AGROPLANNER

TECHNICAL DOCUMENTATION

By Bonucci Federico
0323567

ISPW 2025/2026

INDEX

Note on Diagram Resolution: Due to the document's formatting constraints and the high level of detail inherent in the UML models, high-resolution versions of all diagrams are archived in the diagrams directory of the GitHub repository. [https://github.com/FeDevv/AGROPLANNER]

1. SOFTWARE REQUIREMENTS SPECIFICATION

1.1 INTRODUCTION

1.1.1 AIM OF THE DOCUMENT

The main point of the following document is to objectively present the fundamental specifications and functional requirements of the Agroplanner system. A comprehensive review of the internal and external aspects of the system will be conducted, supported by UML diagrams to illustrate the architecture and behavior. Specifically, this document details the system's modular architecture, focusing on the MVC pattern implementation across dual interfaces (CLI and JavaFX) and the central control flow managed by the Orchestrator.
Furthermore, it outlines the core logic driven by a Genetic Algorithm designed for spatial optimization, alongside the software engineering principles and GoF (Gang of Four) Design Patterns utilized to ensure system maintainability and scalability.

1.1.2 OVERVIEW OF THE DEFINED SYSTEM

The system presented herein is an advanced agricultural planning application designed to assist users in optimizing the spatial arrangement of crops within a defined plot of land. Its primary function is to automate the complex task of land layout planning by utilizing computational intelligence to maximize land usage while respecting biological constraints.
The application allows the user to define the geometric properties of their terrain (e.g., rectangular, circular, or elliptical shapes) and select specific plant varieties from a curated inventory. To ensure accurate botanical data, the system distinguishes between standard Users, who create projects, and Agronomists, who are responsible for populating and maintaining the detailed sheets of plant varieties available in the system.
Once the parameters are defined, Agroplanner employs a sophisticated Genetic Algorithm to generate a valid and optimized solution for plant distribution. The user can visualize the result, save the solution using various persistence methods (Database or File System), and export the final layout to standard formats (such as .xlsx, .csv, or .txt) for practical use.

Furthermore, the system is designed with high flexibility, offering dual interaction modes: a lightweight Command Line Interface (CLI) for rapid execution and a fully featured Graphical User Interface (JavaFX) for enhanced visualization and interaction.

1.1.3 HARDWARE AND SOFTWARE REQUIREMENTS

Hardware Requirements The fundamental hardware requirement for the system is a workstation capable of executing Java applications with a graphical user interface. Given the computational nature of the Genetic Algorithm (specifically the use of parallel processing in the gasystem module), a multi-core processor is highly recommended to ensure optimal performance during solution generation. Additionally, sufficient disk space is required to store persistence data (H2 Database files or raw File System exports) when the application is not operating in volatile mode.

Software Requirements Regarding the software environment, the primary requirement is the installation of the Oracle JDK 23 (or a compatible OpenJDK distribution) to correctly execute the application JAR file and support the JavaFX dependencies. For data persistence, the system utilizes an H2 Database Engine. Since the application handles the connection via the internal DBConnection singleton, no external SQL server installation is strictly required; however, the operating environment must grant the application sufficient read/write privileges to the local file system to manage the database files and export directories.

1.1.4 RELATED SYSTEMS, PROS AND CONS

Other systems related to the domain of agricultural planning and layout optimization include:

WEB-BASED GRID PLANNERS
(e.g., Garden Planner, Kitchen Garden Aid) These are typically web or desktop applications that allow users to drag and drop plant icons onto a discretized grid representing the garden.

- **Pros**: They usually offer a very intuitive graphical user interface and cloud-based storage, making them accessible to non-technical users.

- **Cons**: The main disadvantage is the reliance on a fixed grid (discrete geometry), which often leads to suboptimal land usage compared to the continuous coordinate system used in Agroplanner. Furthermore, they lack automated solving capabilities; the user must manually place every plant, making the process tedious for large plots.

MANUAL SPREADSHEET OR CAD TOOLS

This category refers to the use of general-purpose tools like Microsoft Excel or CAD software to map out crop fields.

- **Pros**: These tools offer infinite flexibility regarding the data entered and are widely available in professional environments.
- **Cons**: They provide no domain-specific logic. There is no automatic validation of biological constraints (e.g., plant distance), and no visual feedback on collisions. Most importantly, they lack any form of algorithmic optimization, leaving the burden of finding an efficient layout entirely on the user.

1.2 USER STORIES

1.2.1 USER STORY #1

> **As a** User,
> **I want to** define precise terrain dimensions and configure a specific inventory of plant species,
> **so that** I can utilize the system to generate an optimized spatial layout that respects biological constraints and minimizes waste.

1.2.2 USER STORY #2

> **As a** User,
> **I want to** save the generated planting solutions to a persistent storage,
> **so that** I can retrieve the data later for further analysis or export it for external usage.

1.2.3 USER STORY #3

> **As an** Agronomist,
> **I want to** create and define detailed plant variety sheets, including specific spatial requirements,
> **so that** my professional expertise can be standardized and utilized by the user base to generate accurate agricultural projects.

1.3 FUNCTIONAL REQUIREMENTS

1.3.1 FUNCTIONAL REQUIREMENT #1

> The system shall provide a data entry interface allowing the **Agronomist** to define and persist **Plant Variety Sheets** encompassing mandatory biological constraints and professional attribution data.

1.3.2 FUNCTIONAL REQUIREMENT #2

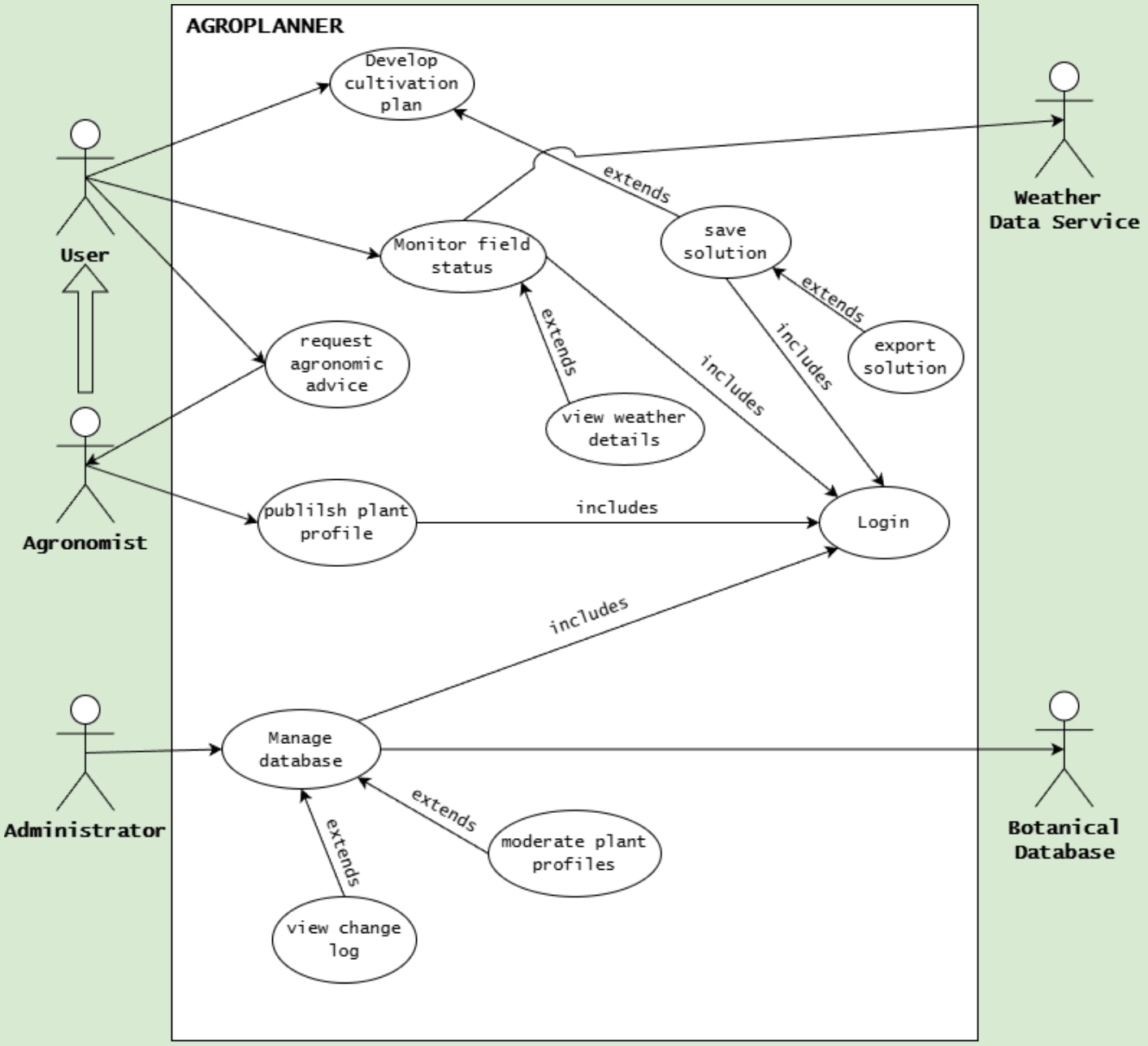
> The system shall provide a configuration mechanism allowing the **User** to initialize a cultivation project by defining the terrain's geometric properties and selecting a subset of available plant varieties from the global inventory.

1.3.3 FUNCTIONAL REQUIREMENT #3

> The system shall provide an administrative interface allowing the **Administrator** to maintain the integrity of the database by permanently removing plant variety sheets deemed non-compliant or erroneous.

1.4 USE CASES

1.4.1 OVERVIEW DIAGRAM



1.4.2 INTERNAL STEPS

Use Case: Develop cultivation plan

1) **The User** Requests to initialize a new cultivation project.

2) **The System** Retrieves and displays the list of available terrain geometric shapes.
3) **The User** Selects the desired terrain shape.
4) **The System** Requests the specific geometric parameters required for the selected shape.
5) **The User** Inputs the specific terrain dimensions (in meters).
6) **The System** Retrieves and displays the categorization of available Plant Types.
7) **The User** Selects a specific Plant Type category.
8) **The System** Retrieves and displays the list of specific **Plant Variety Sheets** available within that category.
9) **The User** Selects one or more specific Plant Varieties to populate the project inventory.
10) **The System** Executes the **Genetic Algorithm** to generate a solution and, upon completion, displays the calculated **Fitness Score**.
11) **The System** Prompts the User to choose whether to visualize the full detailed solution layout.
12) **The User** Confirms the visualization request.
13) **The System** Renders and displays the complete optimized planting solution.

Extensions:

5a. Invalid Geometric Input: The User enters invalid numerical values (e.g., negative numbers) or geometrically inconsistent dimensions. The System detects the validation error, displays a warning message explaining the geometric constraint, and prompts the User to re-enter the dimensions (**Return to step 4**).

10a. Optimization Failure: The System fails to converge on a valid solution within the maximum generation limit. The System halts the process, displays an optimization failure message, and returns the User to the initial configuration phase to adjust parameters (**Return to step 2**).

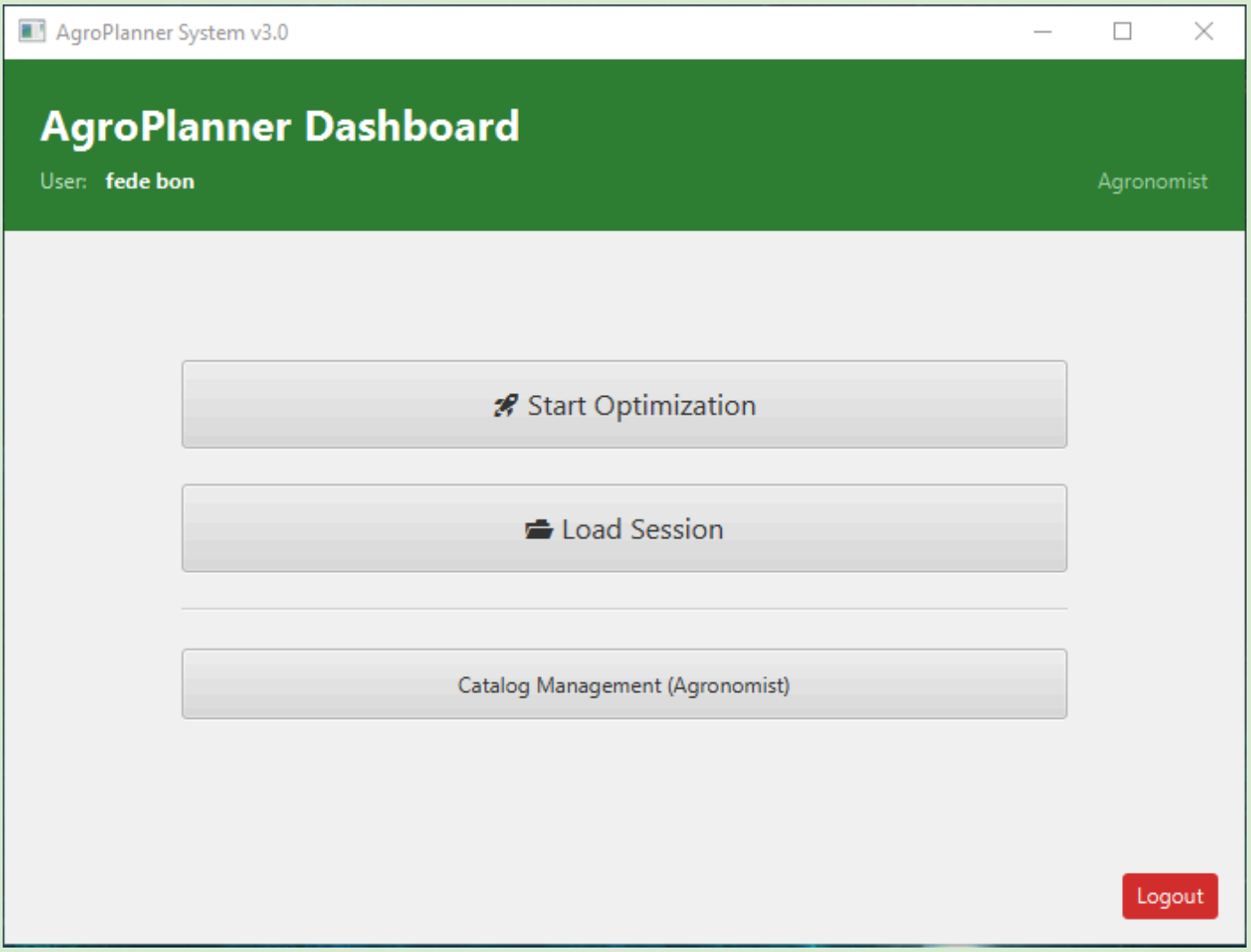
10b. Execution Timeout: The calculation exceeds the predefined time limit due to computational constraints. The System halts the execution, displays a timeout error message, and resets the process (**Return to step 2**).

11a. User Declines Details: The User reviews the Fitness Score but declines the option to view the full layout details. The System acknowledges the choice, and the Use Case concludes.

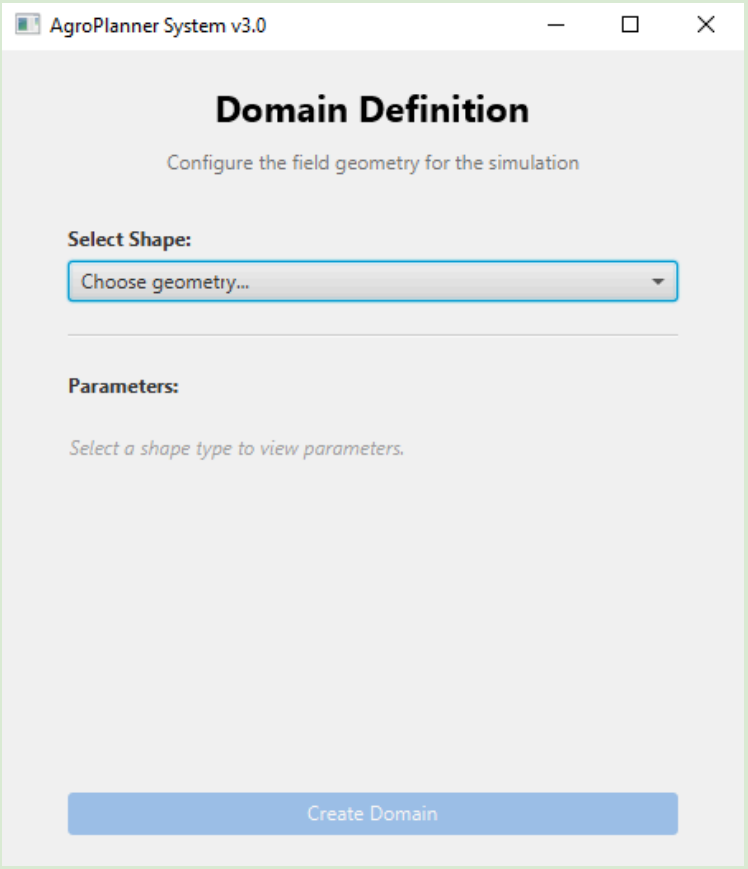
---

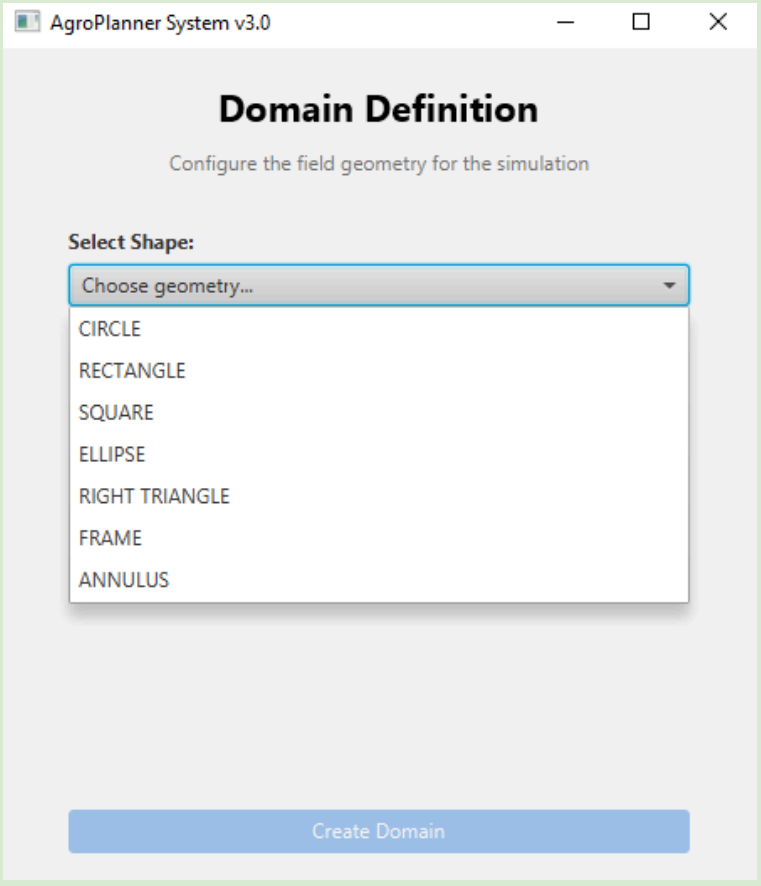
2. STORYBOARDS

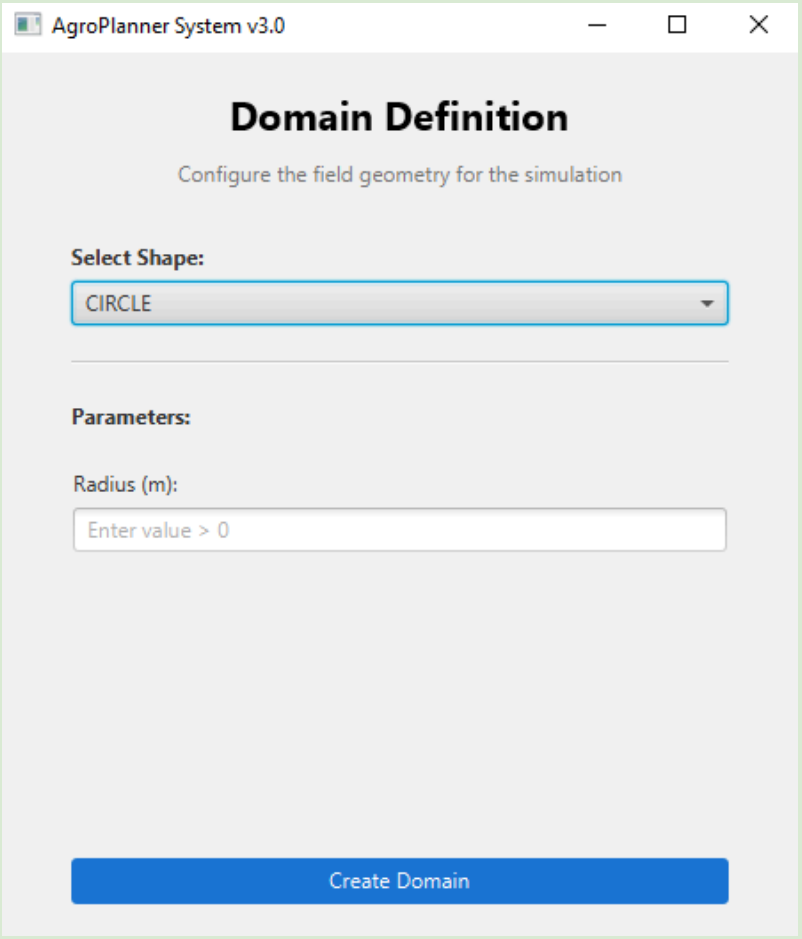
Storyboard 1: Main Menu Dashboard This scene depicts the primary landing interface following successful authentication. The screen is designed to provide immediate feedback on the active session context, clearly displaying the user's identity and assigned privilege level (User, Agronomist, or Admin). It serves as the central navigational hub, granting access to the system's core functionalities through a set of role-specific interactive controls.



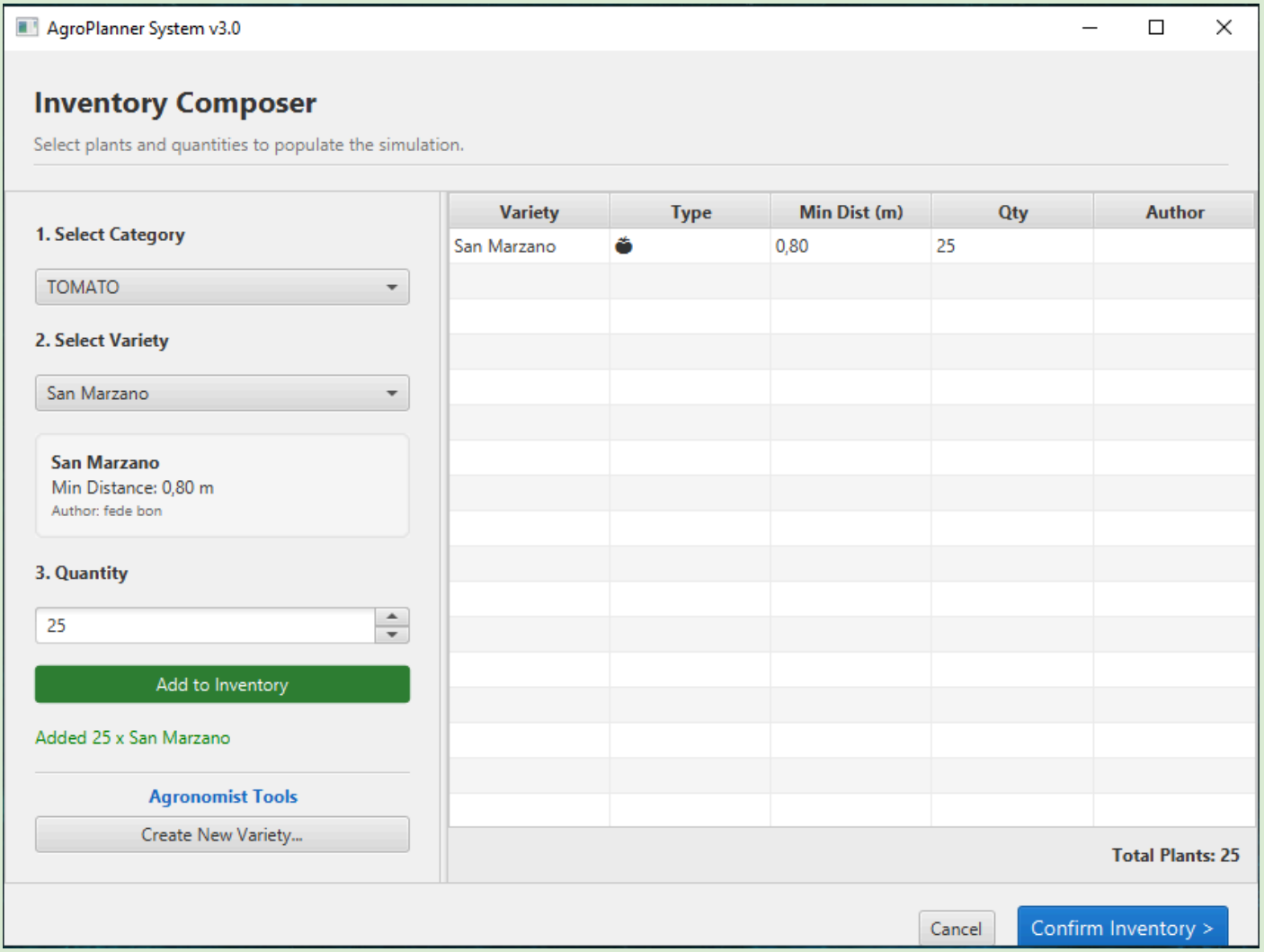
Storyboards 2-4: The Domain Creation Protocol These sequences illustrate the end-to-end workflow for defining a new cultivation field. The interface features a **dynamic, context-aware design**: starting from the selection of the geometric topology, the system adaptively updates the input form to request only the parameters strictly necessary for that specific shape (e.g., hiding 'Radius' when 'Rectangle' is chosen).


AgroPlanner System v3.0
Domain Definition
Configure the field geometry for the simulation
Select Shape:
Choose geometry...
Parameters:
Select a shape type to view parameters.
Create Domain

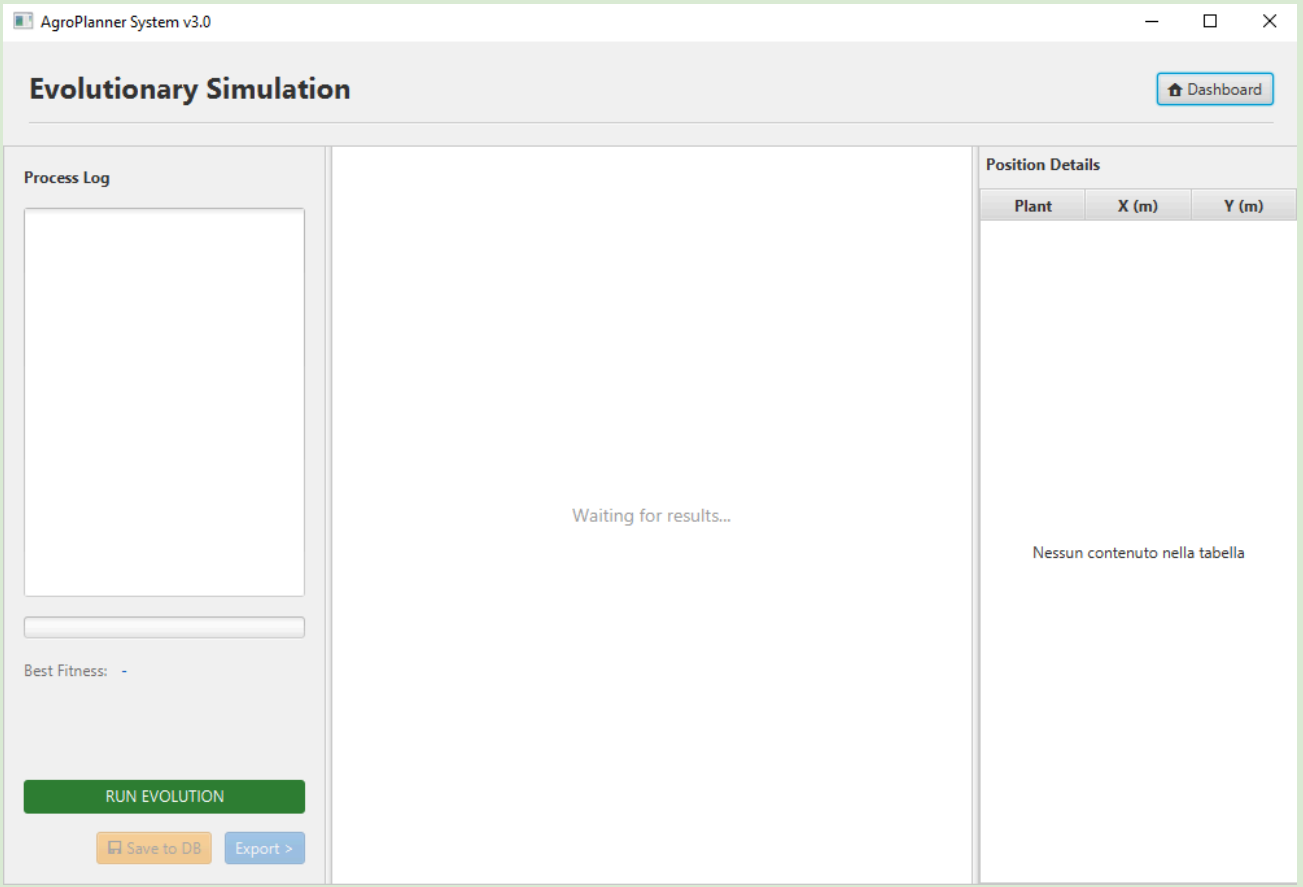

AgroPlanner System v3.0
Domain Definition
Configure the field geometry for the simulation
Select Shape:
Choose geometry...
CIRCLE
RECTANGLE
SQUARE
ELLIPSE
RIGHT TRIANGLE
FRAME
ANNULUS
Create Domain

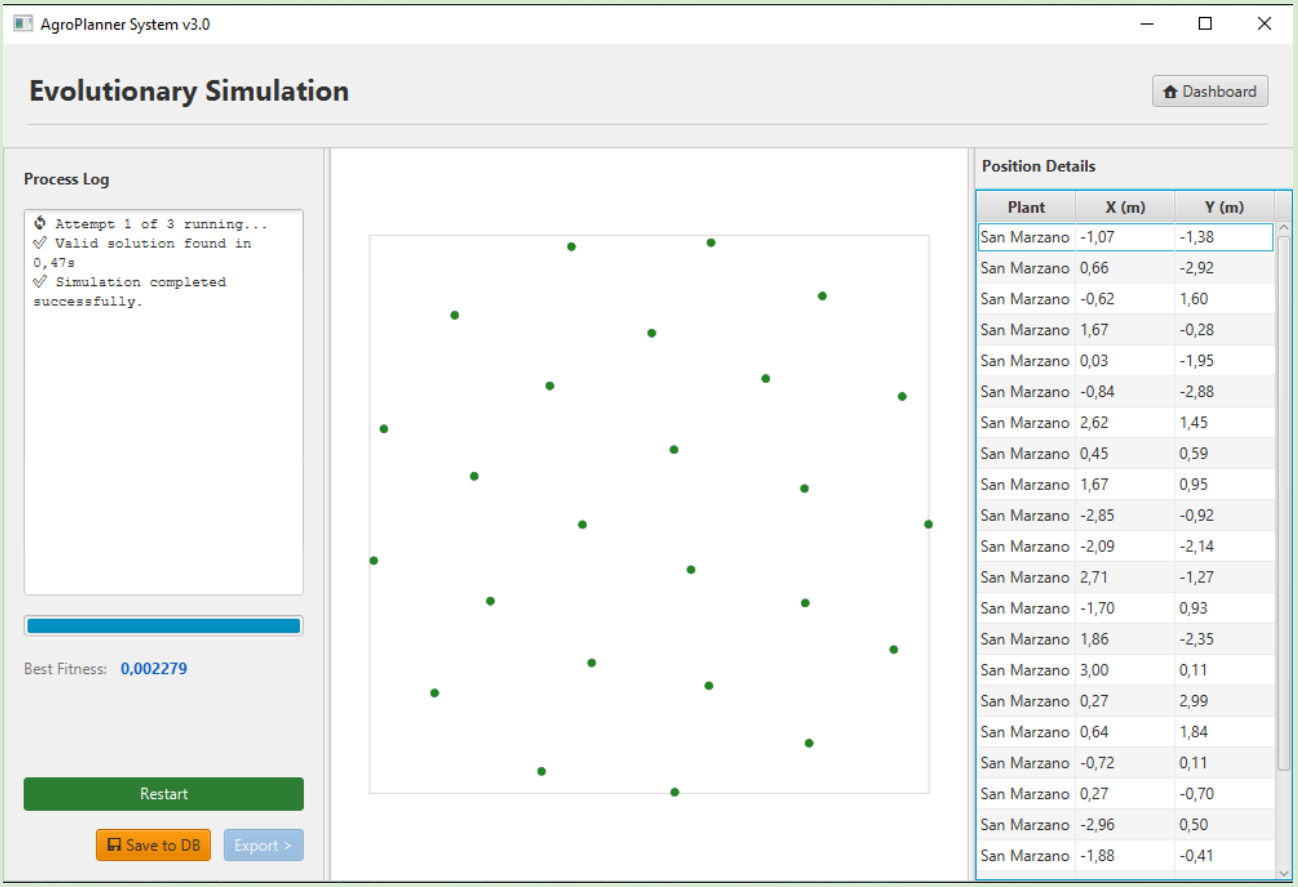

AgroPlanner System v3.0
Domain Definition
Configure the field geometry for the simulation
Select Shape:
CIRCLE
Parameters:
Radius (m):
Enter value > 0
Create Domain

Storyboard 5: Plant Selection & Inventory Composition This interface presents the 'Composer' module, where the user defines the biological inputs for the cultivation plan. Through an interactive catalog, the user selects specific plant varieties and assigns target quantities to each. This step effectively populates the PlantInventory, establishing the population constraints that the optimization engine will attempt to arrange within the domain.



Storyboards 6-7: Execution and Result Visualization These scenes capture the transition from configuration to computation. The first view shows the system in its 'Ready State', awaiting the user's command to trigger the **Genetic Algorithm**. Upon clicking 'Run Evolution', the interface updates to display the optimized solution. The results are presented in a dual format: a **graphical rendering** of the spatial arrangement within the domain boundaries, and a corresponding **tabular list** detailing the precise (x, y) coordinates for each positioned plant.

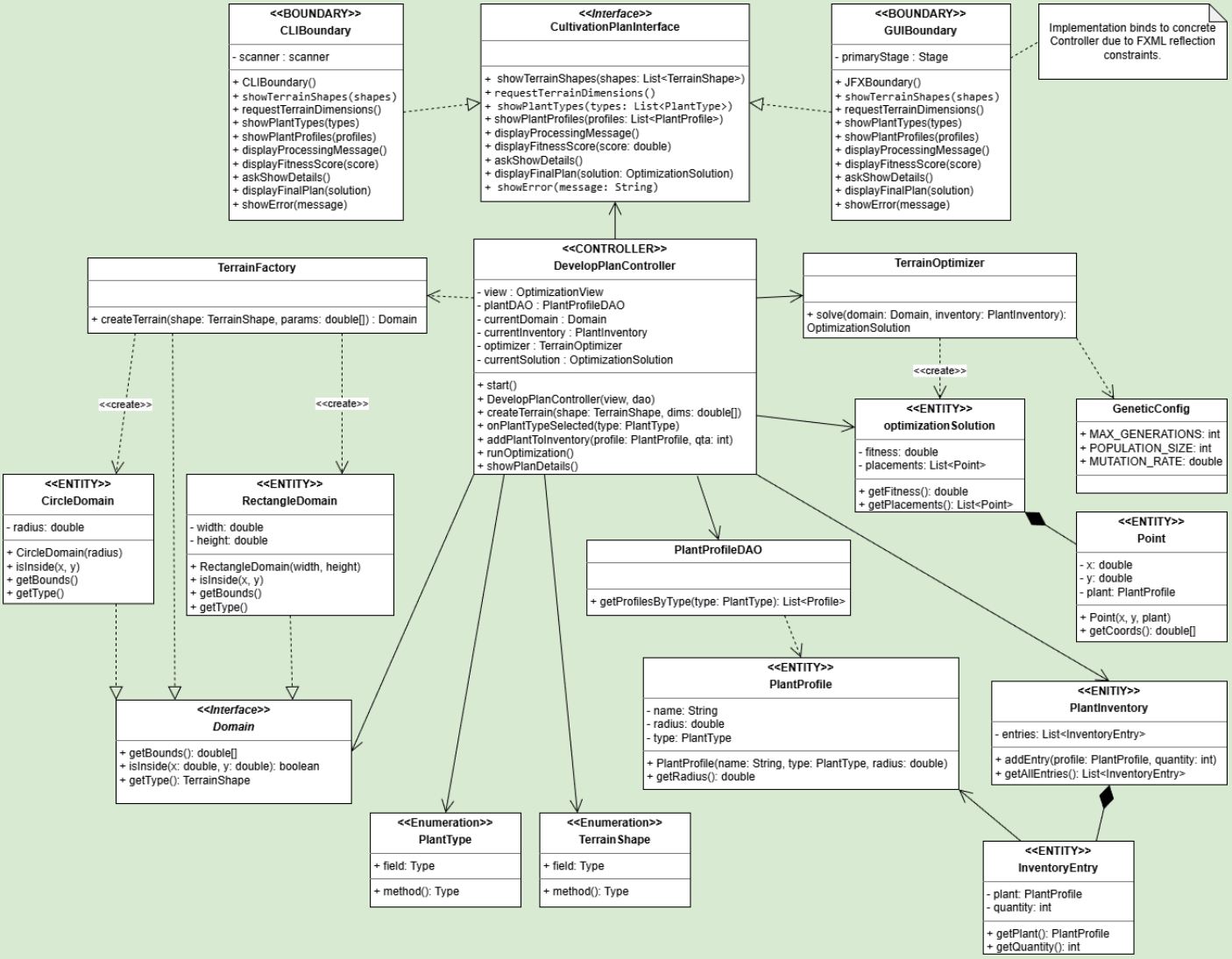



AgroPlanner System v3.0
Evolutionary Simulation
Dashboard
Process Log
Attempt 1 of 3 running...
Valid solution found in 0,47s
Simulation completed successfully.
Best Fitness: 0,002279
Restart
Save to DB
Export >
Position Details
Plant X (m) Y (m)
San Marzano -1,07 -1,38
San Marzano 0,66 -2,92
San Marzano -0,62 1,60
San Marzano 1,67 -0,28
San Marzano 0,03 -1,95
San Marzano -0,84 -2,88
San Marzano 2,62 1,45
San Marzano 0,45 0,59
San Marzano 1,67 0,95
San Marzano -2,85 -0,92
San Marzano -2,09 -2,14
San Marzano 2,71 -1,27
San Marzano -1,70 0,93
San Marzano 1,86 -2,35
San Marzano 3,00 0,11
San Marzano 0,27 2,99
San Marzano 0,64 1,84
San Marzano -0,72 0,11
San Marzano 0,27 -0,70
San Marzano -2,96 0,50
San Marzano -1,88 -0,41

3. DESIGN

3.1 CLASS DIAGRAM

3.1.1 VOPC (ANALYSIS)



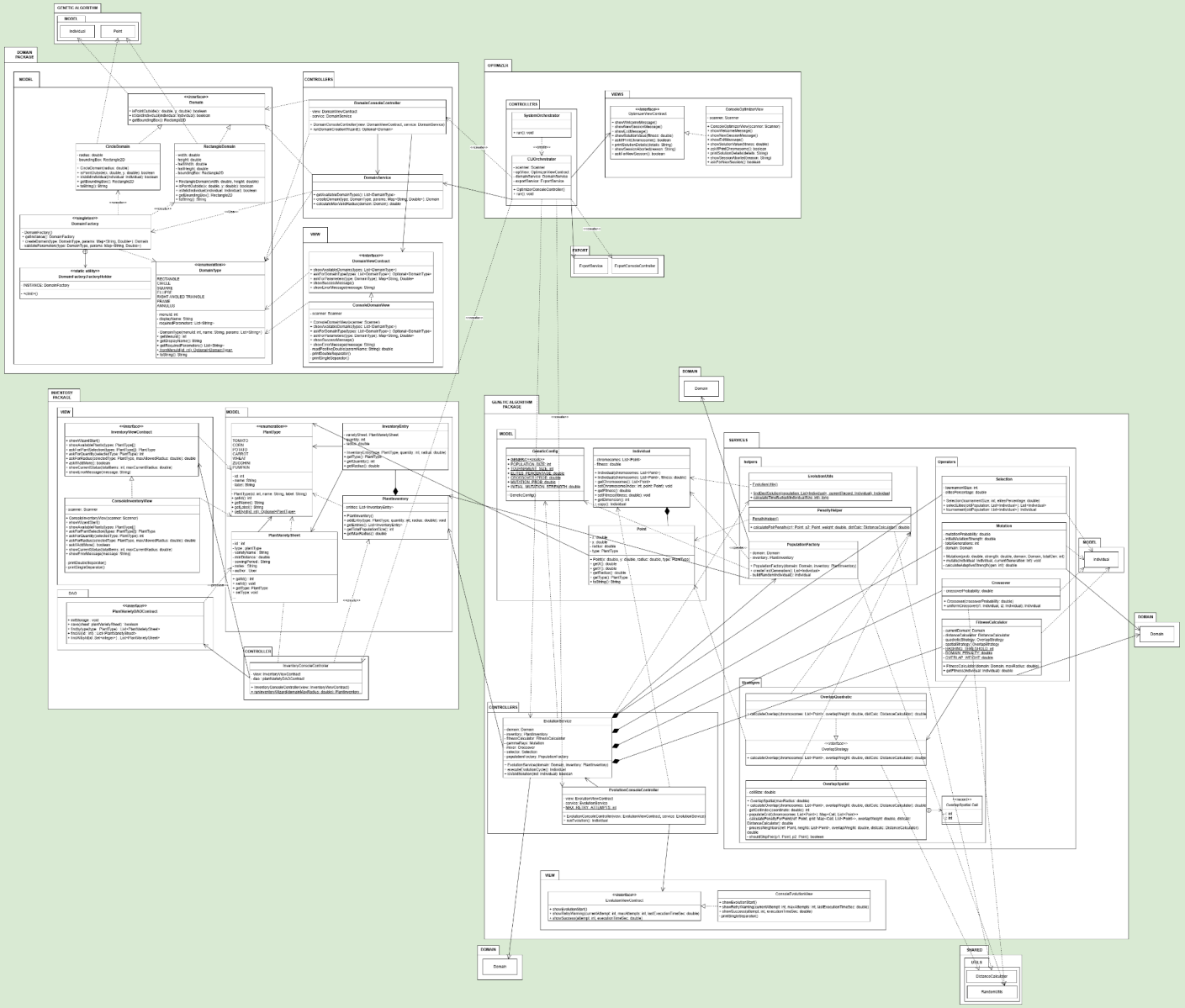
Architectural Note: View-Controller Coupling

The reader may observe a **structural distinction** between the two boundary implementations. While the **CLIBoundary** strictly adheres to the Dependency Inversion Principle by interacting solely with the **CultivationPlanInterface**, the **JFXBoundary** exhibits a direct dependency on the concrete DevelopPlanController. This coupling is an inherent constraint of the **JavaFX framework** when using **FXML** files. The **FXMLLoader** utilizes reflection to instantiate the controller specified in the **fx:controller** attribute, necessitating a concrete class reference rather

than an interface. Although the implementation differs technically to accommodate the framework, the **JFXBoundary logically respects** the same interaction contract defined by the use case interface.

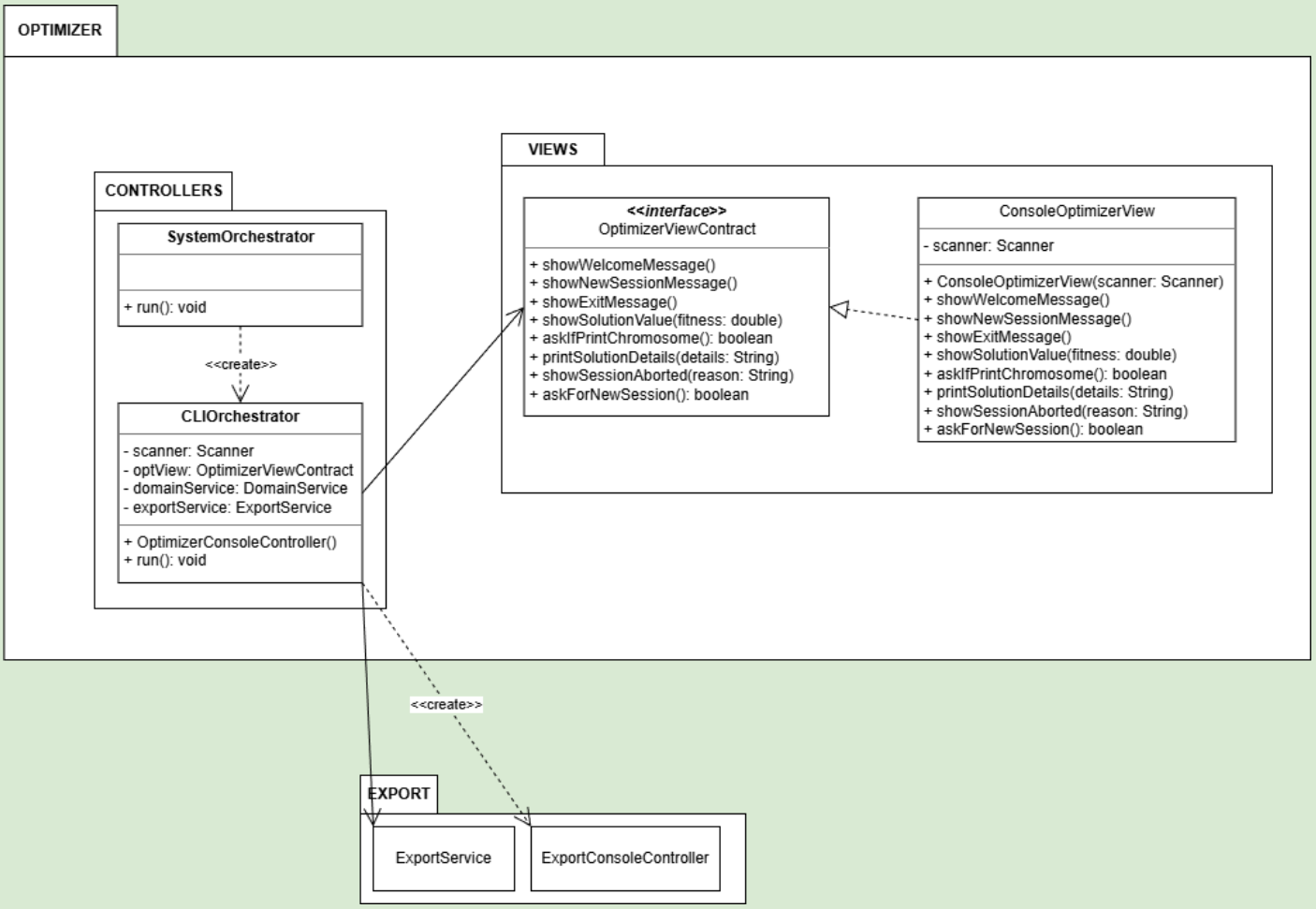
3.1.2 DESIGN LEVEL DIAGRAM

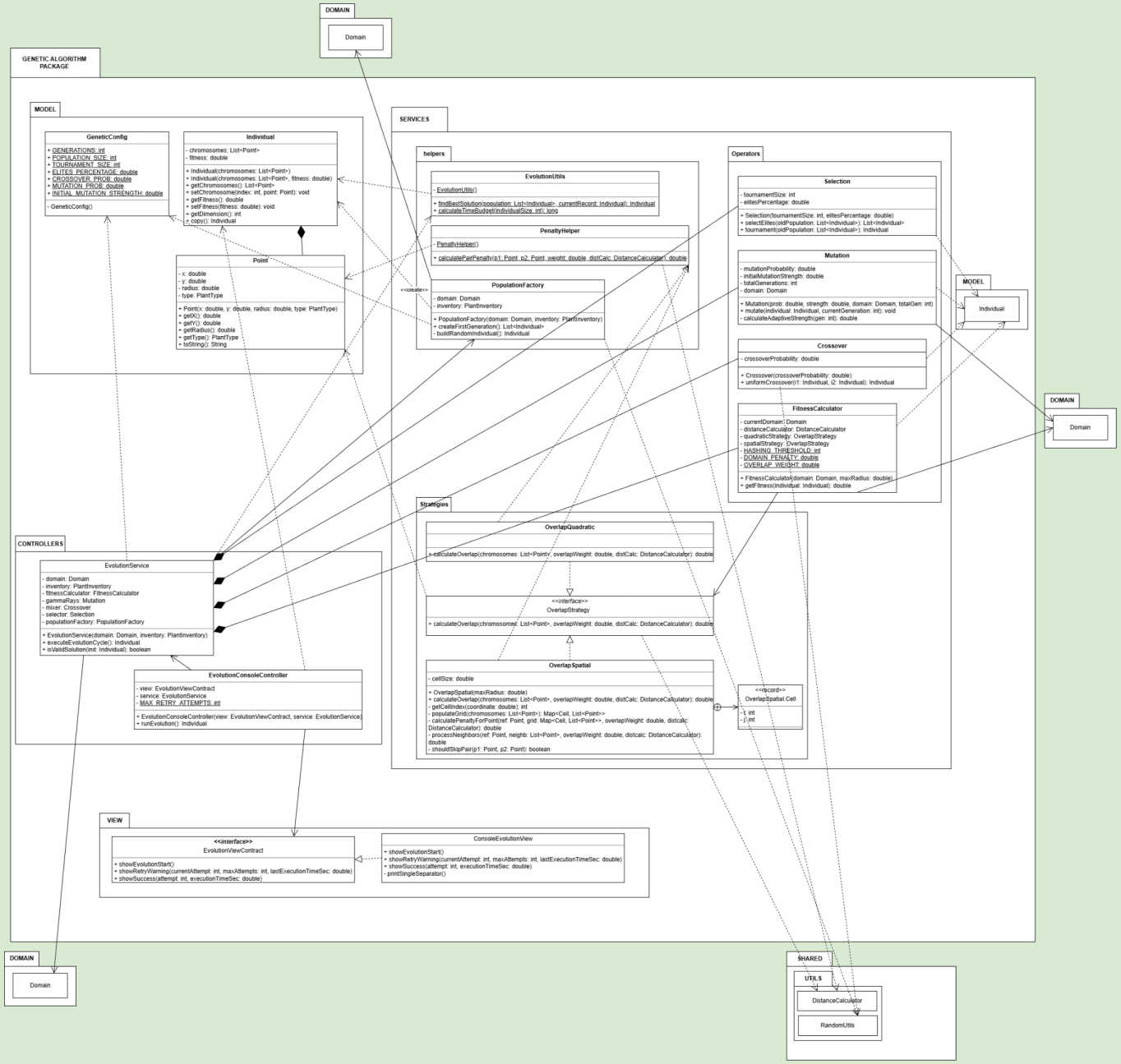
The following diagram details the static structure and architectural composition of the "**Develop Cultivation Plan**" use case. It provides a comprehensive view of the concrete classes, interfaces, and specific relationships implemented in the system. The diagram focuses on the **CLI** operational context, exposing the exact attributes and method signatures utilized to orchestrate the generation of the agricultural layout.

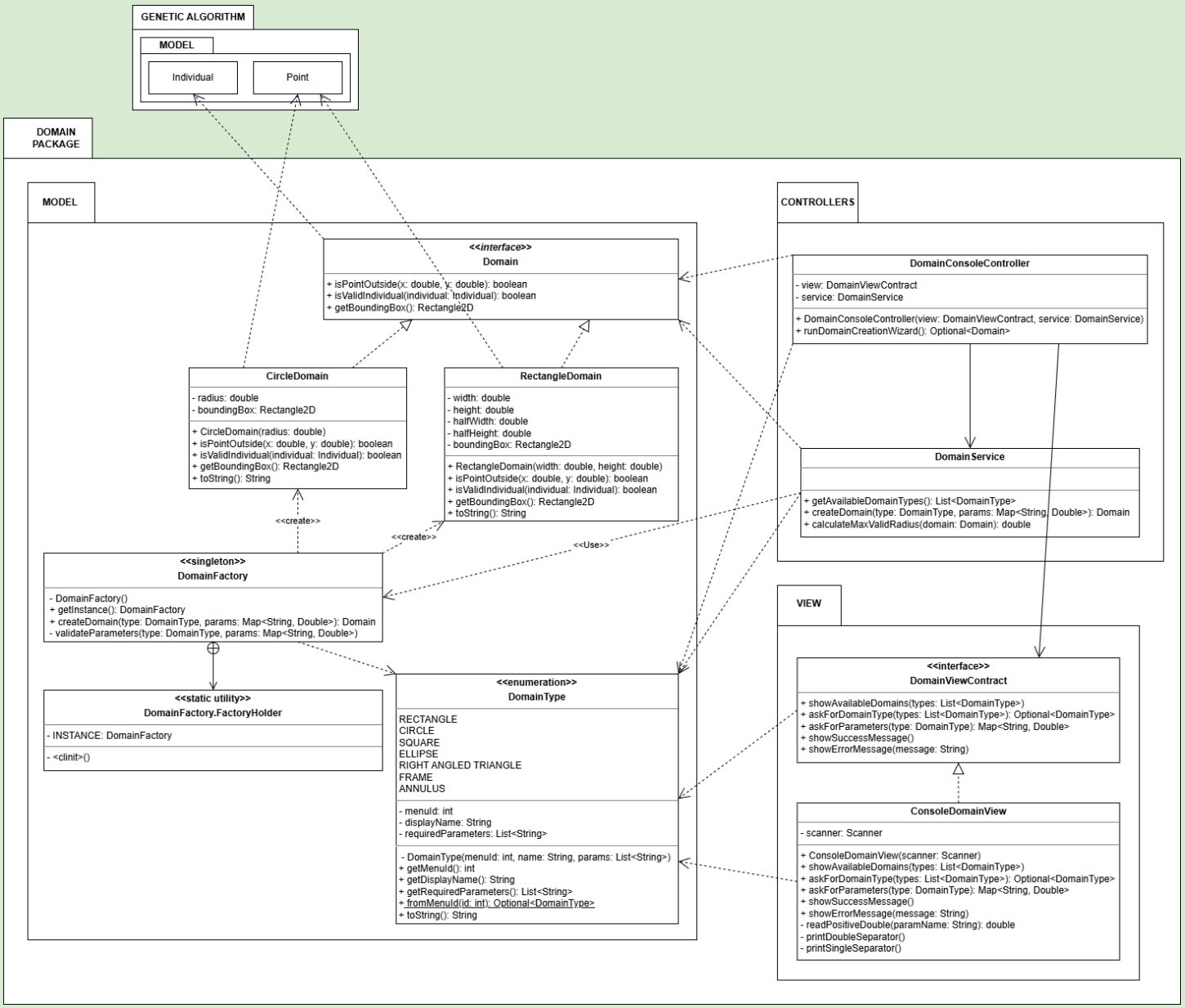


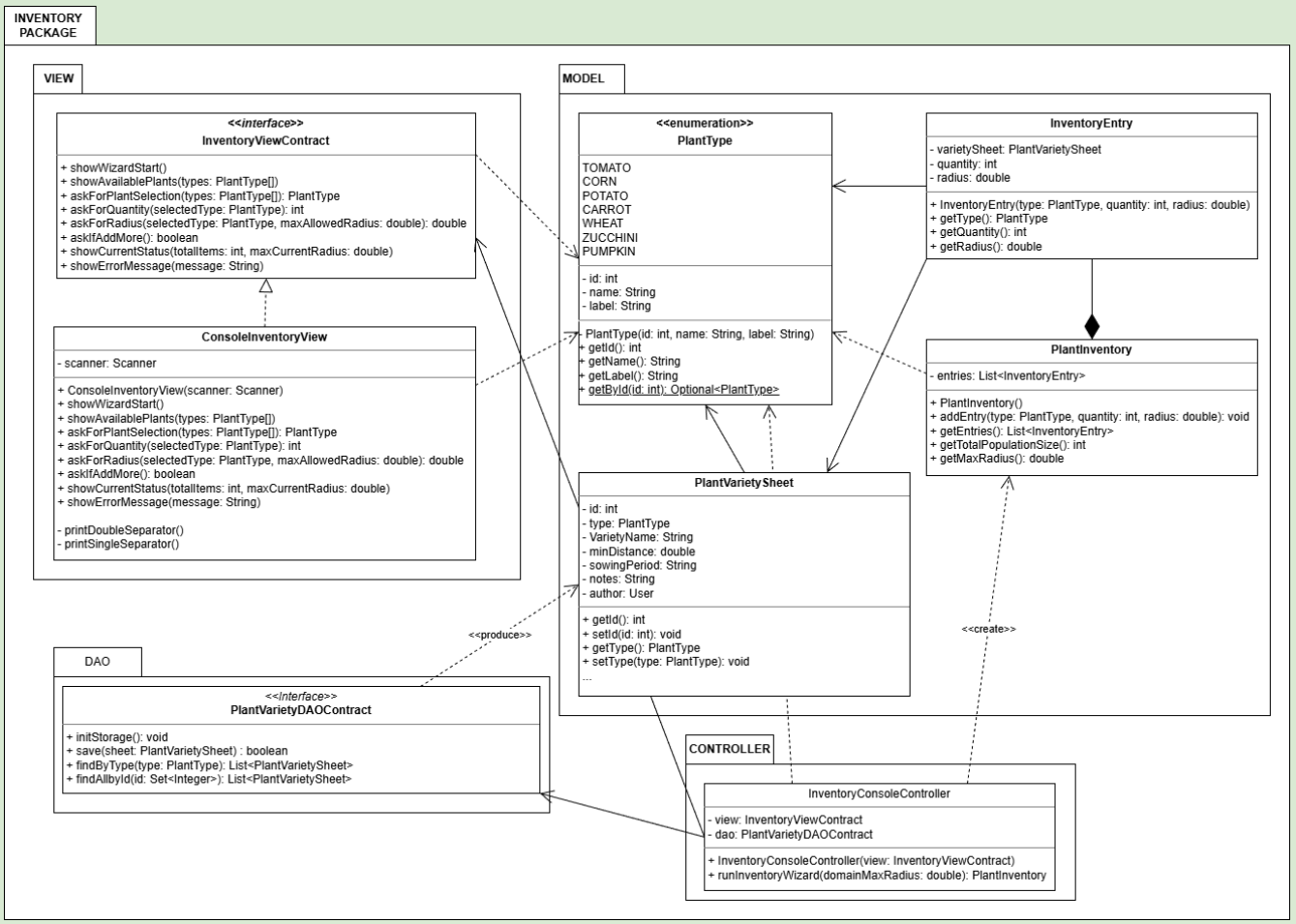
Note on Visual Representation: To ensure maximum legibility and minimize visual clutter caused by overlapping connectors, certain classes (used by multiple distinct packages) may appear as **visual proxies** (duplicates) in different sections of the diagram. These proxies represent the same logical entity and are positioned locally to simplify relationship tracking.

Due to the complexity and scale of the complete Design Class Diagram, the following section presents **detailed views** of specific packages. These magnified excerpts allow for a more granular analysis of the internal class attributes and local relationships without the visual clutter of the global architecture.




DOMAIN
Domain
GENETIC ALGORITHM PACKAGE
MODEL
GeneticConfig
+ GENERATIONS: int
+ POPULATION_SIZE: int
+ TOURNAMENT_SIZE: int
+ ELITES_PERCENTAGE: double
+ CROSSOVER_PROB: double
+ MUTATION_PROB: double
+ INITIAL_MUTATION_STRENGTH: double
- GeneticConfig()
Individual
- chromosomes: List<Point>
- fitness: double
+ Individual(chromosomes: List<Point>)
+ Individual(chromosomes: List<Point>, fitness: double)
+ getChromosomes(): List<Point>
+ setChromosome(index: int, point: Point): void
+ getFitness(): double
+ setFitness(fitness: double): void
+ getDimension(): int
+ copy(): Individual
Point
- x: double
- y: double
- radius: double
- type: PlantType
+ Point(x: double, y: double, radius: double, type: PlantType)
+ getX(): double
+ getY(): double
+ getRadius(): double
+ getType(): PlantType
+ toString(): String
SERVICES
helpers
EvolutionUtils
- EvolutionUtils()
+ findBestSolution(population: List<Individual>, currentRecord: Individual): Individual
+ calculateTimeBudget(individualSize: int): long
PenaltyHelper
- PenaltyHelper()
+ calculatePairPenalty(p1: Point, p2: Point, weight: double, distCalc: DistanceCalculator): double
<<create>>
PopulationFactory
- domain: Domain
- inventory: PlantInventory
+ PopulationFactory(domain: Domain, inventory: PlantInventory)
+ createFirstGeneration(): List<Individual>
- buildRandomIndividual(): Individual
Operators
Selection
- tournamentSize: int
- elitesPercentage: double
+ Selection(tournamentSize: int, elitesPercentage: double)
+ selectElites(oldPopulation: List<Individual>): List<Individual>
+ tournament(oldPopulation: List<Individual>): Individual
Mutation
- mutationProbability: double
- initialMutationStrength: double
- totalGenerations: int
- domain: Domain
+ Mutation(prob: double, strength: double, domain: Domain, totalGen: int)
+ mutate(individual: Individual, currentGeneration: int): void
- calculateAdaptiveStrength(gen: int): double
Crossover
- crossoverProbability: double
+ Crossover(crossoverProbability: double)
+ uniformCrossover(i1: Individual, i2: Individual): Individual
FitnessCalculator
- currentDomain: Domain
- distanceCalculator: DistanceCalculator
- quadraticStrategy: OverlapStrategy
- spatialStrategy: OverlapStrategy
- HASHING_THRESHOLD: int
- DOMAIN_PENALTY: double
- OVERLAP_WEIGHT: double
+ FitnessCalculator(domain: Domain, maxRadius: double)
+ getFitness(individual: Individual): double
MODEL
Individual
DOMAIN
Domain
Strategies
OverlapQuadratic
+ calculateOverlap(chromosomes: List<Point>, overlapWeight: double, distCalc: DistanceCalculator): double
<<interface>>
OverlapStrategy
+ calculateOverlap(chromosomes: List<Point>, overlapWeight: double, distCalc: DistanceCalculator): double
OverlapSpatial
- cellSize: double
+ OverlapSpatial(maxRadius: double)
+ calculateOverlap(chromosomes: List<Point>, overlapWeight: double, distCalc: DistanceCalculator): double
- getCellIndex(coordinate: double): int
- populateGrid(chromosomes: List<Point>): Map<Cell, List<Point>>
- calculatePenaltyForPoint(ref: Point, grid: Map<Cell, List<Point>>, overlapWeight: double, distcalc: DistanceCalculator): double
- processNeighbors(ref: Point, neighb: List<Point>, overlapWeight: double, distcalc: DistanceCalculator): double
- shouldSkipPair(p1: Point, p2: Point): boolean
<<record>>
OverlapSpatial.Cell
- i: int
- j: int
CONTROLLERS
EvolutionService
- domain: Domain
- inventory: PlantInventory
- fitnessCalculator: FitnessCalculator
- gammaPlays: Mutation
- mixer: Crossover
- selector: Selection
- populationFactory: PopulationFactory
+ EvolutionService(domain: Domain, inventory: PlantInventory)
+ executeEvolutionCycle(): Individual
+ isValidSolution(ind: Individual): boolean
EvolutionConsoleController
- view: EvolutionViewContract
- service: EvolutionService
- MAX_RETRY_ATTEMPTS: int
+ EvolutionConsoleController(view: EvolutionViewContract, service: EvolutionService)
+ runEvolution(): Individual
VIEW
<<interface>>
EvolutionViewContract
+ showEvolutionStart()
+ showRetryWarning(currentAttempt: int, maxAttempts: int, lastExecutionTimeSec: double)
+ showSuccess(attempt: int, executionTimeSec: double)
ConsoleEvolutionView
+ showEvolutionStart()
+ showRetryWarning(currentAttempt: int, maxAttempts: int, lastExecutionTimeSec: double)
+ showSuccess(attempt: int, executionTimeSec: double)
- printSingleSeparator()
DOMAIN
Domain
SHARED
UTILS
DistanceCalculator
RandomUtils


GENETIC ALGORITHM
MODEL
Individual
Point
DOMAIN PACKAGE
MODEL
<<interface>>
Domain
+ isPointOutside(x: double, y: double): boolean
+ isValidIndividual(individual: Individual): boolean
+ getBoundingBox(): Rectangle2D
CircleDomain
- radius: double
- boundingBox: Rectangle2D
+ CircleDomain(radius: double)
+ isPointOutside(x: double, y: double): boolean
+ isValidIndividual(individual: Individual): boolean
+ getBoundingBox(): Rectangle2D
+ toString(): String
RectangleDomain
- width: double
- height: double
- halfWidth: double
- halfHeight: double
- boundingBox: Rectangle2D
+ RectangleDomain(width: double, height: double)
+ isPointOutside(x: double, y: double): boolean
+ isValidIndividual(individual: Individual): boolean
+ getBoundingBox(): Rectangle2D
+ toString(): String
<<create>>
<<create>>
<<Use>>
<<singleton>>
DomainFactory
- DomainFactory()
+ getInstance(): DomainFactory
+ createDomain(type: DomainType, params: Map<String, Double>): Domain
- validateParameters(type: DomainType, params: Map<String, Double>)
<<static utility>>
DomainFactory.FactoryHolder
- INSTANCE: DomainFactory
- <clinit>()
<<enumeration>>
DomainType
RECTANGLE
CIRCLE
SQUARE
ELLIPSE
RIGHT ANGLED TRIANGLE
FRAME
ANNULUS
- menuId: int
- displayName: String
- requiredParameters: List<String>
- DomainType(menuId: int, name: String, params: List<String>)
+ getMenuId(): int
+ getDisplayName(): String
+ getRequiredParameters(): List<String>
+ fromMenuId(id: int): Optional<DomainType>
+ toString(): String
CONTROLLERS
DomainConsoleController
- view: DomainViewContract
- service: DomainService
+ DomainConsoleController(view: DomainViewContract, service: DomainService)
+ runDomainCreationWizard(): Optional<Domain>
DomainService
+ getAvailableDomainTypes(): List<DomainType>
+ createDomain(type: DomainType, params: Map<String, Double>): Domain
+ calculateMaxValidRadius(domain: Domain): double
VIEW
<<interface>>
DomainViewContract
+ showAvailableDomains(types: List<DomainType>)
+ askForDomainType(types: List<DomainType>): Optional<DomainType>
+ askForParameters(type: DomainType): Map<String, Double>
+ showSuccessMessage()
+ showErrorMessage(message: String)
ConsoleDomainView
- scanner: Scanner
+ ConsoleDomainView(scanner: Scanner)
+ showAvailableDomains(types: List<DomainType>)
+ askForDomainType(types: List<DomainType>): Optional<DomainType>
+ askForParameters(type: DomainType): Map<String, Double>
+ showSuccessMessage()
+ showErrorMessage(message: String)
- readPositiveDouble(paramName: String): double
- printDoubleSeparator()
- printSingleSeparator()



3.2 DESIGN PATTERNS

In the development of Agroplanner, several standard GoF (Gang of Four) design patterns were implemented to ensure code modularity, scalability, and maintainability.
The following list highlights the most significant implementations within the system architecture:

SINGLETON (Creational)

- **Context**: This pattern ensures that a class has only one instance and provides a global point of access to it.

- **Implementation**: It is utilized in the DBConnection class. Using the "Bill Pugh" initialization technique (Holder Class), the system ensures a thread-safe and lazy-loaded connection to the H2 database. This prevents resource conflicts and

guarantees that the persistence layer operates on a unified session context throughout the application lifecycle.

ABSTRACT FACTORY (Creational)

- **Context**: This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- **Implementation**: It is the core of the persistence layer. The abstract class AgroPersistenceFactory defines the contract, while concrete implementations (SqlPersistenceFactory, FilePersistenceFactory, MemoryPersistenceFactory) generate the specific family of DAOs (UserDao, InventoryDao, etc.). This allows the system to switch the entire storage mechanism (from Database to File System or Volatile memory) seamlessly at boot time without altering the business logic.

BUILDER (Creational)

- **Context**: This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

- **Implementation**: It is used in the User class within the Access module. Since the User entity contains multiple attributes (some mandatory, some optional), the Builder pattern avoids "telescoping constructors" and improves code readability when instantiating new user profiles during registration or retrieval.

STRATEGY (Behavioral)

- **Context**: This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.

- **Implementation**: It is critical in the gasystem (Genetic Algorithm). The OverlapStrategy interface allows the engine to swap between OverlapQuadratic (precise, $O(N^2)$) and OverlapSpatial (optimized spatial hashing, $O(N)$) based on the population complexity. It is also used in the DomainSystem to handle different terrain shapes (Circle, Rectangle...) polymorphically.

TEMPLATE METHOD (Behavioral)

- **Context**: This pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses.

- **Implementation**: It is applied in the ExportSystem. The abstract exporter class defines the standard workflow for file generation (export routine, resolve file

path), while the concrete subclasses implement the specific logic for writing .csv, .xlsx, .txt ... formats.

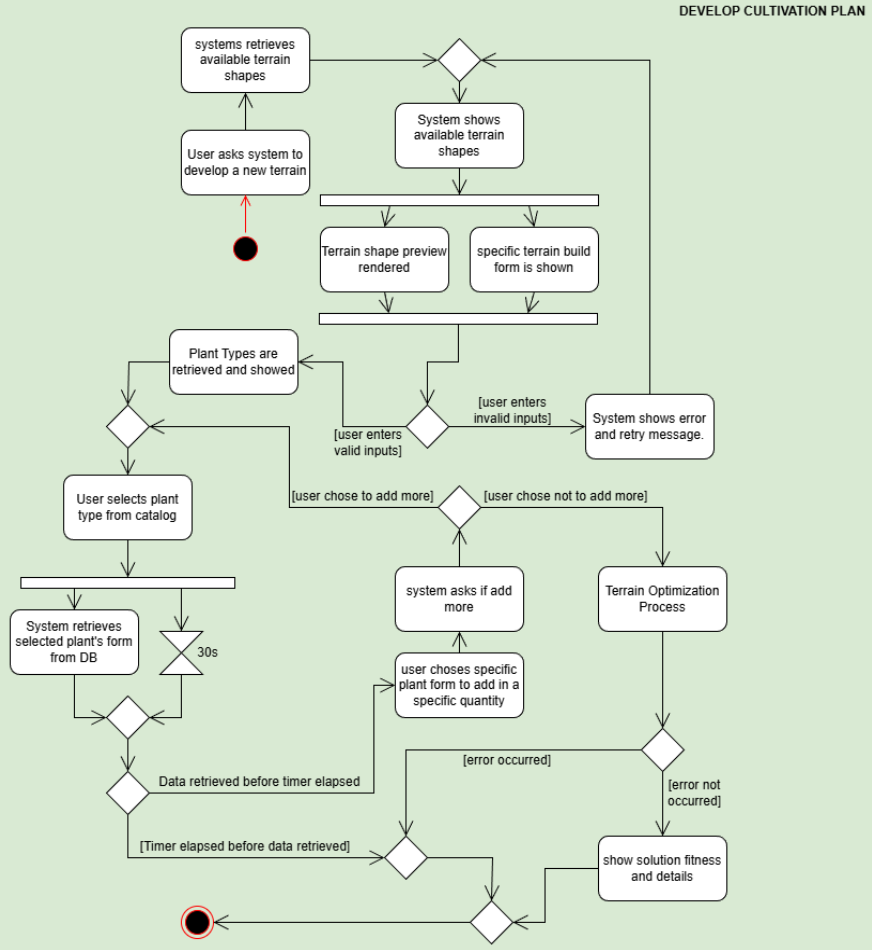
ARCHITECTURAL PATTERN: MVC (Model-View-Controller)

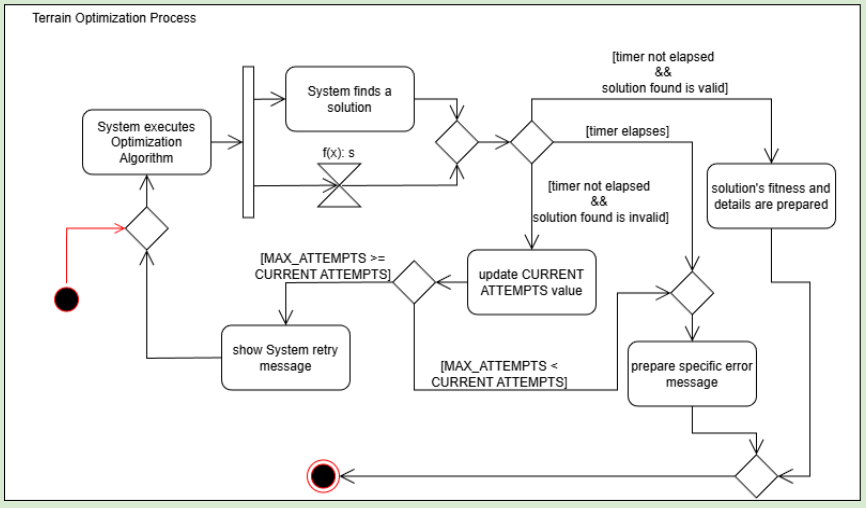
- **Context**: While not a GoF pattern, MVC is the architectural backbone of the system.

- **Implementation**: Almost every subsystem (Access, Inventory, Domain...) adopts this separation. A notable adaptation in Agroplanner is the Dual-Controller approach: a logic controller handles the business rules, while two distinct view controllers manage the specific interactions for the CLI and JavaFX interfaces, ensuring total decoupling between the core logic and the presentation layer.

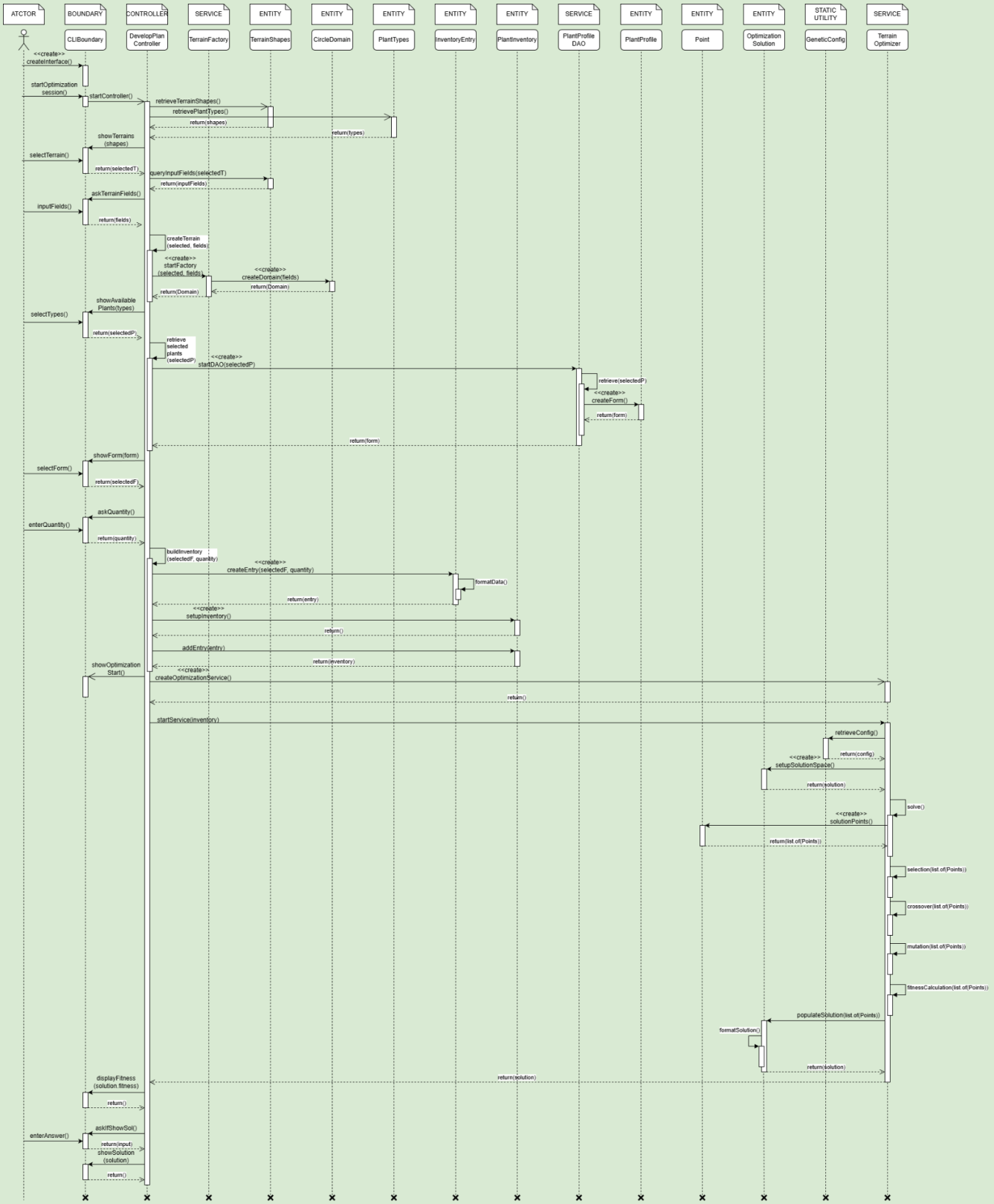
3.3 ACTIVITY DIAGRAM

The following activity diagram depicts the operational workflow of the '**Develop Cultivation Plan**' use case. It illustrates the sequence of actions required to generate an optimized layout, mapping the process from the initial geometric configuration and plant selection to the execution of the Genetic Algorithm and the final visualization of the results.





3.4 SEQUENCE DIAGRAM

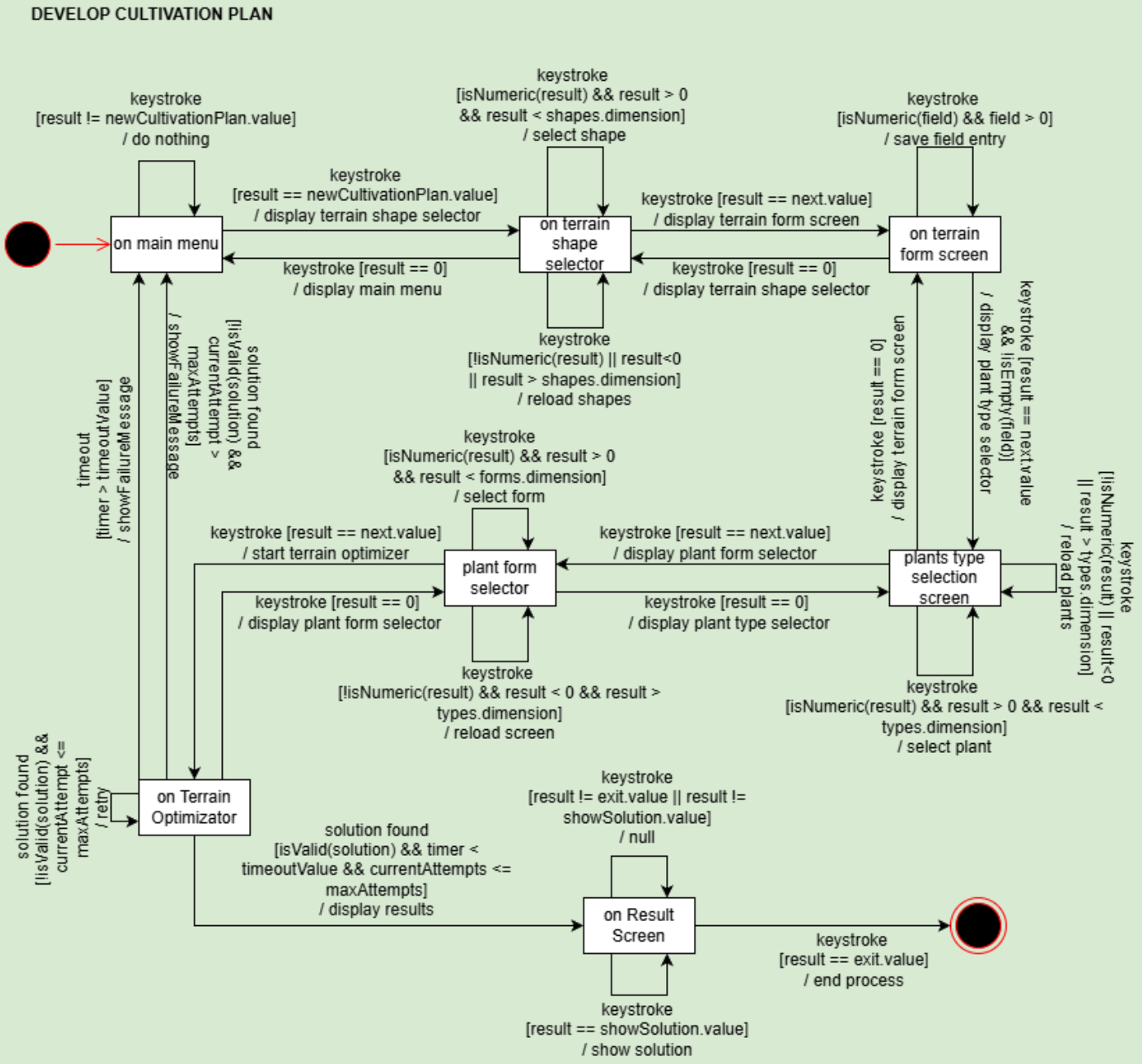


The sequence diagram presented above details the chronological interactions between objects during the execution of the 'Develop Cultivation Plan' use case.

Specifically modeled on the CLI implementation, it visualizes the control flow starting from the user's input, triggering the instantiation of domain entities via the Factory, and orchestrating the execution of the TerrainOptimizer. Finally, it depicts the synchronous return of the optimization results to the standard output.

3.5 STATE DIAGRAM

The following State Diagram illustrates the dynamic flow of the "Develop cultivation plan" use case. It depicts the sequence of states the system traverses during this specific process, highlighting the transitions triggered by user interactions and the corresponding logical responses.



4. TESTING

In the verification phase, a suite of unit tests was implemented using the JUnit framework to validate the correctness, robustness, and boundary behaviors of the system's critical components. The testing strategy focused on four key areas: geometric primitives, persistence logic, algorithmic core, and business rule enforcement.
[https://github.com/FeDevv/AGROPLANNER/tree/main/AGROPLANNER/src/test/java/org/agroplanner]

1. DistanceCalculatorTest
This class is responsible for validating the DistanceCalculator utility, which is fundamental for the entire spatial analysis.

- **shouldCalculateCorrectEuclideanDistance**: Verifies the mathematical accuracy of the Euclidean distance formula between two distinct points in the 2D plane.
- **shouldReturnZero_WhenPointsAreIdentical**: Tests the identity property of a metric space, ensuring that the distance between a point and itself is exactly zero.
- **shouldHandleNegativeCoordinates**: Validates the calculator's ability to handle points in different Cartesian quadrants, ensuring that negative coordinate values do not corrupt the magnitude result.

2. PersistenceServiceTest
This class targets the PersistenceService, specifically focusing on the correct implementation of the Abstract Factory pattern.

- **shouldInitializeMemoryFactory_WhenConfiguredForMemory**: Verifies that the service correctly initializes the MemoryPersistenceFactory when the configuration explicitly requests the MEMORY strategy. This ensures that the internal dispatch logic correctly maps the configuration Enum to the appropriate concrete implementation of the factory.

3. OverlapQuadraticTest
This class validates the logic of the OverlapQuadratic strategy ($O(N^2)$), which is critical for fitness calculation.

- **shouldReturnZero_WhenListHasInsufficientPoints**: Based on combinatorial logic, the number of pairs C(n, 2) is zero for n < 2. This test confirms that the algorithm yields a zero penalty for empty or singleton lists, correctly

bypassing the inner comparison loop.

- **shouldDetectOverlap_WhenTwoPointsIntersect & shouldReturnZero_WhenPointsAreFarApart**: These tests validate the boolean logic of collision detection, ensuring positive penalties for intersecting entities and zero penalties for disjoint ones.

- **shouldAccumulatePenalties_ForMultipleOverlaps**: Validates the "Triangular Loop" aggregation logic. For a cluster of 3 overlapping points (A, B, C), it ensures the total penalty is the sum of the 3 distinct interactions (A-B + A-C + B-C).

4. DomainServiceTest
This class ensures the integrity of the DomainService and the enforcement of geometric constraints.

- **shouldCalculateCorrectMaxRadius_WhenDomainIsCreated**: Verifies that the service enforces geometric limits based on the Minimum Bounding Rectangle (MBR). It checks that the maximum permissible radius for a plant does not exceed half of the shortest dimension of the field ($R_{max} = min(w, h)/2$).

- **shouldThrowException_WhenCreatingDomainWithMissingParams**: Validates the "Deep Protection" mechanism, confirming that the service correctly propagates exceptions from the lower layers when invalid configurations (e.g., missing mandatory parameters) are attempted.

---

5. EXCEPTIONS

In the Handling Exceptions part, the system adopts a robust, two-tiered approach to ensure stability and provide meaningful feedback to the user.
[https://github.com/FeDevv/AGROPLANNER/tree/main/AGROPLANNER/src/main/java/org/agroplanner/shared/exceptions]

1. **Custom Exception Hierarchy**
The first mechanism is a dedicated hierarchy of custom exceptions rooted in the abstract base class AgroPlannerException. By extending RuntimeException, this class allows for unchecked exception propagation. This design supports a flexible handling strategy where low-level technology-specific errors (like SQLException or IOException) are caught within try-catch blocks at the boundary layers and

translated into meaningful, domain-specific exceptions (e.g., mapping a database unique constraint violation to a DuplicateUserException). This preserves the abstraction level of the business logic, decoupling it from the underlying infrastructure.

The concrete implementations cover specific failure scenarios within the application:

- **DataPersistenceException**: Thrown when an operation involving the persistence layer fails. It acts as a wrapper for lower-level errors encountered during data retrieval or storage.
- **DomainConstraintException**: Thrown when an operation results in a state that violates the business rules or physical constraints of a domain entity.
- **DuplicateUserException**: Thrown when an attempt is made to register or update an entity using a unique identifier that is already present in the system repository.
- **EvolutionTimeoutException**: Thrown when the computational evolution process of the Genetic Algorithm exceeds its allocated time limit.
- **ExportException**: Thrown when the data export process fails, typically wrapping I/O errors occurring during file generation.
- **InvalidInputException**: Thrown when input data fails to meet format requirements or validation preconditions.
- **MaxAttemptsExceededException**: Thrown when an iterative algorithm reaches its maximum allowed iterations without converging to a result.
- **UIInitializationException**: Thrown when a critical error occurs during the initialization of the JavaFX interface.

2. **Defensive Programming & Safety Mechanisms**
The second type of handling involves defensive programming techniques to prevent exceptions before they occur. Instead of relying solely on try-catch blocks, the system validates the state of objects early in the process. A prime example is found within the domainsystem: concrete implementations (such as geometric shapes) perform rigorous preliminary checks on data validity. They ensure that dimensions are non-negative and enforce structural constraints (e.g., verifying

that an inner dimension is strictly smaller than an outer dimension in complex shapes) before the domain object is even instantiated.

---

6. DATABASES

The system implements a flexible persistence layer capable of utilizing three distinct storage strategies: an H2 Relational Database, a Flat File System, and a Volatile Memory storage.
[https://github.com/FeDevv/AGROPLANNER/tree/main/AGROPLANNER/src/main/java/org/agroplanner/persistence]

6.1 H2

The primary persistence mechanism uses an embedded H2 Database Engine. This allows for SQL-based data management without requiring a standalone server installation. The **db.properties** file externalizes the connection parameters and authentication credentials required to establish the database session. The internal schema is defined by the following relational tables:

- **users**: Stores authentication credentials (encrypted), personal details, and role assignments (USER, AGRONOMIST, ADMIN).
- **plant_varieties**: Contains the botanical catalog, including plant names, specific variety details, and spacing constraints defined by Agronomists.
- **domains**: Stores the geometric definitions of the terrains created by users (dimensions, shapes, and types).
- **solutions & solution_items**: Implements a one-to-many relationship. The solutions table holds metadata (fitness score, timestamp, owner), while solution_items stores the specific coordinates of every plant within that solution.

6.2 FILE SYSTEM

The file system strategy persists data into structured text and CSV files located in the system's root data/ directory. The file distribution is as follows:

1. **data/users.txt** A text-based repository containing the serialized records of registered users.

2. **data/plants.txt** A text-based repository containing the list of available plant varieties and their attributes.

3. **data/domains.txt** A text-based repository containing the definitions of saved terrain configurations.

4. **data/solutions/** Unlike other entities, solutions are stored as individual files within this specific directory. The naming convention follows a dynamic Regex pattern: sol_u(\d+)_(.*)_(\d{8}_\d{6})\.csv.
    - sol_u(\d+): Represents the User ID who owns the solution.
    - (.*): Represents the custom name of the project/domain.
    - (\d{8}_\d{6}): Represents the timestamp (YYYYMMDD_HHMMSS) of creation.
    - .csv: The file format is Comma Separated Values, allowing for easy export and external auditing.

6.3 VOLATILE

The system also supports a "Demo Mode" where persistence is handled exclusively in RAM using Java Collections (ArrayList). In this mode, no files are created, and no database connections are established. All data is lost upon application termination.

7. SONAR CLOUD

To complement the unit testing phase and ensure the architectural integrity of the codebase, the project underwent rigorous **Static Code Analysis** using the **SonarCloud** platform.
This automated inspection tool was employed to continuously monitor the code quality throughout the development cycle. Specifically, SonarCloud was utilized to detect and mitigate:

- **Bugs**: Coding errors that could lead to runtime failures or unexpected behaviors.
- **Vulnerabilities**: Security weaknesses and "Security Hotspots" that could compromise system data.
- **Code Smells**: Maintainability issues, complex logic, and anti-patterns that increase technical debt.

The analysis validates that the functions are not only operational but also adhere to Clean Code principles and industry standards. The comprehensive quality report and detailed results are accessible at the following link: https://sonarcloud.io/organizations/fedevv/projects

8. PROJECT INFORMATION & CREDITS

Video Demonstration
To provide a comprehensive visual overview, a short video demonstration illustrating the system's functionality and workflow has been uploaded to the following address: [https://github.com/FeDevv/AGROPLANNER/tree/main/video] (The file is available in both .mpeg and .mov formats).

Project Information

- **Developer:** Federico Bonucci - 0323567
- Institution: Università degli Studi di Roma "Tor Vergata"
- **Repository:** https://github.com/FeDevv/AGROPLANNER/tree/main
- **Tools Used:** Java, JavaFX, H2 Database, Maven
- Contacts: ricobon03@gmail.com , federico.bonucci@students.uniroma2.eu