The system models an extended version of the [Game of Life (GOL)](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)as a series of evolving cell configurations. The project provides a basic GOL implementation, and its purpose is to extend it.

**Cells**

In the provided basic game, cells follow the standard GOL rules. The extended implementation enriches this basic model by introducing specialized cell types with distinct metabolic behaviors. Certain cells can delay death in underpopulation or overpopulation scenarios, surviving multiple generations under stress. Other specialized cells tolerate different neighborhood conditions, thriving either in isolation or in crowded environments. Additionally, cells exchange energy with neighbors, representing life points or resources influencing their evolution.

**Board**

The basic board consists of a grid holding a set of tiles, where each tile holds a cell. In its extended form, the board has interactive tiles, capable of absorbing or donating energy to cells. These interactions can impact cell behaviors, and it is necessary to perform analyses to assess the board’s evolving properties and states.

**Game**

In its basic version, the game operates by simulating the standard GOL evolution, managing cell interactions based on neighbors' states from a given initial configuration. All tiles hold a cell that remains the same along the simulation, toggling its state between alive and dead. The extended implementation introduces global events affecting all tiles and cells across the board at specific generations. These events, which include different scenarios, dramatically alter the board's state and cell dynamics.

**Persistence**

The state of the extended GOL must be persistently stored and retrievable using JPA/Hibernate with an in-memory H2 database. Entities are saved and can be reloaded later, enabling inspection or resumption of the simulation at any given generation.

**Detailed Requirements**

The Extended GOL project provides from the start an implementation of the basic GOL to be extended.

**Provided Code**

GOL classes:

* ExtendedGameOfLife: a facade class where methods required for testing are stubbed.
* Cell: represents a cell in a generation.
* Tile: represents a tile on the board.
* Board: represents a simple static game board.
* Game: defines the simulation context.
* Generation: representation of a game state.
* Interactable and Evolvable: the interfaces required to be implemented by some of the entities
* CellType, CellMood and EventType: enums including the foreseen typologies of cell types, cell moods and event types, respectively, in the extended game.
* Coord: embeddable, hashable JPA value object that encapsulates integer (x, y) coordinates, with proper equals() and "hashCode()" overrides to use it as a key in maps and sets.
* ExtendedGameOfLife: a custom exception class for the extended GOL
* Persistence:
	+ JPAUtil: provides a singleton-based utility to manage the EntityManagerFactory
	+ GenericExtGOLRepository: a generic repository class to be implemented by entity-specific repository classes.
* Configuration:
	+ persistence.xml: configures Hibernate with an in-memory H2 database.
	+ pom.xml: includes dependencies for Hibernate ORM, JPA, H2, and JUnit 4.

**R1 Cells**

**Basic Behavior**

In the provided basic game, cells follow the classical GOL rules:

* **Survival:** A live cell with two or three alive neighbors lives to the next generation.
* **Death by underpopulation:** An alive cell with fewer than two alive neighbors dies.
* **Death by Overpopulation:** An alive cell with more than three alive neighbors dies.
* **Respawn:** A dead cell turns alive when having exactly three alive neighbors.

The game board is considered to be finite: cells on corners have 3 neighbors, cells on edges have 5 neighbors, and central cells have 8 neighbors.

To model interactions with other cells and the environment, cells must implement an interface called Evolvable. The project provides the basic version of the cells through the Cell class, implementing Evolvable, and contains an override of its evolve() method supporting classical GOL rules.

**Extended Behaviors**

**Specialized Cell Types**

Each cell has a lifePoints attribute, representing the energy level of the cell. The base Cell class has this attribute set to the default value 0. The extended GOL implements three different cell types as derived classes of the Cell class:

* Highlander: it can survive for three generations in GOL-related death-inducing conditions.
* Loner: Thrives in isolation, moving the lower bound of survival conditions to 1 neighbor only.
* Social: It moves the upper bound of survival conditions up to a maximum of 8 neighbors.

All cells have a cellType attribute, and basic cells are marked as BASIC.

At each generation, the evolve() method updates the cell's lifePoints according to its neighborhood and interactions.

* Death decreases it by one
* Survival-maintaining conditions increase it by one
* Respawn resets its lifePoints to 0

In addition to the basic GOL rules compliance, a cell needs to have 0 ore more in order to be alive. Yet, cells in death-inducing conditions according to GOL die even if they have positive energy levels. Dead cells do not update lifepoints otherwise.

**Vampires and healers**

Each Cell can have three moods: NAIVE, HEALER, or VAMPIRE. When two cells interact, by implementing Interactable, depending on their respective mood, different outcomes follow:

* HEALER + NAIVE: The HEALER generates 1 lifePoint for the NAIVE.
* HEALER + HEALER: Nothing happens.
* HEALER + VAMPIRE: The VAMPIRE absorbs 1 lifePoint from the HEALER.
* VAMPIRE + VAMPIRE: Nothing happens.
* VAMPIRE + NAIVE: The VAMPIRE absorbs 1 lifePoint from the NAIVE and turns them into a VAMPIRE.
* NAIVE + NAIVE: Nothing happens.

All cells have a cellMood attribute. The mood can change multiple times for the same cell, depending on its interactions with other Cells on the Board and the events that occur during a Game.

**Persistence**

All Cell-derived classes, as well as the base class, must be annotated as JPA entities, ensuring that the entire cell hierarchy can be saved and loaded through your JPA repositories. The provided generic repository class GenericExtGOLRepository<E,I> must be implemented parametrizing it for Cell to provide basic operations and a load method so that the Cell’s state is persistently stored and retrievable via JPA. For example:

public class CellRepository extends GenericExtGOLRepository<Cell, Long> {

 public CellRepository() { super(Cell.class); }

}

Each repository must implement the load(...) method (and any custom queries) so that the board’s and game’s full state can be saved and reloaded via JPA.

**R2 Board**

**Basic Behaviors**

The board is a grid with fixed dimensions (M×N), corresponding to M\*N instances of simple Tile objects with coordinates x and y, each one holding a single Cell.

**Extended Behaviors**

**Interactive Tiles**

On the Board, each Tile has a lifePointModifier attribute value, by default 0, and implements the Interactable interface. Tiles can have different impacts on the Cell's lifePoints, depending on the lifePointModifier attribute value: if they have a positive lifePointModifier they adding its current value to the lifePoints to the cell, if it is negative they subtract, and if it is zero they have no effect. Interactions with the tile impact the cell at the beginning of each generation.

**Visualization**

The Board supports string-based visualization through the method visualize(). Each Tile hosting a dead Cell is represented with a 0, while Cells on the board are represented with:

* Cell: C
* Highlander: H
* Loner: L
* Social: S

This method produces visualizations with the following format:

0C00H

L00CS

0H000

C0000

**Analytical Methods**

The Board class provides the following analysis methods. Each accepts a Generation instance (or generation range) and returns the requested information:

| **Method Signature** | **Description** |
| --- | --- |
| public Integer countCells(Generation gen) | Returns the **total number** of alive cells in gen. |
| public Cell getHighestEnergyCell(Generation gen) | Finds the **single** cell with the highest lifePoints, picks the one closer to the top left corner in case of tie. |
| public Map<Integer, List<Cell>> getCellsByEnergyLevel(Generation gen) | Groups **alive cells** by their current lifePoints. |
| public Map<CellType, Integer> countCellsByType(Generation gen) | Counts alive cells **per** CellType. Tip: use custom querying in the dedicated repository. |
| public List<Cell> topEnergyCells(Generation gen, int n) | Returns the **top n** cells sorted by descending lifePoints. |
| public Map<Integer, List<Cell>> groupByAliveNeighborCount(Generation gen) | Groups cells by their **number of live neighbors**. |
| public IntSummaryStatistics energyStatistics(Generation gen) | Computes summary statistics (count, min, max, sum, average) over all cells’ lifePoints. |
| public Map<Integer, IntSummaryStatistics> getTimeSeriesStats(int fromStep, int toStep) | Returns a **time series** of energy statistics for each generation step in [fromStep, toStep]. |

**Persistence**

The Board's complete configuration must be fully persistable: all entities must be annotated with appropriate ID and relationship mappings. The provided generic repository class GenericExtGOLRepository<E,I> must be implemented parametrizing it for Board to provide basic operations and a load method so that the board’s state, including the position and complete states of Cells and Tiles it holds, is persistently stored and retrievable via JPA. For example:

public class BoardRepository extends GenericExtGOLRepository<Board, Long> {

 public BoardRepository() { super(Board.class); }

}

**Each repository must implement the load(...) method (and any custom queries) so that the board’s and game’s full state can be saved and reloaded via JPA.**

**R3 Game**

**Basic Behaviors**

The Game orchestrates the evolution of the Board along Generations using standard GOL rules and basic cell behavior. This starts with setting the initial Board configuration and performing the game evolution routing until a target number of Generations is reached. The routine includes:

1. **Neighbor Detection**: At initialization, scan the Board to determine each cell's neighbors.
2. **Neighborhood evaluation**: The Game triggers every Cell to evaluate the aliveness states of its neighbors at the current Generation.
3. **Evolution**: The Game sets the new state according to GOL rules of each Cellfor the next Generation following the neighborhood evaluation.

**Extended Behaviors**

**Events**

The extended Game can trigger global events setting the state of all Tiles on the Board for a single Generation. Possible events are:

* **Cataclysm**: all Tiles reset all lifePoints from the Cell they hold to 0.
* **Famine**: all Tiles absorb exactly 1 lifePoints from the Cell they hold.
* **Bloom**: Tiles grant exactly 2 lifePoints to the cells sitting on them.
* **Blood Moon**: every VAMPIRE seated on a Tile automatically absorbs 1 lifePoints from each adjacent NAIVE or HEALER and turns them into new VAMPIREs.
* **Sanctuary**: all tiles grant each HEALER 1 lifePoints, while blocking all VAMPIREs by turning them NAIVE.

Each event impacts a single generation. Each generation includes at most one event. Events impact all cells on the board at the beginning of each generation.

**Persistence**

The persistence layer must capture every Generation snapshot (board layout and cell states) together with its associated Game entities. Upon loading, loadEvents() is responsible for querying and re-attaching each event to the corresponding Generation in the Game, thereby fully restoring the simulation’s timeline and enabling replay or inspection. The provided generic repository class GenericExtGOLRepository<E,I> must be implemented parametrizing it for Game to provide basic operations and a load method so that the Game’s state, including all of its Generations and events, is persistently stored and retrievable via JPA. For example:

public class GameRepository extends GenericExtGOLRepository<Game, Long> {

 public GameRepository() { super(Game.class); }

}

Each repository must implement the load(...) method (and any custom queries) so that the board’s and game’s full state can be saved and reloaded via JPA.