxy and entity_alive. The precondition entity_exists_in_galaxy requires that the entity which the client would like to move is contained within the galaxy, preventing incorrect states in which a call to move_entity results in a new entity being added to or attempting to be removed from the galaxy. The precondition entity_alive requires that an entity is not dead before moving it to another location in the galaxy. While there are checks within advance_turn to ensure that entities being moved have not died, this precondition rules out scenarios in which the entity's is_dead attribute is set back to false after such a check and before attempting to move it. The move_entity feature also has a postcondition called successful_move which ensures that, in the case that the entity has declared it has performed a successful move, one of the

following has changed during the execution of move_entity: the row of the sector in which the entity exists, the column of the sector in which the entity exists, and the position or quadrant of the entity within its sector. This postcondition adheres to the definition of a successful move as one which causes the entity to have changed position at least at the sector level, i.e. changing its quadrant without changing its sector at minimum. The warp_entity feature contains identical contracts to move_entity, as it similarly utilizes the move_helper feature to change the location of the entity, but with different implementation due to different specifications.

The feature end_game contains a single precondition called in_game, which simply checks the value of the GAME class' BOOLEAN attribute of the same name, requiring that a game be in progress before attempting to put the game in an inactive state. The feature also contains a postcondition which ensures the STRING attribute called mode has been reset, or is empty, which is the way GAME signals that it is inactive.

The feature new_game has a precondition called valid_mode which requires that the user has provided a game mode input which is one of those supported, "test" and "play". This precondition prevents errors during output when the output is determined by which mode the game is being played in. This feature also contains postconditions called game_incremented and mode_set. The former postcondition ensures that the game count is incremented by exactly one when a new game is started. The latter precondition ensures that the mode attribute is set to the desired value; as alluded to earlier, this means that a subsequent call to in_game will return true.

## 6. Summary of Testing Procedures

The following table shows a variety of acceptance tests that were performed to satisfy the specifications given by the client:

| Test file | Description | Passed |
|---|---|---|
| *at001.txt* | When the explorer wins in play mode | ✓ |
| *at002.txt* | When the explorer wins in test mode | ✓ |
| *at003.txt* | When the explorer loses by running out of fuel | ✓ |
| *at004.txt* | When explorer passes through a wormhole and then tries to land | ✓ |
| *at005.txt* | When the explorer gets destroyed by an asteroid | ✓ |
| *at006.txt* | When the explorer wins with multiple movement and passes | ✓ |
| *at007.txt* | Attempting to start another game when the user is already in a game | ✓ |
| *at008.txt* | Starting and aborting multiple games using the commands Test and abort | ✓ |
| *at009.txt* | Using the command pass x20 | ✓ |
| *at010.txt* | Moving the explorer before a game has started | ✓ |
| *at011.txt* | Checking the position of the explorer after passing the boundaries of the board | ✓ |
| *at012.txt* | Checking the position of the explorer after moving pass the boundaries of the board (movement West) | ✓ |
| *at013.txt* | Landing on a planet then lifting off | ✓ |
| *at014.txt* | When the Explorer get devoured by a blackhole | ✓ |
| *at015.txt* | When the Explorer lands in a sector with two planets and a yellow dwarf | ✓ |
| *at016.txt* | Checking the status of the Explorer | ✓ |
| *at017.txt* | When explorer and multiple asteroids die | ✓ |
| *at018.txt* | When explorer tries to move to a sector that is full | ✓ |
| *at019.txt* | When planets and janitaur gets devoured by blackhole | ✓ |
| *at020.txt* | When asteroids get imploded by janitaur | ✓ |
| *at021.txt* | Checking if the position of the explorer remains consistent after multiple pass and move commands | ✓ |
| *at022.txt* | Scenario when the explorer moves in multiple directions, attempting to land on a planet | ✓ |
| *at023.txt* | Scenario when a game is recreated with a very low threshold in test mode | ✓ |
| *at024.txt* | Scenario when a game is created with a very high threshold in test mode | ✓ |
| *at025.txt* | Scenario when the user starts and ends a game multiple times while allowing multiple passes to occur | ✓ |

| | | |
|---|---|---|
| *at026.txt* | Scenario when the explorer navigates through the galaxy creates with a low threshold | ✓ |
| *at027.txt* | Scenario when the asteroid threshold is set to a lower number compared to the other movable entities | ✓ |
| *at028.txt* | Scenario when the explorer tries to navigate through the galaxy with the aim of getting destroyed by an asteroid | ✓ |
| *at029.txt* | Scenario that checks the status multiple times after the initial play mode is executed | ✓ |
| *at030.txt* | Checking status command correctness in multiple scenarios | ✓ |
| *at031.txt* | Scenario when the benign, explorer, and malevolent are in the same sector | ✓ |
| *at032.txt* | Scenario when the asteroid, explorer, and janitaur are in the same sector | ✓ |

***Figure 3.*** *Acceptance tests used throughout the development of Sim Odyssey*

The following is an example of unit test procedures used to develop entity sorting and the behaviour of different types of entities:

**ROOT**

Note: * indicates a violation test case

PASSED (4 out of 4)

| Case Type | Passed | Total |
|---|---|---|
| Violation | 0 | 0 |
| Boolean | 4 | 4 |
| All Cases | 4 | 4 |

| State | Contract Violation | Test Name |
|---|---|---|
| Test1 | TEST_SIM_ODYSSEY | |
| PASSED | NONE | t1: test correctness of entity comparator for sorting entities by ID |
| PASSED | NONE | t2: test correctness of CPU_ENTITY states<br>-> check CPU_ENTITY state before encountering sentient entity<br>-> check CPU_ENTITY state after encountering sentient entity |
| PASSED | NONE | t3: test correctness of REPRODUCING_ENTITY states<br>-> check REPRODUCING_ENTITY state before reproducing<br>-> check REPRODUCING_ENTITY state after reproducing |
| PASSED | NONE | t4: test correctness of SENTIENT_ENTITY states<br>-> check SENTIENT_ENTITY state before moving<br>-> check SENTIENT_ENTITY state after moving |

***Figure 4.*** *Unit tests used to check correctness of entity comparator and entity behaviours*

## 7. Appendix (Contract view of classes)

1. The GAME class:

```
note
    description: "Summary description for {GAME}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class interface
    GAME

create
    make

feature -- attributes

    died_this_turn: ARRAY [MOVABLE_ENTITY]

    galaxy: GALAXY

    game: INTEGER_32

    info: SHARED_INFORMATION_ACCESS

    mode: STRING_8

    moved_this_turn: ARRAY [MOVABLE_ENTITY]

feature -- commands

    advance_turn
            -- Allow all movable entities to move once
            -- Entities' behaviours are determined
            -- Entities perform secondary actions if applicable
        require
            galaxy_populated: not info.Shared_info.entities.is_empty
            galaxy_has_movable:
                across
                    info.Shared_info.entities is entity
                some
                    attached {MOVABLE_ENTITY} entity
                end
        ensure
            dead_entities_removed:
                across
                    (old info.shared_info.entities.twin) is entity
                all
                    (attached {MOVABLE_ENTITY} entity as m_e and then m_e.is_dead)
                     implies
                     not info.shared_info.entities.has (m_e)
                end
```

```
    check_entity (l_entity: MOVABLE_ENTITY)
            -- Check the state of the given movable entity
            -- Check if the entity has used fuel, gained fuel, or died if applicable
        ensure
            entity_died:
                l_entity.is_dead
                implies
                died_this_turn.has (l_entity) and not l_entity.death_message.is_empty

    end_game
            -- End the current game, resetting the state of the galaxy
        require
            in_game
        ensure
            game_over: mode.is_empty

    move_entity (l_entity: MOVABLE_ENTITY; dir: INTEGER_32): BOOLEAN
            -- Move given movable entity in given direction
        require
            entity_exists_in_galaxy: info.Shared_info.entities.has (l_entity)
            entity_alive: not l_entity.is_dead
        ensure
            successful_move:
                not l_entity.failed_to_move implies
              (  l_entity.sector.row /= l_entity.prev_sector_row or
                 l_entity.sector.column /= l_entity.prev_sector_col or
                 l_entity.pos /= l_entity.prev_sector_pos              )

    new_game (m: STRING_8;
                  a_threshold, j_threshold, m_threshold, b_threshold, p_threshold: INTEGER_32)
            -- Re-initialize game with given game mode and threshold
            -- Keep track of the current game number
        require
            valid_mode: m ~ "test" or m ~ "play"
        ensure
            game_incremented: game = old game + 1
            mode_set: mode ~ m

    warp_entity (l_entity: SENTIENT_ENTITY)
            -- Move given entity to a random location in the galaxy
        require
            entity_exists_in_galaxy: info.Shared_info.entities.has (l_entity)
            entity_alive: not l_entity.is_dead
        ensure
            successful_move:
                      not l_entity.failed_to_move implies
                    (  l_entity.sector.row /= l_entity.prev_sector_row or
                       l_entity.sector.column /= l_entity.prev_sector_col or
                       l_entity.pos /= l_entity.prev_sector_pos              )

feature -- queries

    check_move (l_entity: MOVABLE_ENTITY; dir: INTEGER_32): BOOLEAN
            -- Determines if moving the given entity in given direction is a valid action

    find_entity (e: ENTITY): detachable ENTITY
            -- Attempts to return the given entity as it exists in the galaxy
            -- Uses comparator to determine if an object equivalent entity exists in the galaxy


    get_game: INTEGER_32
            -- Returns the current game number
```

```
    get_mode: STRING_8
            -- Returns the current game mode

    in_game: BOOLEAN
            -- Checks if a game is in progress

    out: STRING_8
            -- New string containing terse printable representation
            -- of current object

invariant
    valid_game: game >= 0

end -- class GAME
```

***Figure 5.*** *The contract view of GAME.e*

2. The GALAXY class:

```
note
    description: "Galaxy represents a game board in simodyssey."
    author: "Kevin B"
    date: "$Date$"
    revision: "$Revision$"

class interface
    GALAXY

create
    make,
    make_dummy

feature -- attributes

    gen: RANDOM_GENERATOR_ACCESS

    grid: ARRAY2 [SECTOR]
        -- the board

    shared_info: SHARED_INFORMATION

    shared_info_access: SHARED_INFORMATION_ACCESS

feature -- queries

    has_free_sector: BOOLEAN
            -- determine if the grid contains at least one sector with at least one free location
        ensure
            grid_not_full: Result implies shared_info.entities.count < 100

    out: STRING_8
            -- Returns grid in string form

feature -- commands
```

```
    create_stationary_item (sector: SECTOR; id: INTEGER_32): ENTITY
            -- this feature randomly creates one of the possible types of stationary actors

    put_item (e: ENTITY; row: INTEGER_32; col: INTEGER_32)
            -- place given entity in sector located at [row, col] in the grid
        ensure
            added_to_grid: shared_info.entities.has (e)
            added_to_sector: e.sector.contents.has (e)

    remove_item (e: ENTITY; row: INTEGER_32; col: INTEGER_32)
            -- remove given entity in sector located at [row, col] in the grid
        ensure
            removed_from_grid:
                (old shared_info.entities.deep_twin).has (e)
                 implies not
                 shared_info.entities.has (e)
            removed_from_sector:
                (old e.sector.contents.deep_twin).has (e)
                 implies not
                 e.sector.contents.has (e)

    set_stationary_items
            -- distribute stationary items amongst the sectors in the grid.
            -- There can be only one stationary item in a sector

feature --constructor

    make
            -- creates a dummy of galaxy grid

    make_dummy

end -- class GALAXY
```

***Figure 6.*** *The contract view of GALAXY.e*

3. The SECTOR class:

```
note
    description: "Represents a sector in the galaxy."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class interface
    SECTOR

create
    make,
    make_dummy

feature -- Queries

    has_stationary: BOOLEAN
            -- returns whether the location contains any stationary item
```

```
    is_full: BOOLEAN
            -- Is the location currently full?

    out: STRING_8
            -- returns string representation of this sector

    print_sector: STRING_8
            -- Printable version of location's coordinates with different formatting

    sorted_contents: ARRAY [ENTITY]
            -- returns sorted array of entities existing in current sector
            -- sorted by ID in ascending order

feature -- attributes

    chars_out: STRING_8
            -- string representation of entities existing in current sector

    column: INTEGER_32

    contents: ARRAYED_LIST [ENTITY]
            -- holds 4 quadrants

    gen: RANDOM_GENERATOR_ACCESS

    row: INTEGER_32

    shared_info: SHARED_INFORMATION

    shared_info_access: SHARED_INFORMATION_ACCESS

feature -- commands

    make_dummy
            --initialization without creating entities in quadrants

    populate
            -- this feature creates 1 to max_capacity-1 components to be initially stored in the
            -- sector. The component may be a planet or nothing at all.

feature -- constructor

    make (row_input: INTEGER_32; column_input: INTEGER_32; a_explorer: ENTITY)
            -- initialization
        require
            valid_row: (row_input >= 1) and (row_input <= shared_info.Number_rows)
            valid_column: (column_input >= 1) and (column_input <= shared_info.Number_columns)

end -- class SECTOR
```

***Figure 7.*** *The contract view of SECTOR.e*