## **Detailed Breakdown of Game Development Tasks**

Our project, developed for a class assignment, is a game prototype built using Java Swing. The game involves several entities, including a farmer, corn plants, crows, and scarecrows.

We have divided the project into several specific tasks, each encompassing a set of functionalities:

* **Task 1: Class Creation**

Implementing the basic classes for the game entities: farmer, corn, crows, and scarecrows. Each class defines the basic attributes and methods for these entities. For example, the Corn class might include properties for the corn's position and whether it has been eaten. The Farùer, Crow and Scarecrow classes, on the other hand, would include properties for their positions, movement, and interactions with other entities.

* **Task 2: Frame Creation**

Creating the main window frame for our game using the JFrame class provided by Java Swing. The frame serves as the container for all other graphical components in our game, such as the game panel and other UI elements. We've set the size of the frame to match the dimensions of our game world, set the frame to close when the user clicks the close button, and made the frame visible at the end of the main method.

* **Task 3: Game Engine Creation**

Creating the GameEngine class, the heart of our game. It manages the game's state and interactions between different game entities. It has methods to add and remove units, get all units, get specific types of units (farmer, crows, corns, scarecrows), and generate a crow. The GameEngine class is responsible for updating the game state and triggering the necessary actions based on the game's rules.

* **Task 4: Farmer Spawn and Display**

Creating a JPanel canvas where a farmer spawns in the middle. The farmer is represented as a solid red dot. The farmer's position is managed by the GameEngine class, and it's displayed on the JPanel canvas. The farmer's position can change over time, depending on the user's actions.

* **Task 5: Farmer Selection**

Developping a GamePanel that detects mouse clicks. A left-click on the farmer selects the farmer. To indicate selection, the red dot representing the farmer turns green. A subsequent click on the farmer deselects it. The selection circle, always centered on the farmer's coordinates, follows the farmer's movements. This allows the user to interact with the farmer and control its actions.

* **Task 6: Farmer Movement**

When the farmer is selected, a right-click on another location on the canvas sets that location as the destination. The farmer then moves directly towards this destination. We've drawn the path of movement with a dashed line. A green dashed line connects the farmer's current position to the destination, updating as the farmer moves. This gives the user visual feedback on the farmer's movement.

* **Task 7: Corn Spawn**

The canvas randomly spawns 'n' corn plants with a minimum spacing of 100 in the x direction and 100 in the y direction. The corn plants are represented as solid yellow dots. They are smaller in volume than the farmer and are represented by yellow circles. Unlike the farmer, the corn does not move and is generated only once, stored in an ArrayList. This creates the game's environment and provides the crows with their objective.

* **Task 8: Scarecrow Creation and display of remaining effective time**

Creating a Scarecrow class and added a scarecrow to the game. The scarecrow is placed at a specific location on the canvas. The scarecrow serves as a deterrent for the crows, preventing them from reaching the corn. Left-clicking on a scarecrow will show the remaining effective time of the scarecrow. Also, implementing methods for picking up and placing down scarecrows, with checks for proximity to the farmer.

* **Task 9: Crow Generation**

Creating a CrowThread class that generates crows at fixed intervals. It uses a Timer to schedule the generation of crows. This creates a dynamic game environment where crows continuously enter the game, providing a continuous challenge for the player.

* **Task 10: Crow Movement**

Creating a Crow class that represents a crow in the game. It extends the MovingUnits class, inheriting its movement capabilities. The Crow class has several properties and methods that define its behavior in the game, such as locating corn, farmer, and scarecrow, fleeing from threats, moving towards a destination, updating its scared state, eating corn, and leaving the game. This makes the crows intelligent entities that react to the game's state.

* **Task 11: Harvesting Corn**

Implementing a feature where the user can click to harvest corn. The system checks the position of the farmer relative to the corn. If they are sufficiently close, it checks the maturity state of the corn. Depending on the maturity state, points are added to the player's score or a message is returned that the corn cannot be harvested because it is not mature.

* **Task 12: Testing**

Including tests for the basic functionalities to ensure the correct operation of our game. This includes unit tests for the individual classes and integration tests for the game as a whole. Testing is crucial to ensure that our game works as expected and helps us identify and fix any bugs or issues.

## **Class Architecture Overview**

* **GameEngine Class**

The GameEngine class is the heart of the game's logic. It manages the game state, including all the units in the game (Farmer, Crow, Corn, and Scarecrow). It also provides methods to add and remove units, get the list of units, and update the game state.

**Key Components:**

- units: A list of all units in the game. This list is used to keep track of all the game entities and their states.

**Functionalities:**

* addUnit(Units unit): Adds a new unit to the game. This method is used to introduce new entities into the game world.
* removeUnit(Units unit): Removes a unit from the game. This method is used to remove entities from the game world when they are no longer needed.
* getUnits(): Returns the list of all units in the game. This method is used to access the game state and perform operations on all units.
* getFarmer(): Returns the farmer unit.
* getCorns(): Returns the list of corns.
* getCrows(): Returns the list of crows.
* getScarecrows(): Returns the list of scarecrows.

The GameEngine class is integral to the game's user interface and gameplay. It manages the game state and is responsible for executing game commands, such as adding and removing units, updating the game state, and handling game logic. The GameEngine class works in conjunction with the Farmer, Crow, Corn, and Scarecrow classes to create a dynamic and interactive environment for the game.

* **Units Class**

The Units class is the foundational class for all entities in our game. It was designed with the idea of encapsulating the basic properties that all units share, thereby promoting code reusability and reducing complexity. The primary attribute of this class is a position in the game world, represented by a Point object. This class is designed to be extended by more specific unit types, each with their unique behaviors and attributes, following the principles of object-oriented programming.

**Key Components:**

* Position (Point position): Each unit has a position attribute, a Point object that stores the x and y coordinates of the unit in the game world. This attribute is used in various parts of the game logic, such as rendering the unit on the screen and for collision detection with other units or game elements.
* GameEngine (GameEngine gameEngine): The game engine is a crucial component that allows the unit to interact with the game world. It provides access to the game's state and other units, enabling the unit to make decisions based on the current game situation. This attribute is used in various methods within the class and its subclasses to access and manipulate the game state.

**Functionalities:**

* Constructor (Units(Point position, GameEngine gameEngine)): Initializes a unit at a specified location. The position parameter sets the initial position of the unit. The gameEngine parameter is used to interact with the game world. This constructor is called when a new instance of a unit is created, setting up its initial state.
* **MovingUnits Class**

MovingUnits is an abstract class that extends Units, adding the capability for units to move towards a destination. This class was designed to be the parent of all mobile entities in our game, such as the Farmer and Crow. It introduces an additional attribute, destination, and an abstract method, move(), which must be implemented by subclasses to define their specific movement behaviors. This design allows for a high level of flexibility and customization in defining how different types of units move.

**Key Components:**

* destination (Point destination): This attribute marks the target location to which a unit is moving. The destination can be set dynamically during the game, such as through player input. This attribute is used in the move() method to determine the direction and distance of movement.
* speed (double speed): This attribute determines the speed at which a unit moves. It is used in the calculation of the unit's movement vector, which is then used to update the unit's position. This attribute is used in the move() method to control the rate of movement.

**Functionalities:**

* Constructor (MovingUnits(Point position, GameEngine gameEngine)): Sets the initial position and default destination of the moving unit to its starting location. This ensures that a unit has a defined state upon instantiation. This constructor is called when a new instance of a moving unit is created.
* getDestination(): Returns the current destination of the unit. This method facilitates the implementation of movement logic in subclasses, allowing them to query the target location as they calculate the movement path.
* setDestination(Point destination): Allows for the dynamic updating of a unit's destination. This is key for interactive gameplay, enabling player commands to direct unit movement.
* move(): An abstract method that defines how a unit moves towards its destination. The actual implementation is left to subclasses, allowing for diverse movement behaviors tailored to each unit type. The move method uses the speed attribute to determine the rate at which the unit moves towards its destination.
* **Farmer Class**

The Farmer class is a key component of our game, representing the player-controlled character. It extends the MovingUnits class, inheriting its movement capabilities and attributes. This class introduces additional attributes and methods specific to the Farmer's role in the game.

**Key Components:**

* isSelected: A boolean flag indicating whether the Farmer is currently selected by the player. This attribute is used in the game logic to enable or disable certain actions based on the Farmer's selection state.
* moveTimer: A javax.swing.Timer object used to control the movement of the Farmer. It facilitates smooth, continuous movement towards a destination point and allows for real-time updates of the Farmer's position on the game canvas.

**Functionalities:**

* Constructor (Farmer(Point position, GameEngine gameEngine)): Initializes a Farmer at a specified location and sets up its initial state. It also sets up the moveTimer with a specified delay of 10 milliseconds. This timer repeatedly calls the move method, which contains the logic for moving the Farmer towards the destination.
* move(): Overrides the move method from MovingUnits. It calculates the difference in x and y coordinates between the Farmer's current position and the destination. It then determines the distance to the destination and calculates unit vectors to represent the direction of movement. By multiplying these unit vectors by a predefined speed, we obtain the actual movement vector for each timer tick. If the distance to the destination is less than or equal to the movement speed, the Farmer's position is directly set to the destination, ensuring precise arrival without overshooting.
* getPosition() and setPosition(Point position): Accessor and mutator for the Farmer's position, allowing for position retrieval and updates.
* isSelected() and setSelected(boolean isSelected): Methods to get and set the selection state of the Farmer.
* getDestination() and setDestination(Point destination): Accessor and mutator for the Farmer's destination, facilitating the dynamic updating of the target location based on player input.
* isClickWithinCircle(int x, int y, int radius): A utility method to determine if a mouse click is within a certain radius of the Farmer, aiding in the selection and deselection logic.
* startMoveTimer() and stopMoveTimer(): Methods to control the execution of the moveTimer, enabling or disabling the Farmer's movement.

**Movement Logic :**

The Farmer class extends the MovingUnits class, which provides the basic movement logic. The movement logic for the Farmer is encapsulated in the `move()` method.

In the `move()` method, the Farmer calculates the direction vector to its destination. This is done by subtracting the Farmer's current position from the destination position. The resulting vector points from the Farmer's current position towards the destination.

The direction vector is then normalized, which means it's scaled to have a length of 1. This is done by dividing the direction vector by its length. The normalized direction vector represents the direction of movement, but not the speed.

The speed of the Farmer is represented by the `speed` attribute. The normalized direction vector is multiplied by the speed to get the actual movement vector. This vector represents both the direction and distance the Farmer should move in one step.

Finally, the Farmer's position is updated by adding the movement vector to the current position. This moves the Farmer one step closer to its destination.

The Farmer class is integral to the game's user interface and gameplay. It represents the player-controlled character and is responsible for executing player commands, such as moving to a specified location. The Farmer class works in conjunction with the GameEngine to create a dynamic and interactive environment for the game.

* **Crow Class**

The Crow class extends the MovingUnits class, inheriting its movement capabilities and attributes. It introduces additional attributes and methods specific to the Crow's role in our game. This class was designed with the specific behaviors and attributes of a Crow in mind, such as being scared and eating corn.

**Key Components:**

* safetyDistance: An integer representing the distance the crow needs to keep from threats to not get scared. This attribute is used in the updateState() method to manage the Crow's scared state.
* isScared: A boolean flag indicating whether the Crow is currently scared. This attribute is used in the move() method to determine the Crow's behavior.
* remainingTime: An integer representing the remaining time for the Crow to be scared. This attribute is used in the updateState() method to manage the Crow's scared state.
* eatingTime: An integer representing the time the Crow spends eating a Corn. This attribute is used in the eatCorn() method to control the Crow's corn-eating behavior.

**Functionalities:**

* Constructor (Crow(Point position, GameEngine gameEngine)): Initializes a Crow at a specified location and sets up its initial state. This constructor is called when a new Crow is created.
* locateCorn(): Returns the nearest Corn to the Crow. This method is used in the move() method to determine the Crow's destination when it's not scared.
* locateFarmer(): Returns the Farmer in the game. This method is used in the updateState() method to check if the Crow should be scared.
* locateScarecrow(): Returns the nearest Scarecrow to the Crow. This method is used in the updateState() method to check if the Crow should be scared.
* move(): Overrides the move method from MovingUnits. It defines how a Crow moves towards its destination, taking into account its scared state and the locations of Corns, the Farmer, and Scarecrows. The Crow has three different behaviors depending on its state: if it's scared, it flees; if it's not scared and there's corn, it goes to the corn; if there's no corn, it leaves the game.
* updateState(): Updates the Crow's scared state based on the proximity of the Farmer and Scarecrows. This method is called in the move() method to update the Crow's state before deciding its behavior.
* eatCorn(Corn nearestCorn): Allows the Crow to eat a Corn if it is close enough. The Crow has a counter that increments each time it's close to a Corn, and when the counter reaches a certain value, the Corn is removed from the game. This method is called in the move() method when the Crow reaches a Corn.

**Movement Logic :**

The Crow class also extends the MovingUnits class, so it inherits the same basic movement logic as the Farmer class. However, the Crow class overrides the `move()` method to implement its own unique movement behaviors.

The Crow's movement logic is more complex because it has to take into account its current state and the locations of other game entities. The Crow has three different behaviors depending on its state:

1. If the Crow is scared, it calls the flee() method. This method calculates a destination that is far away from the nearest threat (either the Farmer or the nearest Scarecrow). The Crow then moves towards this destination using the same basic movement logic as the Farmer.
2. If the Crow is not scared and there's corn, it calls the goLookForCorn() method. This method sets the destination to the location of the nearest Corn. The Crow then moves towards this destination.
3. If there's no corn, the Crow calls the leave() method. This method sets the destination to the nearest corner of the game world, and the Crow moves towards this destination.

In each of these cases, the Crow uses the basic movement logic inherited from the MovingUnits class to move towards its destination. The difference lies in how the destination is determined.

The Crow class is integral to the game's user interface and gameplay. It represents a dynamic entity within the game world that can move, eat Corn, and get scared by the Farmer or Scarecrow. The Crow class works in conjunction with the GameEngine to create a dynamic and interactive environment for the game.

* **Corn Class**

The Corn class extends the base Units class, inheriting the position attribute and related methods. It represents a static entity within our game world that the Farmer and Crows can interact with. This class was designed with the simplicity of a Corn's behavior in mind, as it doesn't move or interact actively with other units.

**Key Components:**

- No additional components are introduced in the Corn class. The Corn class relies on the attributes and methods inherited from the Units class.

**Functionalities:**

- Constructor (Corn(Point position, GameEngine gameEngine)): Initializes a Corn at a specified location. This constructor is called when a new Corn is created.

The Corn class is integral to the game's user interface and gameplay. It represents a static entity within the game world that can be eaten by Crows. The Corn class works in conjunction with the GameEngine to create a dynamic and interactive environment for the game.

* **Scarecrow Class**

The Scarecrow class extends the Units class, representing a static entity within the game world that can scare Crows away. This class was designed with the simplicity of a Scarecrow's behavior in mind, as it doesn't move or interact actively with other units.

**Key Components:**

- efficiencyTime: An integer representing the remaining time for the Scarecrow to be effective. This attribute is used in the game logic to determine if the Scarecrow can still scare Crows away.

**Functionalities:**

- Constructor (Scarecrow(Point position, GameEngine gameEngine)): Initializes a Scarecrow at a specified location. This constructor is called when a new Scarecrow is created.

- getEfficiencyTime(): Returns the remaining time for the Scarecrow to be effective. This method is used in the game logic to check if the Scarecrow can still scare Crows away.

The Scarecrow class is integral to the game's user interface and gameplay. It represents a static entity within the game world that can scare Crows away. The Scarecrow class works in conjunction with the GameEngine to create a dynamic and interactive environment for the game.

* **GamePanel Class**

The GamePanel class is responsible for rendering our game's visual elements onto the screen. It extends the JPanel class and overrides the `paintComponent` method to draw the game's units (Farmer, Crow, Corn, and Scarecrow) onto the panel. It also contains a reference to the game engine, which it uses to access our game's state and units. This class was designed to handle the visual representation of our game, updating the game's display based on the current state of the game.

**Key Components:**

- GameEngine (GameEngine gameEngine): The game engine is a crucial component that allows the GamePanel to interact with our game world. It provides access to our game's state and other units, enabling the GamePanel to render the correct visual representation of our game.

**Functionalities:**

- Constructor (GamePanel(GameEngine gameEngine)): Initializes a GamePanel with a specified game engine. The gameEngine parameter is used to interact with our game world.

- paintComponent(Graphics g): Overrides the `paintComponent` method from JPanel. It is responsible for rendering our game's visual elements onto the screen. It achieves this by iterating over our game's units and drawing each one at its current position. The method utilizes the `Graphics2D.drawImage` method to render the images of the units onto the panel.

* **CrowMovementThread Class**

The CrowMovementThread class extends the Thread class and is responsible for controlling the movement of the Crows in the game. It works in conjunction with the GameEngine and GamePanel to update the game state and redraw the game screen.

**Key Components:**

- gamePanel: A reference to the GamePanel. This is used to redraw the game screen after each update.

- gameEngine: A reference to the GameEngine. This is used to access and update the game state.

**Functionalities:**

- Constructor (CrowMovementThread(GamePanel gamePanel, GameEngine gameEngine)): Initializes a CrowMovementThread with a specified game panel and game engine. The gamePanel and gameEngine parameters are used to interact with the game screen and game state, respectively.

- run(): Overrides the run method from Thread. It is responsible for updating the game state and redrawing the game screen in each iteration of the game loop.

The CrowMovementThread class is integral to the game's user interface and gameplay. It is responsible for controlling the movement of the Crows and updating the game screen. The CrowMovementThread class works in conjunction with the GameEngine and GamePanel to create a dynamic and interactive environment for the game.

* **CrowGenerationThread Class**

The CrowGenerationThread class extends the Thread class and is responsible for controlling the creation of new Crows in the game. It works in conjunction with the GameEngine to update the game state.

**Key Components:**

- gameEngine: A reference to the GameEngine. This is used to access and update the game state.

**Functionalities**:

- Constructor (CrowGenerationThread(GameEngine gameEngine)): Initializes a CrowThread with a specified game engine. The gameEngine parameter is used to interact with the game state.

- run(): Overrides the run method from Thread. It is responsible for creating new Crows and adding them to the game state in each iteration of the game loop.

The CrowGenerationThread class is integral to the game's user interface and gameplay. It is responsible for controlling the creation of new Crows and updating the game state. The CrowGenerationThread class works in conjunction with the GameEngine to create a dynamic and interactive environment for the game.

* **Conclusion**

The design of these classes allows for a flexible and extensible structure within our game. The Units class serves as a base for all game entities, with the MovingUnits class extending it to incorporate movement capabilities. Specific game entities like the Crow and Corn further extend these classes to add behaviors and attributes unique to them. The GamePanel class acts as the glue, rendering our game entities onto the screen. This design enables easy addition of new unit types and behaviors in the future. Embracing inheritance and encapsulation, this design promotes code reusability and reduces complexity, thus enhancing maintainability and extensibility. Moreover, the design adheres to the principles of object-oriented programming, making it a stellar example of structuring a game in Java.

## **Challenges we faced**

Throughout the development of our game, we encountered several challenges that tested our problem-solving skills and understanding of the Java programming language :

1. **Complexity of movement logic**: Implementing the movement logic for the Farmer and Crow classes was a complex task. We had to ensure that the movement was smooth and responsive, which required a deep understanding of vectors and game physics. Additionally, the Crow's movement was influenced by several factors, such as its scared state and the locations of Corns, the Farmer, and Scarecrows, which added to the complexity.
2. **Concurrency issues**: We used multiple threads to control the game's logic and render the game's visual elements. Managing these threads and ensuring that they worked together seamlessly was a challenge. We had to carefully synchronize the threads to avoid race conditions and ensure that the game state was consistently updated.
3. **User interaction** : Creating a user interface that was intuitive and responsive was a challenge. We had to implement mouse listeners to allow the player to interact with the game, and we had to ensure that these interactions were processed correctly to update the game state.
4. **Code organization and reusability**: With multiple classes and a growing codebase, maintaining clean and reusable code was a challenge. We had to carefully design our classes and methods to promote code reusability and maintainability.

We also faced several challenges not only at the individual level but also as a team. Here are some of the team-related difficulties we encountered:

1. **Integration of individual components**: Each team member was responsible for different parts of the game, such as the game logic, user interface, and game entities. Integrating these individual components into a cohesive whole was a significant challenge. We had to ensure that all parts worked together seamlessly, which required careful planning and coordination.
2. **Version control and code merge conflicts**: We used a version control system to manage our codebase. However, we often encountered merge conflicts when multiple team members were working on the same parts of the code. Resolving these conflicts without losing any changes was a complex task that required a good understanding of the codebase and the changes made by each team member.
3. **Communication and coordination**: As with any team project, communication was crucial. We had to ensure that all team members were on the same page regarding the game's design and implementation. Coordinating our efforts and managing our time effectively was also a challenge, especially when working remotely.

Despite these challenges, we were able to learn and grow as a team. We improved our communication and coordination skills, gained experience in using version control systems, and learned how to integrate individual components into a cohesive whole. These challenges also provided us with the opportunity to improve our problem-solving skills and learn how to work effectively as a team.

## **Conclusion**

Throughout the course of these developments, we have discovered many key lessons and best practices, especially since three members out of all four of them had never touched Java before :

* The significance of a well-defined and hierarchical class structure, which promotes code reusability and simplifies maintenance.
* The efficacy of employing timers and event listeners to craft responsive and interactive game elements.
* The imperative of devising efficient algorithms for game mechanics, such as the distribution of game objects, which directly influences the quality of gameplay.

In conclusion, the merge of the Farmer, GamePanel, and GameFrame classes exemplifies a meticulous application of object-oriented programming principles, providing a robust foundation for further development. The thoughtful consideration of game mechanics, coupled with efficient coding practices, establishes a high benchmark for the subsequent phases of the project. As we persist in augmenting the game's features and refining its gameplay, these foundational elements will act as a beacon, ensuring that we strike a balance between innovation, functionality, and user experience.

Throughout this project, we have gained substantial knowledge about object-oriented programming, particularly the practical application of inheritance and encapsulation. Implementing features such as the Farmer's movement and the dynamic generation of Corns has deepened our understanding of event-driven programming and GUI development with Java Swing. A significant lesson learned was how algorithmic thinking aids in efficiently tackling game development challenges, such as object placement and movement.

Nevertheless, challenges persist. Optimizing performance as the game's complexity escalates is a continuous task, necessitating further research and experimentation. Moreover, refining user interactions to create a more engaging gameplay experience is a crucial area that requires improvement. As we move forward, discussing these issues within our team and seeking guidance from our instructor will be vital steps. This collaborative approach not only aids in resolving the current challenges but also enhances our collective learning and problem-solving capabilities.