SOIL MOISTURE MONITORING SYSTEM (SMMS)

FMS (SE-313)

Dr.Mustafa Latif

Group ID: B8

SOFTWARE ENGINEERING

Rajja Farhan(SE-21058)
Usman Siddiqui (SE-21075)

SOIL MOISTURE MONITORING SYSTEM (SMMS)

Scope:

The Soil Moisture Monitoring System (SMMS) is an advanced, automated system engineered to meticulously manage and maintain the soil moisture within critical thresholds in various agricultural settings, including greenhouses, open fields, and urban farming installations. The system's primary function is to ensure that the soil remains within an optimal moisture range, which is crucial for the health and yield of crops. To achieve this, the SMMS is equipped with a network of high-precision **soil moisture sensors** that provide real-time data on the soil's current moisture content.

Upon installation, the user configures the system by setting the desired soil moisture range. This range is typically expressed as a percentage of volumetric water content (VWC), where 0% VWC means completely dry soil and 100% VWC indicates full water saturation. For most agricultural applications, the optimal VWC range is between **20%** and **60%**, where 20% may represent the lower threshold (minimum desired moisture level) and 60% the upper threshold (maximum desired moisture level).

When the Soil Moisture Monitoring System (SMMS) detects that the soil moisture has fallen below the 20% threshold, it sends a signal to the irrigation control unit to initiate watering(INCREASE)), thereby increasing the VWC towards the desired level. If the system detects a moisture content above the 60% threshold , it signals the irrigation system to cease watering (DECREASE), preventing waterlogging and potential root damage due to excessive moisture. Should the soil moisture be within the 20-60% range, the system will send a 'maintain' signal, indicating that the current moisture level is adequate and no adjustment is needed.

The system's user interface allows for remote monitoring and control, granting the user the ability to adjust settings and receive alerts on the go. Alerts can be configured to notify the user when the soil moisture levels are outside of the predetermined range or when system anomalies are detected.

The SMMS represents a significant step forward in precision agriculture, offering a robust solution for optimizing water usage, enhancing crop health, and ultimately increasing agricultural efficiency and productivity.

4+1 VIEW MODEL

The 4+1 architectural view model is a framework designed by Philippe Kruchten to describe the architecture of software systems using five concurrent views. Each view addresses a specific set of concerns for different stakeholders, such as end-users, developers, system engineers, and project managers. For the Soil Moisture Monitoring System (SMMS), these views help to provide a comprehensive understanding of the system's architecture from multiple perspectives.

The SMMS is designed to automate and optimize the irrigation process based on soil moisture levels. It aims to enhance agricultural efficiency, conserve water, and ensure crop health. The 4+1 view model for SMMS breaks down the system into logical, development, process, and physical perspectives, with use cases illustrating real-world scenarios.

Logical View:

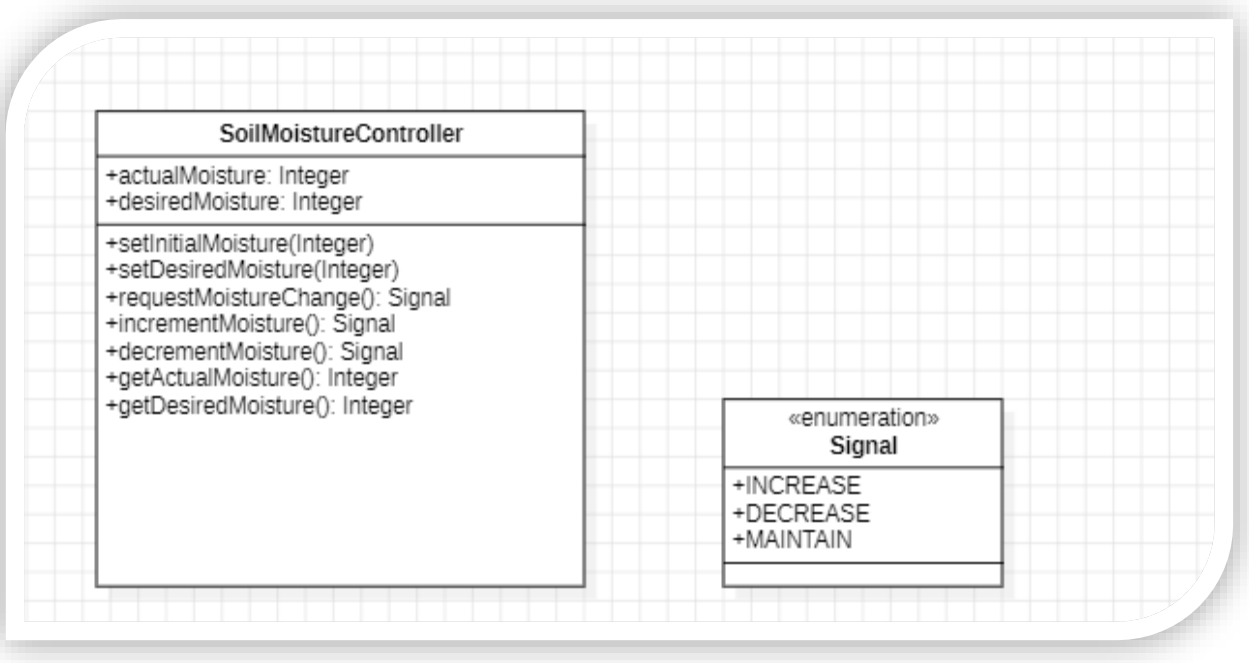
The Logical View of the SMMS primarily focuses on the functionality visible to the end-users. It encompasses the key classes and objects that interact within the system, detailing how the system performs its tasks. The logical view is typically represented by class diagrams and sequence diagrams that illustrate the system's structure and behavior.

For SMMS, the class diagram you've provided includes the following elements:

- SoilMoistureController: This is the central class responsible for the system's logic. It maintains two main properties: **actualMoisture** and **desiredMoisture**, which represent the current and target soil moisture levels, respectively. The methods within this class include:
  - **setInitialMoisture(Integer)**: Sets the baseline moisture level when the system is first configured.
  - **setDesiredMoisture(Integer)**: Updates the target moisture level as needed.
  - **requestMoistureChange()**: Issues a command when a change in moisture level is required, returning a Signal type.
  - **incrementMoisture()**: Increases the moisture level, usually by activating the irrigation system.
  - **decrementMoisture():** Decreases the moisture level, typically by deactivating the irrigation system.
  - **getActualMoisture():** Retrieves the current moisture level from the sensors.
  - **getDesiredMoisture()**: Returns the target moisture level.

- Signal Enumeration: Defines the types of signals that the SoilMoistureController can issue: **INCREASE**, **DECREASE**, and **MAINTAIN**. These correspond to the actions the system should take when adjusting the soil moisture level.

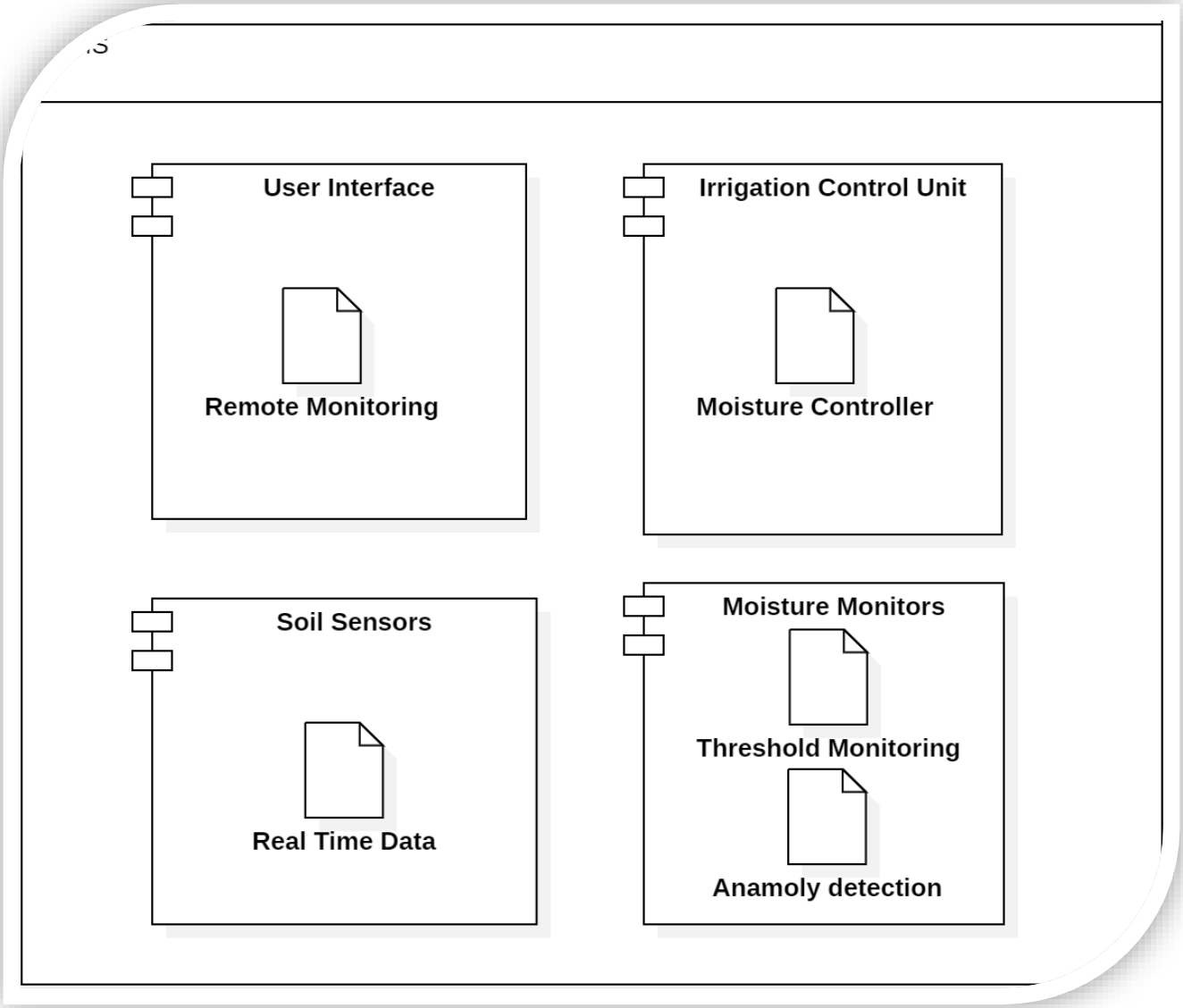
In the context of SMMS, the Logical View with its class diagram provides a blueprint of the system's structure. It shows how the SoilMoistureController interacts with the sensor data to manage the irrigation process effectively. The Signal enumeration is used by the SoilMoistureController to communicate with other system components, like the irrigation control module, to take appropriate action based on the soil's moisture data.



2. Development View:

This view illustrates the system's software architecture, detailing how it is built and decomposed into modules.

- Sensor Integration Module: Responsible for collecting data from individual sensors and forwarding it to the SoilMoistureController.
- Moisture Analysis Module: Contains algorithms that analyze the moisture data and determine the adjustment needed, if any.
- Irrigation Command Module: Translates the SoilMoistureController's signals into actionable commands for the irrigation hardware.



3. Process View:

The Process View in the 4+1 architectural model focuses on the dynamic aspects of the system. It illustrates the system's processes, how they interact, and how data flows through the system at runtime. The sequence diagram you provided is a perfect tool to demonstrate the Process View of the SMMS. Let's walk through your sequence diagram and explain each part in the context of the SMMS.

Process View Description for SMMS:

Initialization:

- The sequence begins with the Central Control Unit initiating the monitoring cycle by sending a start message to the soil moisture sensors.
- The sensors respond with a returns ready status, indicating that they are operational and ready to measure the soil's moisture level.

Data Collection:

- The Central Control Unit instructs the sensors to collect moisture reading.
- Once the reading is taken, the sensors send the actual moisture reading back to the SoilMoistureController.

Decision Process:

- The SoilMoistureController receives the moisture data. It then enters a decision-making sequence (depicted as an alt or alternative fragment in the diagram), which involves evaluating the actual moisture against the desired moisture levels.

Condition Checks and Actions:

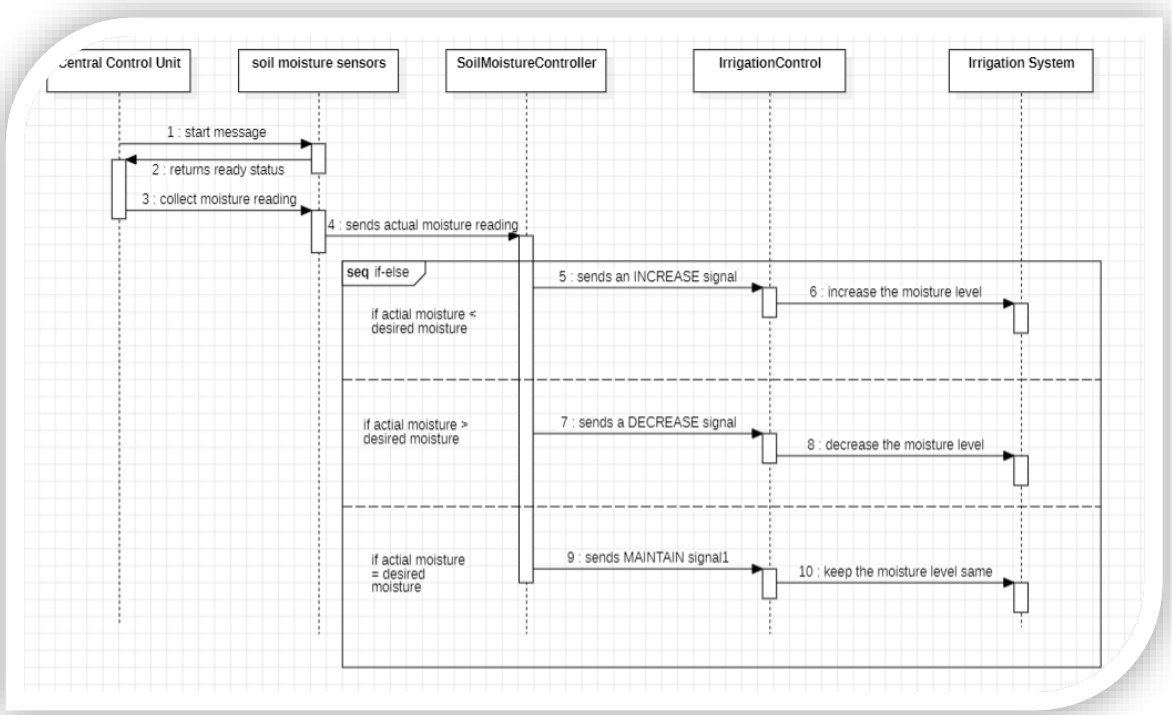
- If the actual moisture is less than desired moisture, the SoilMoistureController sends an INCREASE signal to the IrrigationControl. This signifies that the soil is too dry and needs watering.
- Upon receiving this signal, the IrrigationControl instructs the Irrigation System to increase the moisture level by initiating the irrigation process.

Adjustment Process:

- If the actual moisture is greater than desired moisture, indicating excess water, the SoilMoistureController sends a DECREASE signal to the IrrigationControl.
- The IrrigationControl then sends a command to the Irrigation System to decrease the moisture level, typically by stopping or reducing watering.

Maintenance Process:

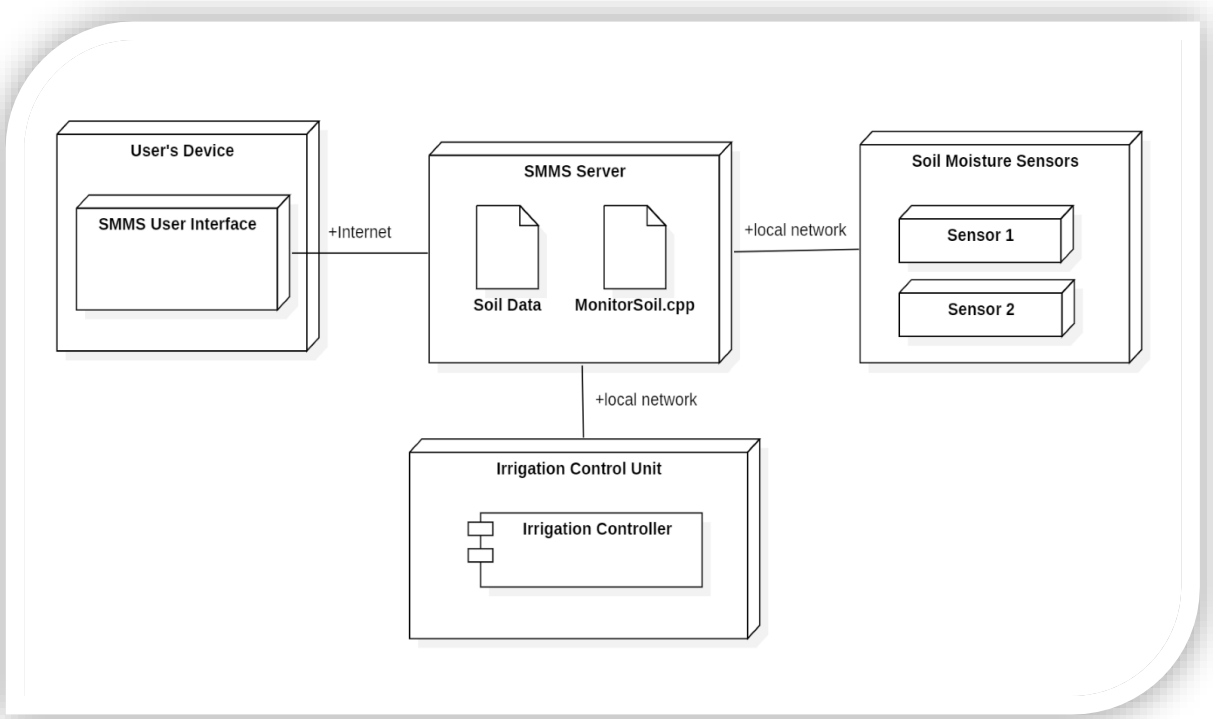
- If the actual moisture equals the desired moisture, which means the soil moisture is at an optimal level, the SoilMoistureController sends a MAINTAIN signal.The IrrigationControl receives this signal and maintains the current state, so the Irrigation System continues to keep the moisture level same, with no change to the irrigation schedule.



4. Physical View:

This view describes the system from a technical standpoint, looking at hardware and network topology.

- Sensor Nodes: The physical hardware for soil moisture sensors, which are distributed strategically across the agricultural area and connected wirelessly to the central system.
- Central Control Unit: A robust physical unit or embedded system that hosts the SoilMoistureController and runs the software components necessary for decision-making.
- Irrigation System: Comprises various physical components like valves, sprinklers, or drip lines that can be manipulated to adjust soil moisture levels.



+1: Scenarios (Use Cases):

The +1 in the 4+1 architectural view model refers to scenarios or use cases which illustrate how the system interacts with users and other systems. These scenarios validate the architecture by demonstrating it in action. Your use case diagram shows the interactions between two types of actors, the User and the System Admin, with the system. Let's describe the scenarios that correspond to these use cases:

Initial Configuration Scenario:

- The User interacts with the system to Set desired moisture range. This use case is initiated when the system is first set up. The user accesses the User Interface provided by the Central Control Unit to enter the optimal soil moisture thresholds that the SMMS should maintain.

Normal Operation Scenario:

- The View Current Soil Moisture use case describes a normal operation where the User can check the soil moisture level at any time. The system provides real-time data, ensuring the User is informed about the current state of the soil.
- The Receive Alerts use case is also part of normal operation. The User receives notifications if the soil moisture deviates from the set range or if the system detects any anomalies.

Drought Response Scenario:

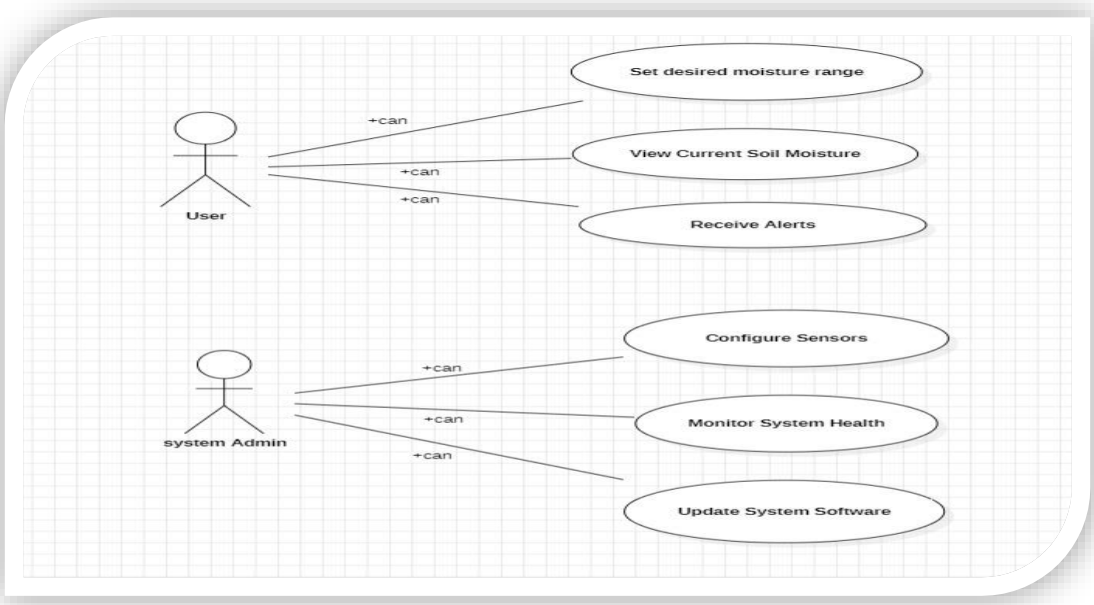
- This scenario comes into play when the soil moisture sensors, part of the Configure Sensors use case handled by the System Admin, detect a moisture level below the lower threshold. The SoilMoistureController then sends an INCREASE signal to the irrigation system to initiate watering, increasing the soil moisture.

Rainfall Response Scenario:

- Conversely, if the sensors detect moisture levels above the desired range, perhaps due to rainfall, the SoilMoistureController issues a DECREASE signal. The IrrigationControl interprets this signal and stops or reduces watering to prevent over-saturation of the soil.

System Maintenance Scenarios:

- The Monitor System Health and Update System Software use cases involve the System Admin who is responsible for ensuring that the SMMS is functioning correctly. Monitoring involves checking the status of sensors and the Central Control Unit, while updating may involve applying software patches or upgrades to enhance system performance or add new features.



These scenarios ensure the system architecture supports all required functions for both normal and exceptional environmental conditions.

VDM SPECIFICATION:

The following formal specification, crafted in the Vienna Development Method Specification Language (VDM-SL), captures the functional essence of the Soil Moisture Monitoring System (SMMS). It serves to maintain optimal soil moisture levels through precise monitoring and control mechanisms.

Our VDM-SL model articulates the types and values representing moisture data and control signals within agronomic constraints. The state of the system encapsulates actual and desired moisture levels, ensuring logical consistency through invariants. Functions and operations are defined to check ranges, update moisture levels, and facilitate user interaction. This specification is fundamental for the development and enhancement of the SMMS, providing a clear and structured definition of system operations.

```
types
  -- MoistureLevel represents a percentage of soil moisture.
  MoistureLevel = real];

  -- Signal is an enumerated type representing the possible control actions.
  Signal = <INCREASE> | <DECREASE> | <MAINTAIN>;
```

MoistureLevel: This is a real number type used to quantify the percentage of water present in the soil. As a real number, it allows for precise measurement and can capture fractional values, which are essential for accurate monitoring and control of soil moisture.
Signal: This is an enumerated type, which means it's a type that can take one of a limited set of values. The values defined for Signal correspond to the control actions the system can enact:

- <INCREASE>: This value represents a command to the system to increase the soil moisture level.
- <DECREASE>: This value indicates a command to reduce the soil moisture level.
- <MAINTAIN>: This command tells the system to maintain the current moisture level without making any changes.

```
values
  -- These values define the acceptable range for soil moisture.
  MIN_MOISTURE : MoistureLevel = 20;
  MAX_MOISTURE : MoistureLevel = 60;
```

Within the VDM specification for the Soil Moisture Monitoring System (SMMS), the values section establishes critical thresholds for the system's operation:

- **MIN_MOISTURE:** This constant represents the lower limit of the soil moisture range deemed acceptable for healthy plant growth. Set at 20, it reflects that soil moisture should not fall below 20%, as such levels may be too dry for most crops and could lead to water stress and hindered growth.
- **MAX_MOISTURE:** Similarly, this constant defines the upper threshold at 60, signifying that soil moisture should not exceed 60%. Excess moisture can lead to over-saturation, creating an anaerobic environment that could harm plant roots and lead to diseases.

```
state SMMS of
  -- The current and desired soil moisture levels.
  actualMoisture : [MoistureLevel]
  desiredMoisture : [MoistureLevel]

inv mk-SMMS(am, dm) △
  -- The invariant ensures that moisture levels are within the defined range or undefined.
  (am = nil V inRange(am)) Λ (dm = nil V inRange(dm))

init mk-SMMS(am, dm) △ am=NIL  Λ dm= NIL
  -- The initial state of the system with undefined moisture levels.

end
```

The state encapsulates the system's current status regarding soil moisture and is defined with two main attributes:

- **actualMoisture:** This attribute represents the current measured moisture level of the soil. It's an optional value, which means it can either hold a MoistureLevel value or be nil if the system has not yet obtained a measurement.
- **desiredMoisture**: This attribute denotes the target moisture level that the system aims to achieve and maintain. Similar to actualMoisture, it is optional and can be nil when a target has not been set.

The concept of invariants in the state definition is crucial for maintaining the system's integrity. In VDM, an invariant is a condition that must always hold true whenever the state is changed. The invariant for the SMMS, denoted as inv mk-SMMS, serves as a rule that enforces the actual and desired moisture levels to either be absent (nil) or fall within the acceptable moisture range defined by MIN_MOISTURE and MAX_MOISTURE. This ensures that any operations on the system's state do not result in moisture levels outside the specified limits.
The initial state, expressed by init mk-SMMS, is the system's configuration when it first becomes operational. Here, both actualMoisture and desiredMoisture are initialized to nil, signifying that the SMMS is in a state awaiting its first set of data. This starting point is important for the system's lifecycle, as it dictates that the SMMS should not act on undefined or uninitialized moisture data.
By setting out these elements, the state definition ensures that the SMMS starts in a known configuration

and that its operation is constrained within defined parameters, thereby avoiding erratic or unintended behavior.

Functions

-- Function to check if a moisture level is within the defined range.
inRange(ml :R) RESULT : B

pre TRUE
-- The pre-condition is always true, as the function does not have restrictions on its input.

post RESULT ⇔ (ml >= MIN_MOISTURE Λ ml <= MAX_MOISTURE);
-- The post-condition ensures the result is true if and only if ml is within the range.

In this function definition:

- The pre-condition is specified as TRUE, which means this function accepts any value of type MoistureLevel without constraints. In other words, it will process any input given to it.
- The post-condition uses the RESULT keyword (in VDM-SL, it's conventionally written in uppercase) to refer to the outcome of the function. The post-condition asserts that the function's result will be true if and only if the input moisture level (ml) lies within the bounds defined by MIN_MOISTURE and MAX_MOISTURE

operations
-- Operation to set the initial moisture level of the soil.

SetInitialMoisture(ml:R)
ext wr actualMoisture : [MoistureLevel]
-- The external clause specifying that this operation writes to actualMoisture.
pre inRange(ml) Λ actualMoisture = nil
-- The precondition checks if the input moisture level is within the range and if actualMoisture is not yet set.
post actualMoisture = ml
-- The postcondition ensures that actualMoisture is set to the input moisture level.

- External Clause (ext wr actualMoisture : [MoistureLevel]): This clause specifies that the operation has write access to the actualMoisture state variable. It indicates that SetInitialMoisture can modify the value of actualMoisture.
- Precondition (pre inRange(ml) and actualMoisture = nil): Before executing the operation, two conditions must be met. First, the input moisture level (ml) must be within the acceptable

range, as defined by the inRange function. Second, the actualMoisture must be uninitialized (i.e., nil). This ensures the operation is only executed when valid and necessary.

- Postcondition (post actualMoisture = ml): After the operation, the postcondition guarantees that the actualMoisture state variable is updated to reflect the input moisture level. This postcondition establishes the effect of the operation on the system's state.

```
-- Sets the desired moisture level for the soil.

SetDesiredMoisture(ml :R)
ext wr desiredMoisture : [MoistureLevel]
  -- The external clause specifying that this operation writes to desiredMoisture.
pre inRange(ml)
  -- The precondition checks if the input moisture level is within the range.
post desiredMoisture = ml
  -- The postcondition ensures that desiredMoisture is set to the input moisture level.

-- Requests a change in the moisture level.

RequestMoistureChange(ml:R) signalOut:signal
ext rd actualMoisture    : [MoistureLevel]
      wr desiredMoisture:  [MoistureLevel]
  -- The external clause specifying that this operation reads actualMoisture and writes to
desiredMoisture.
pre  inRange(ml) Λ  actualMoisture ≠ nil
  -- The precondition checks that actualMoisture is not nil.
post desiredMoisture = ml Λ
   (actualMoisture < desiredMoisture)  Λ signalOut=<INCREASE> |
   (actualMoisture > desiredMoisture) Λ  signalOut= <DECREASE> |
   (actualMoisture = desiredMoisture) Λ signalOut= <MAINTAIN>
```

- SetDesiredMoisture: This operation allows setting the desired moisture level for the soil. The precondition checks if the input moisture level (ml) falls within the acceptable range. The postcondition ensures that the desiredMoisture in the system state is updated to this new value.
- RequestMoistureChange: This operation analyzes the current soil moisture level and requests a change if necessary. It reads the actualMoisture and writes to controlSignal. The precondition ensures that actualMoisture is defined. The postcondition uses a case analysis to set the controlSignal to <INCREASE>, <DECREASE>, or <MAINTAIN>, depending on whether the actualMoisture is below, above, or equal to the desiredMoisture, respectively.

```
-- Increments the moisture level by a predefined step.
IncrementMoisture() signalOut:signal
ext rd desiredMoisture : [MoistureLevel]
```

wr actualMoisture : [MoistureLevel]
pre actualMoisture ≠ nil and desiredMoisture ≠ nil and actualMoisture < desiredMoisture
post actualMoisture = actualMoisture + 1 Λ
(actualMoisture < desiredMoisture Λ signalOut= <INCREASE> V
actualMoisture = desiredMoisture Λ signalOut= <MAINTAIN>);

-- Decrements the moisture level by a predefined step.
DecrementMoisture() signalOut :signal
ext rd desiredMoisture : [MoistureLevel]
wr actualMoisture : [MoistureLevel]
pre actualMoisture ≠ nil and desiredMoisture ≠ nil and actualMoisture > desiredMoisture
post actualMoisture = actualMoisture - 1 and
(actualMoisture > desiredMoisture => RESULT = <DECREASE> or
actualMoisture <= desiredMoisture => RESULT = <MAINTAIN>);

- IncrementMoisture: This operation increases the actualMoisture by one unit if it is below the MAX_MOISTURE level. The signalOut is set to <INCREASE> if the increment happens, and to <MAINTAIN> otherwise. The precondition ensures that the operation is only executed if actualMoisture and desiredMoisture are set and actualMoisture is below desiredMoisture. The postcondition confirms the increase in moisture level and sets the appropriate signal.

- DecrementMoisture: Similar to IncrementMoisture, this operation decreases the actualMoisture by one unit, but only if it is above the MIN_MOISTURE level. The signalOut reflects whether a decrease occurs (<DECREASE>) or the moisture level remains the same (<MAINTAIN>). The precondition and postcondition mirror those in IncrementMoisture, adapted for the decrement logic.

-- Retrieves the current actual moisture level.
GetActualMoisture: () RESULT : [R]
ext rd actualMoisture : [MoistureLevel]
pre TRUE
-- The pre-condition is always true as there are no restrictions
post RESULT = actualMoisture
-- The post-condition ensures the returned value (RESULT) is equal to the actual moisture level.

-- Retrieves the desired moisture level.
GetDesiredMoisture() RESULT : [R]
ext rd desiredMoisture : [MoistureLevel]

pre TRUE
-- The pre-condition is always true
post RESULT = desiredMoisture
-- The post-condition ensures the returned value (RESULT) is equal to the desired moisture level.

- GetActualMoisture: This operation is designed to retrieve the current actual moisture level (actualMoisture) from the system state. The external clause (ext rd actualMoisture : [MoistureLevel]) indicates that it only requires read access to actualMoisture. The pre-condition is always true, meaning this operation can be called at any time. The post-condition confirms that the result returned by this operation (RESULT) is the current actual moisture level.
- GetDesiredMoisture: Similarly, this operation fetches the desired moisture level (desiredMoisture). It also only reads from the state variable (desiredMoisture), and the pre-condition is universally true, allowing unrestricted access. The post-condition ensures that the operation's result matches the desiredMoisture in the system state.

types
-- MoistureLevel represents a percentage of soil moisture.
MoistureLevel = real];
-- Signal is an enumerated type representing the possible control actions.
Signal = <INCREASE> | <DECREASE> | <MAINTAIN>;

values
-- These values define the acceptable range for soil moisture.
MIN_MOISTURE : MoistureLevel = 20;
MAX_MOISTURE : MoistureLevel = 60;

State SMMS of
-- The current and desired soil moisture levels.
actualMoisture : [MoistureLevel]
desiredMoisture : [MoistureLevel]

inv mk-SMMS(am, dm) $\triangle$
-- The invariant ensures that moisture levels are within the defined range or undefined.
(am = nil V inRange(am)) Λ (dm = nil V inRange(dm))
init mk-SMMS(am, dm) $\triangle$ am=NIL Λ dm= NIL
-- The initial state of the system with undefined moisture levels.
End

Functions
-- Function to check if a moisture level is within the defined range.
inRange(ml :R) RESULT : B
pre TRUE
-- The pre-condition is always true, as the function does not have restrictions on its input.

post RESULT ⇔ (ml >= MIN_MOISTURE Λ ml <= MAX_MOISTURE);
-- The post-condition ensures the result is true if and only if ml is within the range.

operations
-- Operation to set the initial moisture level of the soil.

SetInitialMoisture(ml:R)
ext wr actualMoisture : [MoistureLevel]
-- The external clause specifying that this operation writes to actualMoisture.
pre inRange(ml) Λ actualMoisture = nil
-- The precondition checks if the input moisture level is within the range and if actualMoisture is not yet set.
post actualMoisture = ml
-- The postcondition ensures that actualMoisture is set to the input moisture level.
-- Sets the desired moisture level for the soil.

SetDesiredMoisture(ml :R)
ext wr desiredMoisture : [MoistureLevel]
-- The external clause specifying that this operation writes to desiredMoisture.
pre inRange(ml)
-- The precondition checks if the input moisture level is within the range.
post desiredMoisture = ml
-- The postcondition ensures that desiredMoisture is set to the input moisture level.

-- Requests a change in the moisture level.

RequestMoistureChange(ml:R) signalOut:signal
ext rd actualMoisture : [MoistureLevel]
wr desiredMoisture: [MoistureLevel]
-- The external clause specifying that this operation reads actualMoisture and writes to desiredMoisture.
pre inRange(ml) Λ actualMoisture ≠ nil
-- The precondition checks that actualMoisture is not nil.
post desiredMoisture = ml Λ
(actualMoisture < desiredMoisture) Λ signalOut=<INCREASE> |
(actualMoisture > desiredMoisture) Λ signalOut= <DECREASE> |
(actualMoisture = desiredMoisture) Λ signalOut= <MAINTAIN>

-- Increments the moisture level by a predefined step.
IncrementMoisture() signalOut:signal
ext rd desiredMoisture : [MoistureLevel]
wr actualMoisture : [MoistureLevel]
pre actualMoisture ≠ nil Λ desiredMoisture ≠ nil Λ actualMoisture < desiredMoisture
post actualMoisture = actualMoisture + 1 Λ
(actualMoisture < desiredMoisture Λ signalOut= <INCREASE> V
actualMoisture = desiredMoisture Λ signalOut= <MAINTAIN>);

-- Decrements the moisture level by a predefined step.
DecrementMoisture() signalOut :signal
ext rd desiredMoisture : [MoistureLevel]
wr actualMoisture : [MoistureLevel]
pre actualMoisture ≠ nil Λ desiredMoisture ≠ nil Λ actualMoisture > desiredMoisture
post actualMoisture = actualMoisture - 1 Λ
(actualMoisture > desiredMoisture Λ signalOut= <DECREASE> V
actualMoisture = desiredMoisture Λ signalOut = <MAINTAIN>);

-- Retrieves the current actual moisture level.
GetActualMoisture() RESULT : [R]
ext rd actualMoisture : [MoistureLevel]
pre TRUE
post RESULT = actualMoisture
-- Retrieves the desired moisture level.
GetDesiredMoisture() RESULT : [R]
ext rd desiredMoisture : [MoistureLevel]
pre TRUE
post RESULT = desiredMoisture

Translation of VDM-SL Specification to C++ Implementation

The transition from a formal VDM specification to an executable C++ codebase is a crucial step in software development. This guide presents a side-by-side comparison of the two, showcasing how abstract VDM concepts are concretely implemented in C++. Each segment of VDM-SL's rigorous notation for a soil moisture control system is translated into practical C++ components, bridging the gap between formal definition and application

VDM-SL	C++ Code
types MoistureLevel = real;	private: optional<double> actualMoisture; optional<double> desiredMoisture;

In VDM-SL, MoistureLevel is a type that can represent any real number, which is typically used for continuous values. In the context of the soil moisture system, this allows for precise tracking of moisture levels. The C++ equivalent uses std::optional<double> to capture this notion, providing the ability to represent a real number that can also be in an undefined state. This is particularly useful for cases where moisture levels have not yet been set or measured, thus reflecting the VDM notion of a potentially absent (nil) value.

VDM-SL	C++ Code
values MIN_MOISTURE : MoistureLevel = 20; MAX_MOISTURE : MoistureLevel = 60;	const double MIN_MOISTURE = 20.0; const double MAX_MOISTURE = 60.0;

In the VDM-SL specification, MIN_MOISTURE and MAX_MOISTURE are constants that define the acceptable range for soil moisture, ensuring that moisture levels stay within a specific threshold. This is critical for maintaining the desired environment for soil or plant growth. The C++ implementation mirrors this by defining these values as constants with the const keyword, signifying that once set, these values should not be altered during the program's execution. This enforces a clear boundary on the moisture levels that the SoilMoistureController will need to maintain, which is essential for the reliability of the system

VDM-SL	C++ Code
SetDesiredMoisture(ml :R) **ext wr** desiredMoisture : [MoistureLevel]	private: std::optional<double> actualMoisture; std::optional<double> desiredMoisture;

The VDM-SL function SetDesiredMoisture sets a desired moisture level if it falls within an acceptable range. The ext wr keyword indicates that this function can modify the state of desiredMoisture. In C++, this is implemented as a method within the SoilMoistureController class. The function checks if the input ml is within the range using the inRange method before assigning the value to the desiredMoisture member variable, which is of type std::optional<double> to handle potentially undefined (uninitialized) states.

VDM-SL	C++ Code

inv mk-SMMS(am, dm) ≜
(am = nil V inRange(am)) Λ (dm = nil V inRange(dm))
 	
init mk-SMMS(am, dm) ≜ am=NIL Λ dm= NIL
	SoilMoistureController() : actualMoisture(nullopt), desiredMoisture(nullopt) {}
	void SetInitialMoisture(double ml) {
	checkPrecondition(ml, "SetInitialMoisture");actualMoisture = ml;}
	// Function to check the precondition and throw an error if out of range
	void checkPrecondition(double ml, const string& operation) {
	if (!inRange(ml)) {
	throw out_of_range("Moisture level out of range for operation: " + operation);}}

The invariant in VDM-SL is a condition that must always be true for all instances of the system's state. In C++, this concept is not explicitly declared as an invariant but is enforced throughout the class methods. The constructor initializes actualMoisture and desiredMoisture to std::nullopt,

reflecting the VDM-SL specification where the moisture levels can start as undefined. Throughout the C++ class, methods like SetDesiredMoisture and SetInitialMoisture ensure these invariants by checking the range before setting values.

VDM-SL	C++ Code

inRange(ml :R) RESULT : B
pre TRUE
post RESULT ⇔ (ml >= MIN_MOISTURE Λ ml <= MAX_MOISTURE);
	bool inRange(double ml) {
	return ml >= MIN_MOISTURE && ml <= MAX_MOISTURE;
	}

The inRange function in VDM-SL is a boolean function that takes a real number ml and checks whether it falls within the specified moisture range. The C++ equivalent is a method that returns true if ml is between MIN_MOISTURE and MAX_MOISTURE, inclusive.

This function is crucial for maintaining the system's integrity, ensuring that moisture levels remain within the defined thresholds.

VDM-SL	C++ Code

SetInitialMoisture(ml:R)
ext wr actualMoisture : [MoistureLevel] **pre** inRange(ml) Λ actualMoisture = nil
post actualMoisture = ml
 	
SetDesiredMoisture(ml :R)
ext wr desiredMoisture : [MoistureLevel]
pre inRange(ml)
post desiredMoisture = ml
	void SetInitialMoisture(double ml) {
	checkPrecondition(ml, "SetInitialMoisture");
	actualMoisture = ml;
	}
	
	void SetDesiredMoisture(double ml) {
	checkPrecondition(ml, "SetDesiredMoisture");
	desiredMoisture = ml;}

In VDM-SL, SetInitialMoisture is an operation that sets the actualMoisture level, provided it is within the valid range and not already set. The precondition and postcondition in VDM-SL enforce these requirements. Translated to C++, the method sets the actualMoisture only if it is currently std::nullopt (representing nil in VDM-SL) and if the input ml passes the inRange check. This ensures the initial moisture level is set correctly and only once.

VDM-SL	C++ Code

RequestMoistureChange(ml:R)
signalOut:signal
ext rd actualMoisture : [MoistureLevel]
wr desiredMoisture: [MoistureLevel]
	Signal RequestMoistureChange() {
	if (actualMoisture && desiredMoisture) {
	if (*actualMoisture <

pre inRange(ml) Λ actualMoisture ≠ nil
post desiredMoisture = ml Λ
(actualMoisture < desiredMoisture) Λ
signalOut=<INCREASE> |
(actualMoisture > desiredMoisture) Λ
signalOut= <DECREASE> |
(actualMoisture = desiredMoisture) Λ
signalOut= <MAINTAIN>
	*desiredMoisture) return
	Signal::INCREASE;
	if (*actualMoisture >
	*desiredMoisture) return
	Signal::DECREASE;
	}
	return Signal::MAINTAIN;
	}

This operation in VDM-SL takes a moisture level ml and outputs a signal based on the comparison with actualMoisture. The preconditions ensure ml is within the range, and actualMoisture is not undefined. The C++ method mimics this logic: it reads the actualMoisture, compares it with ml, and returns an appropriate Signal enumeration value

VDM-SL	C++ Code
IncrementMoisture() signalOut:signal **ext rd** desiredMoisture : [MoistureLevel] **wr** actualMoisture : [MoistureLevel] **pre** actualMoisture ≠ nil Λ desiredMoisture ≠ nil Λ actualMoisture < desiredMoisture **post** actualMoisture = actualMoisture + 1 Λ (actualMoisture < desiredMoisture Λ signalOut= <INCREASE> V actualMoisture = desiredMoisture Λ signalOut= <MAINTAIN>);	Signal IncrementMoisture() { if (actualMoisture && desiredMoisture && *actualMoisture < *desiredMoisture && *actualMoisture < MAX_MOISTURE) { *actualMoisture += 1; return Signal::INCREASE; } return Signal::MAINTAIN; }

The IncrementMoisture operation in VDM-SL is designed to increase the actualMoisture by a fixed step, outputting a signal to indicate the action taken. In C++, this is translated into a method that increments actualMoisture by 1, only if it is defined, less than desiredMoisture, and below the MAX_MOISTURE limit. The method returns an INCREASE signal if the increment occurs, otherwise, a MAINTAIN signal is returned, adhering to the VDM-SL's postcondition logic.

VDM-SL	C++ Code

DecrementMoisture() signalOut :signal
ext rd desiredMoisture : [MoistureLevel]
wr actualMoisture : [MoistureLevel]
pre actualMoisture ≠ nil Λ desiredMoisture ≠ nil Λ actualMoisture > desiredMoisture
post actualMoisture = actualMoisture - 1 Λ
(actualMoisture > desiredMoisture Λ
signalOut= <DECREASE> V
actualMoisture = desiredMoisture Λ
signalOut = <MAINTAIN>);
	Signal DecrementMoisture() {
	if (actualMoisture && desiredMoisture
	&& *actualMoisture > *desiredMoisture) {
	checkPrecondition(*actualMoisture -
	1, "DecrementMoisture");
	*actualMoisture -= 1;
	return Signal::DECREASE;
	}
	return Signal::MAINTAIN;
	}

The DecrementMoisture function in VDM-SL decrements the actualMoisture level when it is above the desiredMoisture level and above the minimum threshold. The C++ method implements this logic by decrementing actualMoisture by 1 under the specified conditions, returning a DECREASE signal if successful. If the conditions are not met, it defaults to returning MAINTAIN, similar to the VDM-SL postconditions.

VDM-SL	C++ Code

GetActualMoisture() RESULT : [R]
ext rd actualMoisture : [MoistureLevel]
pre TRUE
post RESULT = actualMoisture
 	
GetDesiredMoisture() RESULT : [R]
ext rd desiredMoisture : [MoistureLevel]
pre TRUE
post RESULT = desiredMoisture
	optional<double> GetActualMoisture() {
	return actualMoisture;
	}
	
	optional<double> GetDesiredMoisture() {
	return desiredMoisture;
	}
	};

In VDM-SL, **GetActualMoisture** simply retrieves the current value of **actualMoisture**, which may be undefined. The corresponding C++ method returns the value of the **actualMoisture** member variable, using std::optional<double> to account for the possibility that the moisture level has not been set. This directly reflects the VDM-SL's optional return type and its **postcondition.**

C++ CODE:

```
#include <iostream>
#include <optional>
#include <stdexcept>
#include <string>
using namespace std;


// Enumerated type representing possible control actions
enum class Signal {
   INCREASE,
   DECREASE,
   MAINTAIN
};
```

```
// Convert Signal to a human-readable string
string signalToString(Signal signal) {
    switch (signal) {
        case Signal::INCREASE: return "INCREASE";
        case Signal::DECREASE: return "DECREASE";
        case Signal::MAINTAIN: return "MAINTAIN";
        default: return "UNKNOWN";
    }
}

// SoilMoistureController class definition
class SoilMoistureController {
private:
    optional<double> actualMoisture;
    optional<double> desiredMoisture;

    const double MIN_MOISTURE = 20.0;
    const double MAX_MOISTURE = 60.0;

    bool inRange(double ml) {
        return ml >= MIN_MOISTURE && ml <= MAX_MOISTURE;
    }

    // Function to check the precondition and throw an error if out of range
    void checkPrecondition(double ml, const string& operation) {
        if (!inRange(ml)) {
            throw out_of_range("Moisture level out of range for operation: " + operation);
        }
    }

public:
    SoilMoistureController() : actualMoisture(nullopt), desiredMoisture(nullopt) {}

    void SetInitialMoisture(double ml) {
        checkPrecondition(ml, "SetInitialMoisture");
        actualMoisture = ml;
    }

    void SetDesiredMoisture(double ml) {
        checkPrecondition(ml, "SetDesiredMoisture");
        desiredMoisture = ml;
    }
```

```
    Signal RequestMoistureChange() {
        if (actualMoisture && desiredMoisture) {
            if (*actualMoisture < *desiredMoisture) return Signal::INCREASE;
            if (*actualMoisture > *desiredMoisture) return Signal::DECREASE;
        }
        return Signal::MAINTAIN;
    }

    Signal IncrementMoisture() {
        if (actualMoisture && desiredMoisture && *actualMoisture < *desiredMoisture) {
            checkPrecondition(*actualMoisture + 1, "IncrementMoisture");
            *actualMoisture += 1;
            return Signal::INCREASE;
        }
        return Signal::MAINTAIN;
    }

    Signal DecrementMoisture() {
        if (actualMoisture && desiredMoisture && *actualMoisture > *desiredMoisture) {
            checkPrecondition(*actualMoisture - 1, "DecrementMoisture");
            *actualMoisture -= 1;
            return Signal::DECREASE;
        }
        return Signal::MAINTAIN;}
    optional<double> GetActualMoisture() {
        return actualMoisture;
    }

    optional<double> GetDesiredMoisture() {
        return desiredMoisture; }
};



class SoilMoistureControllerTester {
public:
    static void DisplayMoistureLevels(SoilMoistureController& smc) {
        auto actualMoisture = smc.GetActualMoisture();
        auto desiredMoisture = smc.GetDesiredMoisture();
        cout << "Current Actual Moisture: " << (actualMoisture ? to_string(*actualMoisture) :
"Undefined") << endl;
```

```
        cout << "Current Desired Moisture: " << (desiredMoisture ? to_string(*desiredMoisture)
: "Undefined") << endl;}
    static void IncrementMoisture(SoilMoistureController& smc) {
        auto result = smc.IncrementMoisture();
        cout << "Increment Result: " << signalToString(result) << endl; }
        static void SetDesiredMoisture(SoilMoistureController& smc,double ml) {
        smc.SetDesiredMoisture(ml);
        cout << "Moisture set to: " << ml << endl;}

     static void DecrementMoisture(SoilMoistureController& smc) {
        auto result = smc.DecrementMoisture();
        cout << "Decrement Result: " << signalToString(result) << endl;}
     static void MainTesterLoop() {
        double initMoisture;
        SoilMoistureController smc;
            cout << "Set Initial Moisutre: \n";
            cin>>initMoisture;
            smc.SetInitialMoisture(initMoisture);
        char choice;
        do {
            cout << "\n\t\tSoilMoistureController Tester\n";
            cout << "1. Display moisture levels\n";
            cout << "2. Set desired moisture\n";
            cout << "3. Increment moisture\n";
            cout << "4. Decrement moisture\n";
            cout << "5. Quit\n";
            cout << "Enter choice (1-5): ";
            cin >> choice;
            cout << endl;
            switch (choice) {
                case '1': DisplayMoistureLevels(smc); break;
                case '2':
                double ml;
                cout<<"Set Desired Moisture : "<<endl;
                cin>>ml;
                SetDesiredMoisture(smc,ml);
                break;
                case '3': IncrementMoisture(smc); break;
                case '4': DecrementMoisture(smc); break;
                case '5': cout << "Exiting tester." << endl; break;
                default: cout << "Invalid choice, try again." << endl; break;
            }} while (choice != '5');}};
```

```
int main() {
   SoilMoistureControllerTester::MainTesterLoop();
   return 0; }
```

TEST RESULTS:

- DesM = Desired Moisture
- ActM = Actual Moisture

Test CaseId	Actual Moisture	Desired Moisture	Action	Desired Output	Actual Output	Status
01	34	Null	Increment	MAINTAIN	MAINTAIN	PASS
02	34	Null	Decrement	MAINTAIN	MAINTAIN	PASS
03	34	44	Increment	INCREASE	INCREASE	PASS
04	35	44	Decrement	MAINTAIN	MAINTAIN	PASS
05	35	44	Set desired moisture = 30	DesM : 30	DesM : 30	PASS
06	35	30	Increment	MAINTAIN	MAINTAIN	PASS
07	35	30	Decrement	DECREASE	DECREASE	PASS
08	34	30	Display moisture levels	ActM: 34 DesM: 30	ActM: 34 DesM: 30	PASS
09	34	30	Set desired moisture = 19	ERROR	ERROR	PASS
10	34	30	Set desired moisture = 61	ERROR	ERROR	PASS