There is no reasonable alternative except to have a Device Interface layer. It uses the Simulation layer to provide the entire external environment during a simulation and hardware devices, a file system, and the system clock in a fielded program. In either case, the Device Interface layer shields the rest of the program from having to deal with the differences occasioned by these radically different environments.

User interfaces are usually coded using the user interface toolkit provided by a programming environment. However, since the AquaLush simulation uses a GUI and the fielded program uses ATM-like hardware, the user interface must either be coded in several versions or coded once in a device-independent component that uses virtual devices. The latter alternative is clearly preferable. Furthermore, it makes sense to separate the User Interface layer from the Irrigation layer so that the latter can be unchanged even if the user interface is altered.

Finally, the complexity of the configuration problem suggests the need for a layer responsible for configuring the program at startup. Isolating this task in one place makes it easier to code, test, and modify. The Startup layer is responsible for this task.

B.11 AquaLush Detailed Design Document

1. Mid-Level Design Models

The mid-level design models elaborate each architectural layer and are organized in this section by layer.

1.1 Startup Layer Static Structure

The Startup layer is responsible for initializing the program as part of a fielded product or as a Web simulation, configuring the program according to the hardware available to it, and restoring the previous program state. The Startup layer has the static structure shown in Figure B-11-1.

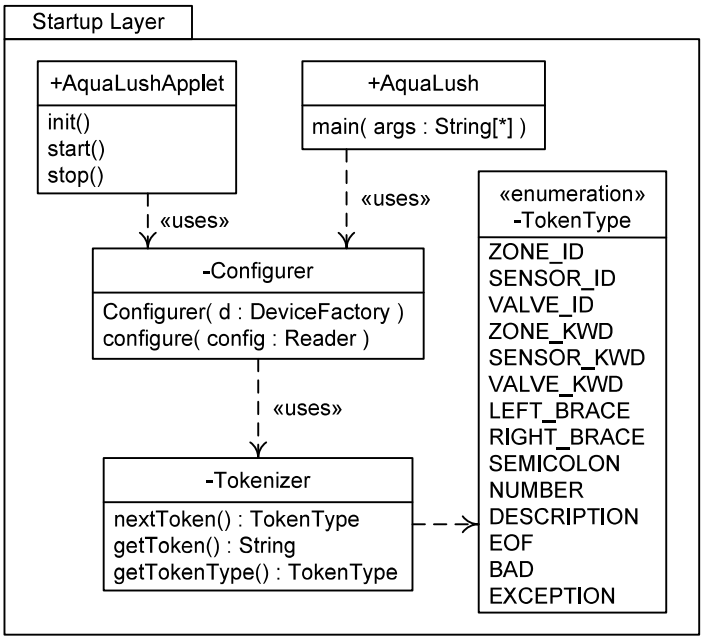

Figure B-11-1 Startup Layer Mid-Level Static Structure

This diagram adds only a few details to the architectural design structure—principally, the Tokenizer and TokenType classes and a few operations in the Configurer and AquaLushApplet classes. These additions are documented in Table B-11-2.

1.1.1 Startup Layer Local Module Responsibilities

Module	Responsibilities
Tokenizer	Process configuration reader input character by character to produce a stream of tokens that the Configurer can parse to interpret the configuration specification.
TokenType	Provide a type-safe token type enumeration for communication between the Tokenizer and the Configurer.

Table B-11-2 Startup Layer Local Module Responsibilities

1.1.2 Startup Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw IllegalArgumentExceptions if the preconditions are violated.

Applet start	*Syntax:*	start()
	Pre:	None.
	Post:	The AquaLush simulated clock is started.
Applet stop	*Syntax:*	stop()
	Pre:	None.
	Post:	The AquaLush simulated clock is stopped.
Configurer construction	*Syntax:*	Configure(d : DeviceFactory)
	Pre:	d is not null.
	Post:	The Configurer is ready to configure the program.
Get next token	*Syntax:*	nextToken() : TokenType
	Pre:	None.
	Post:	Gets the next token from the configuration input stream and returns its type.
Get the text of the current token	*Syntax:*	getToken() : String
	Pre:	nextToken() has been called at least once.
	Post:	The text of the current token is returned. Returns null if the precondition is violated.
Get the type of the current token	*Syntax:*	getTokenType() : TokenType
	Pre:	nextToken() has been called at least once.
	Post:	The type of the current token is returned. Returns TokenType.UNKNOWN if the precondition is violated.

Table B-11-3 Startup Layer Local Module Interface Specifications

1.2 Startup Layer Behavior

The Tokenizer can execute a state machine to recognize configuration file tokens. The diagram in Figure B-11-4 models the tokenizing state machine.

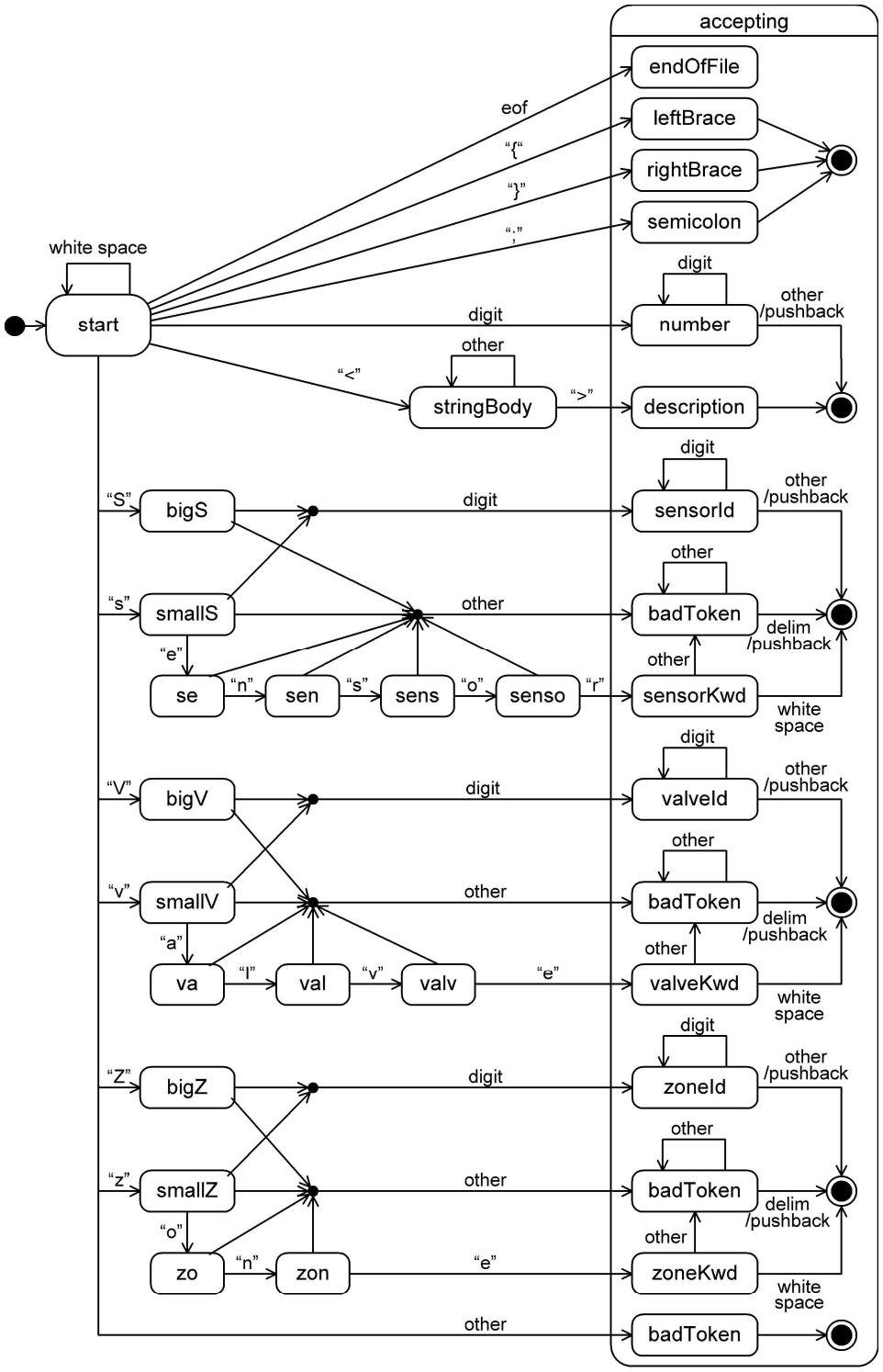

Figure B-11-4 Tokenizer State Machine

In this diagram all events are either the end of the input file, and therefore treated as a character, or single-character inputs. Labels with a single character designate that character; other labels mean the following:

eof—The end of the input file.

white space—A blank, tab, newline, or carriage return character.

digit—The characters "0" through "9."

other—Any character not labeling another arrow emanating from the same state.

The pushback action means that the character last consumed is placed back on the input stream.

1.3 Simulation Layer Static Structure

The Simulation layer includes components that simulate the environment of the program, including system hardware. The Simulation layer contains almost all the user interface code implementing the GUI for the simulation, written using Java Swing components. Most of these entities are not visible at the architectural level. The mid-level design also adds devices that are not explicitly present in the architectural view. Only the details added to the architectural specification are documented in Figure B-11-5, which depicts the detailed structure of the Simulation layer.

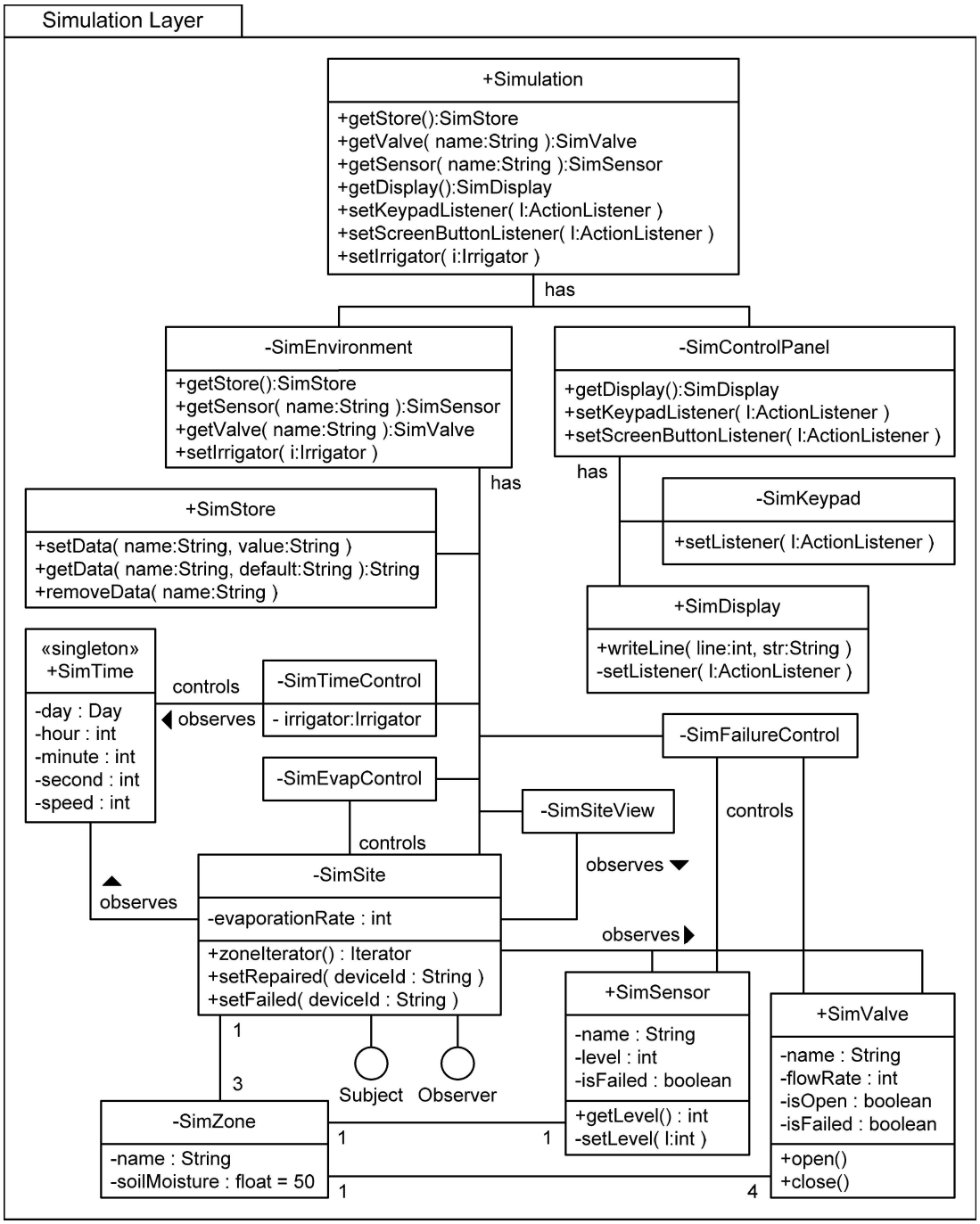

Figure B-11-5 Simulation Layer Mid-Level Static Structure

1.3.1 Simulation Layer Local Module Responsibilities

Module	Responsibilities
SimEnvironment	A Swing panel containing all the displays and controls for the simulated environment, including display of the irrigation site and controls for the simulated time, evaporation rate, and hardware failures and repairs.
SimControlPanel	A Swing panel containing all the widgets for the AquaLush control panel, including the simulated display, screen buttons, and keypad.
SimKeypad	A Swing panel simulating a keypad with 12 buttons.
SimTimeControl	A Swing panel displaying and controlling the simulated time.
SimEvapControl	A Swing panel displaying and controlling the simulated water evaporation rate.
SimSiteView	A Swing panel displaying the irrigation site.
SimFailureControl	A Swing panel displaying and controlling the failure status (failed or running) of the simulated valves and sensors.
SimSite	A collection keeping track of the simulated site evaporation rate and irrigation zones. It is responsible for adjusting the moisture levels in each SimZone.
SimZone	A collection holding a SimSensor and the four SimValves in a portion of the simulated irrigation site.

Table B-11-6 Simulation Layer Local Module Responsibilities

1.3.2 Simulation Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw IllegalArgumentExceptions if the preconditions are violated.

Provide simulated persistent store (SimEnvironment)	*Syntax:*	getStore() : SimStore
	Pre:	None.
	Post:	The simulated storage object is returned.
Provide a simulated valve (SimEnvironment, SimSite, SimZone)	*Syntax:*	SimEnvironment.getValve(name : String) : SimValve
	Pre:	name is not null.
	Post:	The simulated valve whose identifier is named is returned, or null is returned if no such valve exists.
Provide a simulated sensor (SimEnvironment, SimSite, SimZone)	*Syntax:*	getSensor(name : String) : SimSensor
	Pre:	name is not null.
	Post:	The simulated sensor whose identifier is named is returned, or null is returned if no such sensor exists.
Provide the simulated display (SimControlPanel)	*Syntax:*	getDisplay() : SimDisplay
	Pre:	None.
	Post:	The simulated display object is returned.
Register a Keypad listener (SimControlPanel)	*Syntax:*	setKeypadListener(l : KeypadListener)
	Pre:	None..
	Post:	KeypadListener l will start to receive notifications of simulated keypad key presses.

Register a Keypad listener (SimKeypad)	*Syntax:*	setListener(l : KeypadListener)
	Pre:	None.
	Post:	KeypadListener l will start to receive notifications of simulated keypad key presses.
Register a ScreenButton listener (SimControlPanel)	*Syntax:*	setScreenButtonListener(l : ScreenButtonListener)
	Pre:	None.
	Post:	ScreenButtonListener l will start to receive notifications of simulated screen button presses.
Register a ScreenButton listener (SimDisplay)	*Syntax:*	setListener(l : ScreenButtonListener)
	Pre:	None.
	Post:	ScreenButtonListener l will start to receive notifications of simulated screen button presses.
Set a sensor's moisture level (SimSensor)	*Syntax:*	setLevel(l : int)
	Pre:	0 <= l <= 100.
	Post:	The SimSensor's moisture level is changed.
Set the failure status of a sensor (SimSensor)	*Syntax:*	setIsFailed(value : Boolean)
	Pre:	None.
	Post:	The SimSensor's failure status is set as indicated by value.
Set the failure status of a sensor (SimValve)	*Syntax:*	setIsFailed(value : Boolean)
	Pre:	None.
	Post:	The SimValve's failure status is set as indicated by value.

Table B-11-7 Simulation Layer Local Module Interface Specifications

Services Required

The SimStore class will use Java services for persistent storage provided by the `java.util.Properties` class. Properties will be stored in a file called "AquaLushState.xml."

1.3.4 Implementation Notes

The SimSite is supposed to represent reality, so the SimZones and their SimSensors and SimValves are set up explicitly with code in the various class constructors.

1.4 Simulation Layer Behavior

The SimSite observes SimTime using the Observer pattern. When a minute passes, the SimSite passes its evaporationRate attribute (which is in percent per hour) to each SimZone. The SimZones determine how to change the moisture level of each SimSensor. The SimSite notifies its observer (the SimSiteView) when the sensors or valves change. As an optimization, the SimSiteView is not updated on the minute because the SimZones may be in the process of changing. The sequence diagram in Figure B-11-8 models this interaction.

One additional feature must be noted about the Simulation layer's structure and behavior. The SimTimeControl has a button that jumps the simulated time to one hour before the next scheduled irrigation time. This feature does not fit into the architecture because it requires the Simulation layer to use the Irrigation layer, in violation of the layering constraint. This single violation of the Layered style cannot be avoided. However, its effects can be minimized as follows:

- The SimTimeControl is passed a reference to the Irrigator object. It can interrogate the Irrigator when its jump button is pressed to obtain the next irrigation time and day. Thus, the Irrigator does not use the Simulation layer as required in the architecture.

- The AquaLush applet gets a reference to the Irrigator from the Configurer and passes it to the Simulation object, which passes it to SimEnvironment, which passes it to SimTimeControl. Thus, no parts of the program are involved in this violation of architectural constraints except for the applet and the Simulation layer, which causes the problem and is never part of a fielded product.

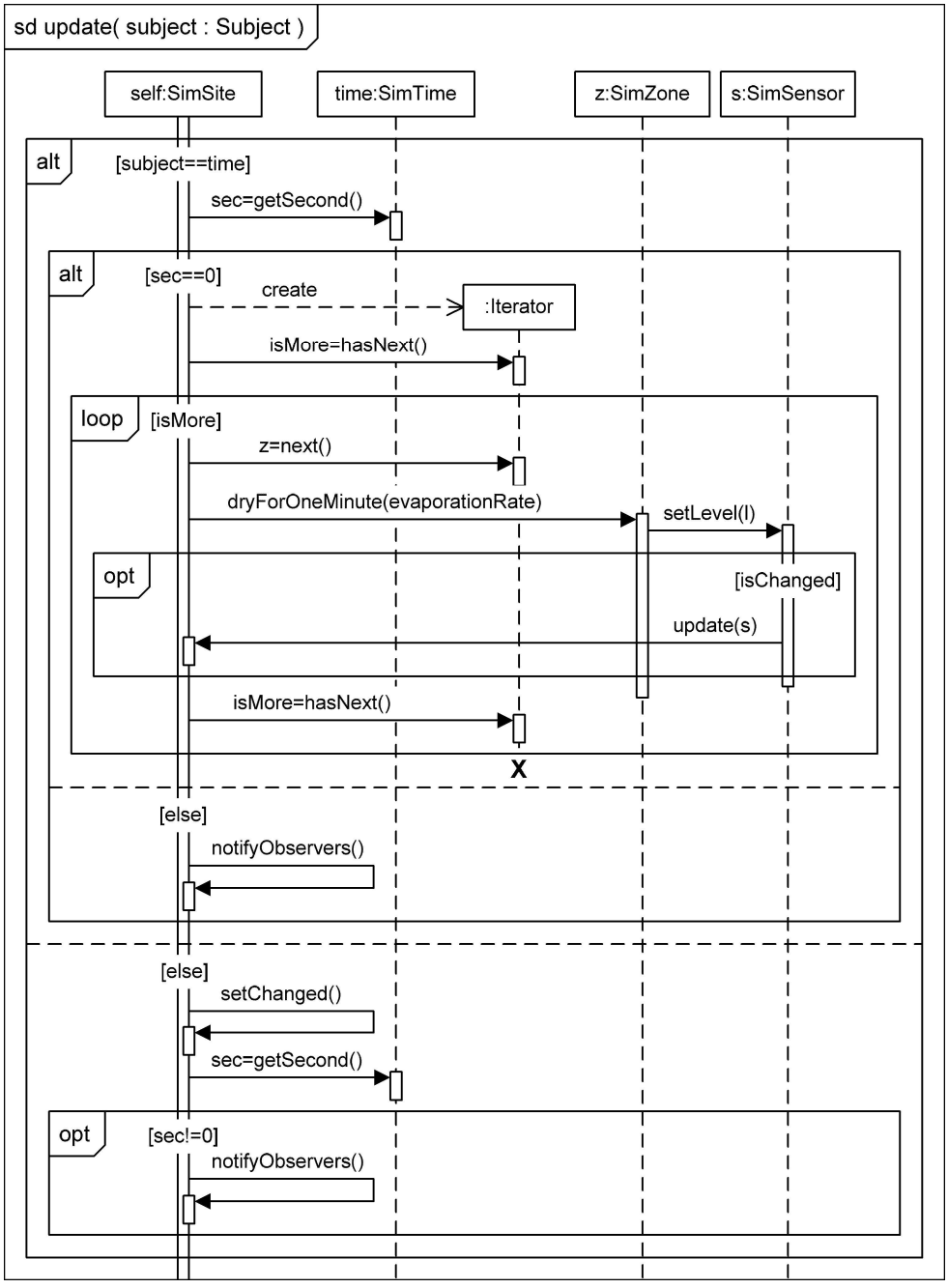

Figure B-11-8 SimSite.update() Behavior

1.5 Device Interface Layer Static Structure

The Device Interface layer provides virtual devices to hide the real or simulated devices used by the program. The mid-level design of this layer merely adds the devices conforming to the interfaces specified at the architectural level of detail. The diagram in Figure B-11-9 includes various "real" device classes. These are placeholders for one or more device drivers for actual hardware devices.

The mid-level static structure of the Device Interface layer is shown in Figure B-11-9.

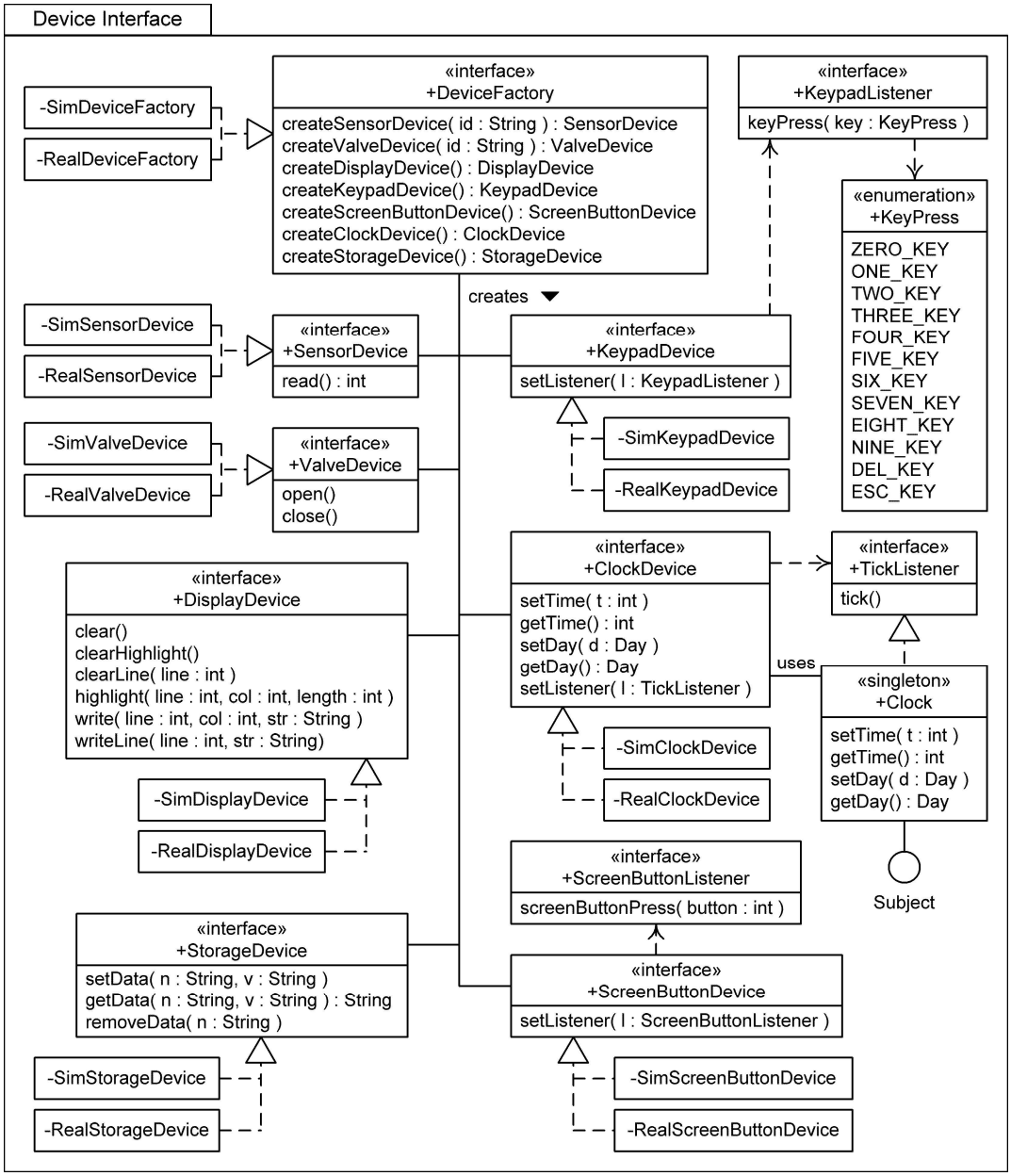

Figure B-11-9 Device Interface Layer Mid-Level Static Structure

1.6 Irrigation Layer Static Structure

The Irrigation layer is the central module of the application. It controls automatic irrigation and acts as the user's agent during manual irrigation. The architectural view of this module presents a façade for controlling manual and automatic irrigation and for retrieving reports about the state of irrigation for display to the user. The mid-level design view adds the classes and operations necessary to realize the system configuration and control irrigation. The mid-level design structure is illustrated in Figure B-11-10.

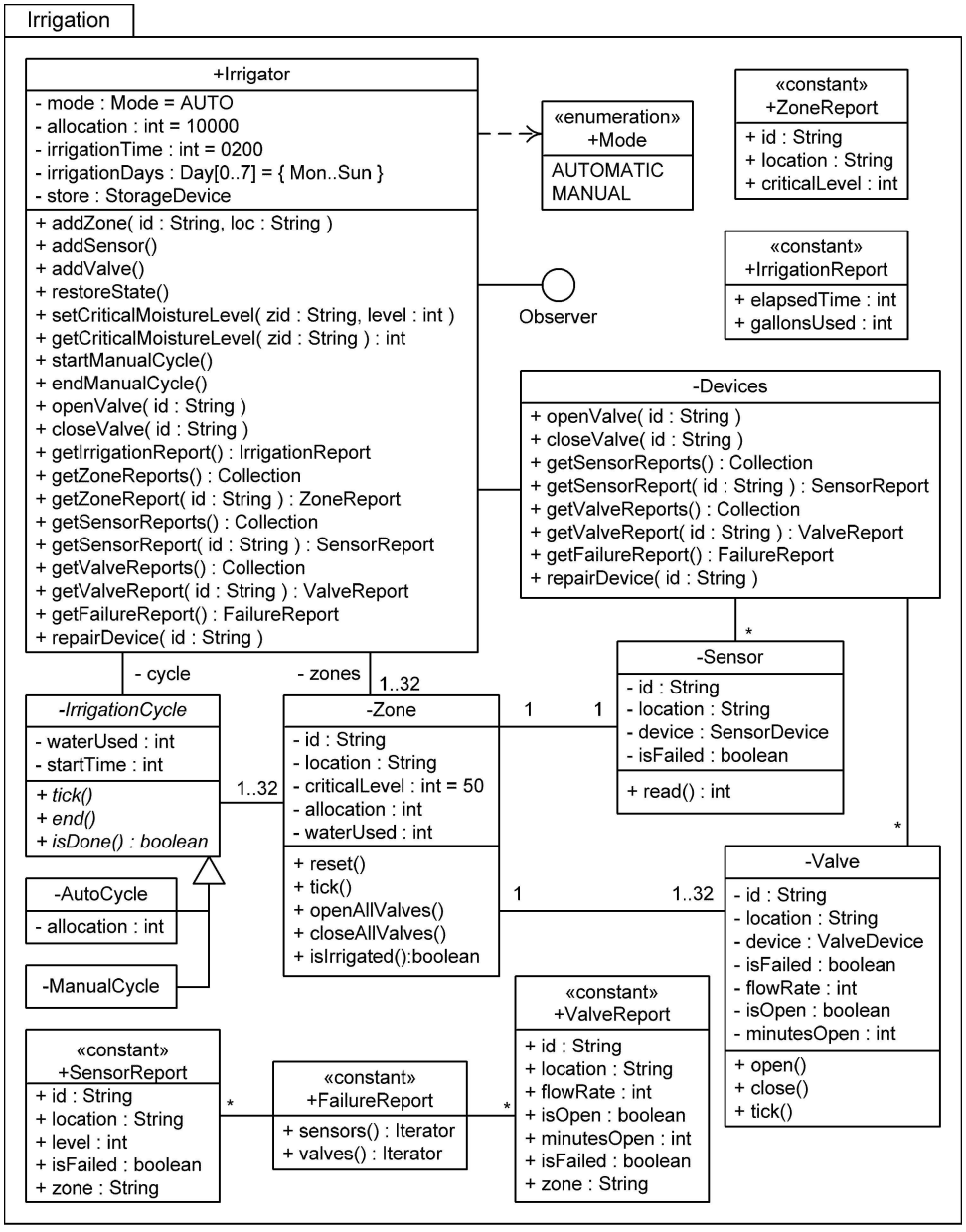

Figure B-11-10 Irrigation Layer Mid-Level Static Structure

1.6.1 Irrigation Layer Local Module Responsibilities

Module	Responsibilities
Devices	Keep track of all devices to make it easier to obtain reports about them and change their failure statuses.
IrrigationCycle	Abstract super-class for all irrigation cycles.
AutoCycle	Control an automatic irrigation cycle.
ManualCycle	Control a manual irrigation cycle.
Zone	Hold zone data and manage automatic irrigation of the zone.
Sensor	Hold sensor data and read a SensorDevice.
Valve	Hold valve data, keep track of how much water a valve uses during irrigation, and control a ValveDevice.

Table B-11-11 Irrigation Layer Local Module Responsibilities

1.6.2 Irrigation Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw IllegalArgumentExceptions if the preconditions are violated.

Create an automatic irrigation cycle (AutoCycle)	*Syntax:*	create(allocation:int, zones:Collection)
	Pre:	0 < allocation and no irrigation cycle is in progress.
	Post:	A new automatic irrigation cycle is created and immediately starts.
Create a manual irrigation cycle (ManualCycle)	*Syntax:*	create(zones : Collection)
	Pre:	No irrigation cycle is in progress.
	Post:	A new manual irrigation cycle is created.
One minute passes (IrrigationCycle, Zone, Valve)	*Syntax:*	tick()
	Pre:	None.
	Post:	Time dependent actions are completed: AutoCycle ticks the current zone and checks if zone irrigation is complete. ManualCycle ticks all zones. Zone ticks its valves and then updates water used. Valve (if open) updates minutes open.
See if a cycle is ended	*Syntax:*	isDone() : Boolean
	Pre:	None.
	Post:	Returns true if automatic irrigation is complete, false otherwise.
A cycle is ended (IrrigationCycle)	*Syntax:*	end()
	Pre:	None.
	Post:	All valves are closed.
Check for auto irrigation completion (Zone)	*Syntax:*	isIrrigated() : boolean
	Pre:	None.
	Post:	Zone returns true if auto irrigation is completed.

Open or close all valves in a zone	*Syntax:*	closeAllValves() openAllValves()
	Pre:	None.
	Post:	All the valves in the zone are opened or closed.
Manually open or close a valve (Irrigator)	*Syntax:*	openValve(id : String) closeValve(id : String)
	Pre:	id is not null and names a registered valve.
	Post:	A valve is opened or closed. Attempts to open or close failed valves do nothing.
Get sensor data (Irrigator)	*Syntax:*	getSensorReports() : Collection getSensorReport(id : String) : SensorReport
	Pre:	id is not null and names a registered sensor.
	Post:	If no sensor is identified, a collection of SensorReports, one for each registered sensor, is populated and returned. If a sensor is identified, a report for that sensor is populated and returned.
Get valve data (Irrigator)	*Syntax:*	getValveReports() : Collection getValveReport(id : String) : ValveReport
	Pre:	id is not null and names a registered valve.
	Post:	If no valve is identified, a collection of ValveReports, one for each registered valve, is populated and returned. If a valve is identified, a report for that valve is populated and returned.
Get a failed hardware report (Irrigator)	*Syntax:*	getFailureReport() : FailureReport
	Pre:	None.
	Post:	A new FailureReport is created and populated. It will contain ValveReports and SensorReports only for failed devices.
Mark a device as repaired (Irrigator)	*Syntax:*	repairDevice(id : String)
	Pre:	id is not null and names a registered valve or sensor.
	Post:	The indicated valve or sensor is marked as repaired. The change is recorded in persistent store.

Table B-11-12 Irrigation Layer Local Module Interface Specifications

1.6.3 Irrigation Layer Design Rationale

The Devices collection is not strictly necessary because each Zone can handle the Devices module responsibilities. However, the Zone class already has several responsibilities, so the reporting and repair status modification responsibilities are passed off to the Devices class to simplify the Zone class.

The AutoCycle class originally had full control over automatic irrigation. For example, the AutoCycle class would query each Zone to obtain its Sensor and critical moisture level, and then read the Sensor to compare it with the critical moisture level. The AutoCycle class would query the Zone to obtain its valve set; go through the set and query each Valve about its flow rate, its allocation, and how long the valve had been open to compute its water usage; and then compare this with its allocation to decide whether the allocation was exhausted. This design made for a very bloated and complex AutoCycle class. An alternative with distributed control lets each Zone decide whether irrigation is complete and lets each Valve decide whether it has used up its allocation. This simplifies the design and was chosen as the better design alternative.

1.7 Irrigation Layer Behavior

The central behavior of the Irrigation layer is its response time. The Irrigator object is an observer of the Clock. The behavior of the Irrigator update() operation is pictured in Figure B-11-13.

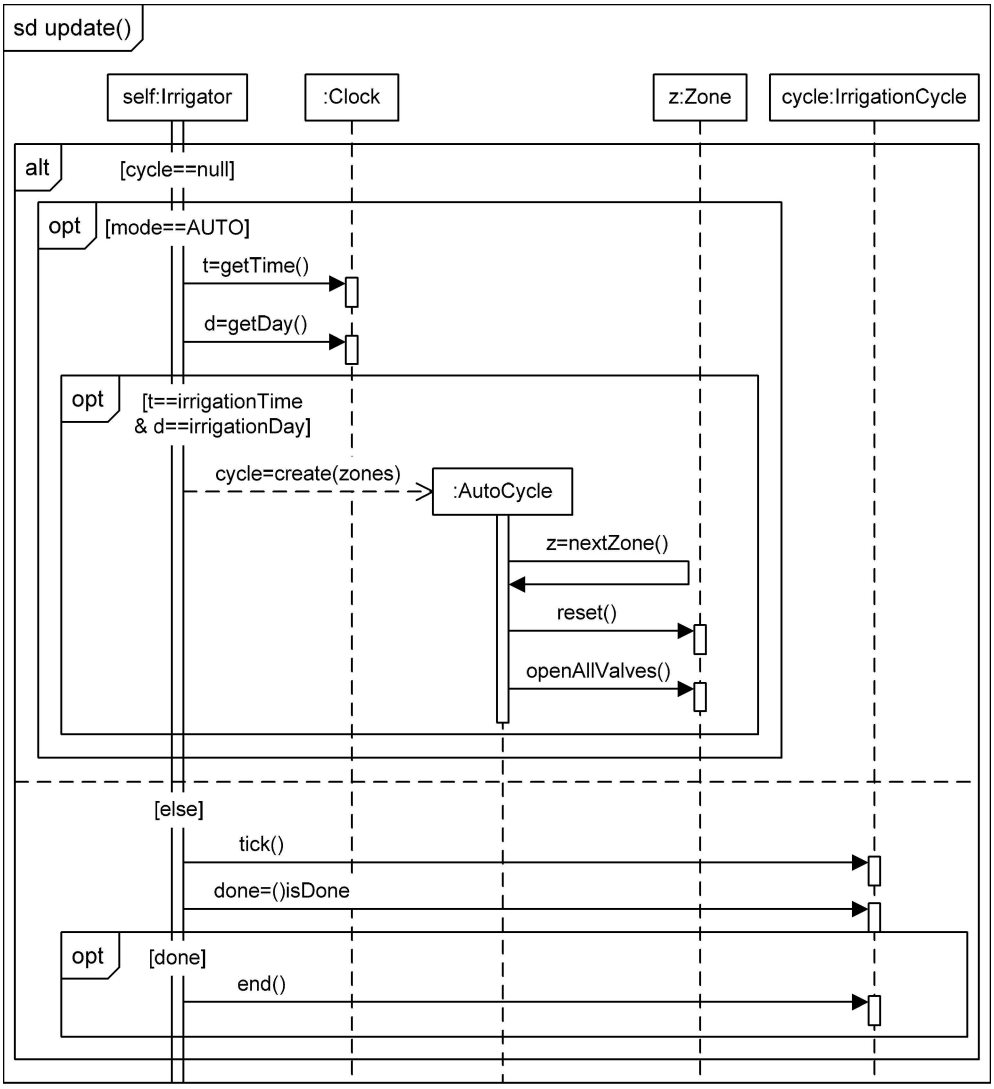

Figure B-11-13 Irrigator.update() Behavior

The Irrigator calls IrrigationCycle.tick() whenever a cycle is in process. The AutoCyle.tick() operation is modeled in Figure B-11-14.

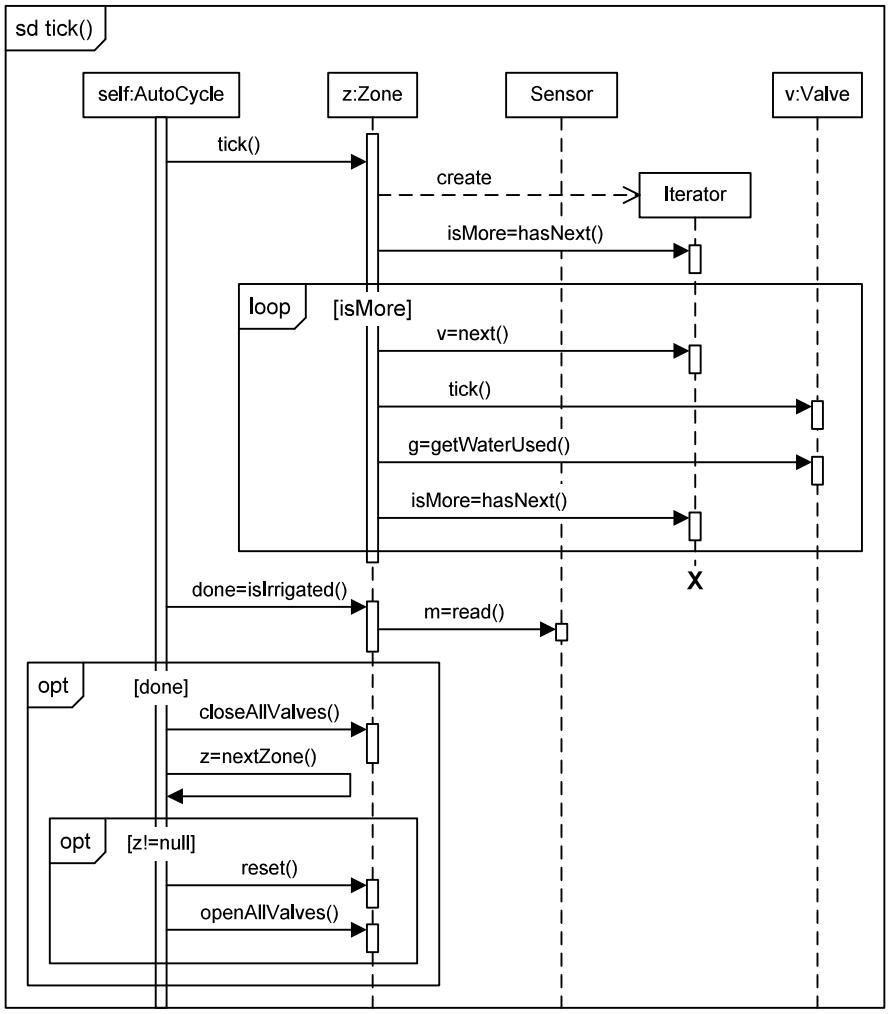

Figure B-11-14 AutoCyle.tick() Behavior

The ManualCycle.tick() operation is pictured in Figure B-11-15.

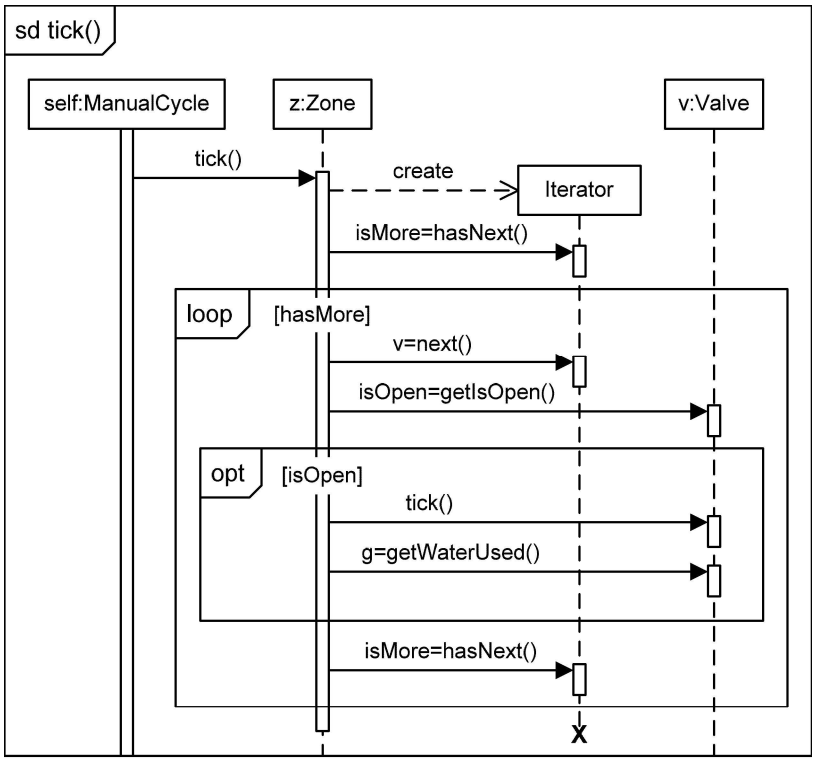

Figure B-11-15 ManualCycle.tick() Behavior

1.8 User Interface Layer Static Structure

The User Interface layer coordinates interaction with a user by consuming input provided via a 12-key keypad and 8 screen buttons and providing output on a monochrome screen of 16 lines with 40 characters per line. This layer has a main UIController class that executes a state machine. Each state corresponds to a screen. Transitions occur when users press screen buttons. Internal actions are prompted by screen button and keypad key presses.

The details added in mid-level design are the classes representing the screen states and auxiliary classes. The static structure of the User Interface layer is shown in Figure B-11-16.

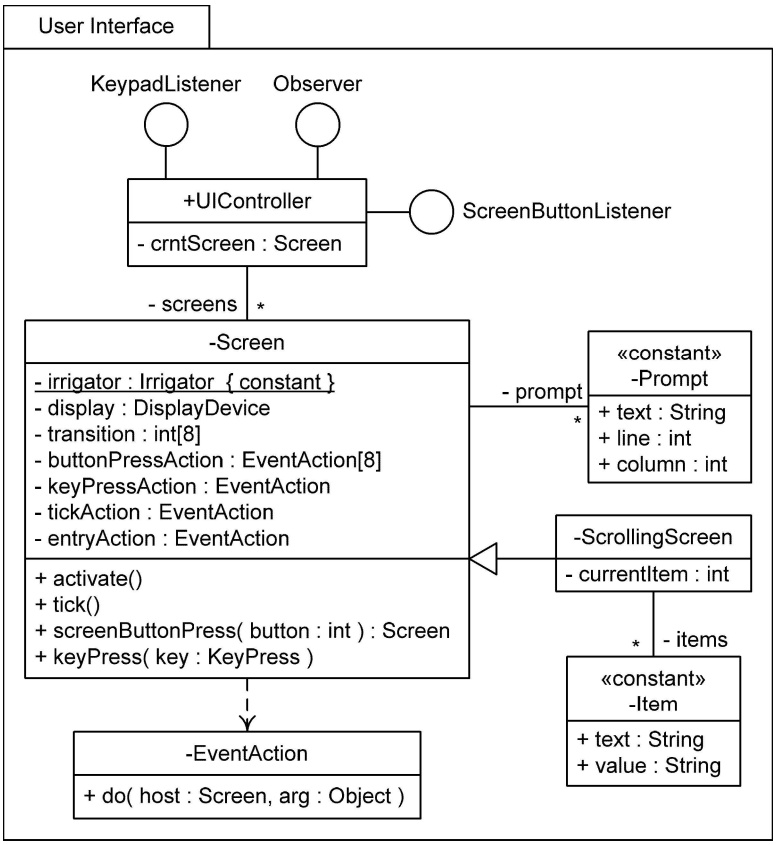

Figure B-11-16 User Interface Layer Mid-Level Static Structure

The Screen super-class is for "plain" screens without scrollable regions. It uses the Command pattern to assign actions for screen button presses and keypad key presses.

1.8.1 User Interface Layer Local Module Responsibilities

Module	Responsibilities
Screen	Control the display and process user input and clock ticks (routed to it from UIController). Each Screen object is a state in a state machine.
ScrollingScreen	Display a scrollable list in the center of the display and handle scrolling commands.
Item	An immutable class holding text and values for the scrollable list.
Prompt	An immutable class holding text written to screens.
EventAction	Command pattern command class whose operation is called when events occur.

Table B-11-17 User Interface Layer Local Module Responsibilities

1.8.2 User Interface Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw IllegalArgumentExceptions if the preconditions are violated.

Activate a screen when it becomes current	*Syntax:*	activate()
	Pre:	None.
	Post:	The display screen is cleared and all prompts are written to it. If activationAction is not null, its do() operation is called with arg set to null.
Notify a screen that time has passed	*Syntax:*	tick()
	Pre:	None.
	Post:	If tickAction is not null, its do() operation is called with arg set to null.
Notify a screen that a screen button has been pressed	*Syntax:*	screenButtonPress(button : int) : Screen
	Pre:	None.
	Post:	If buttonPressAction[button] is not null, its do() operation is called with arg set to the button value. If transition[button] is –1, the result is the current screen; otherwise, the result is screen with number transition[button].
Notify a screen that a keypad key has been pressed	*Syntax:*	keyPress(key : KeyPress)
	Pre:	None.
	Post:	If keyPressAction is not null, its do() operation is called with arg set to the key value.

Table B-11-18 User Interface Layer Local Module Interface Specifications

Usage Guide

The UIController must create and initialize all the Screen and EventAction objects. It must then execute the state machine by keeping track of the current screen. The UIController calls a screen's activate() operation when it becomes current. The UIController must also direct one-minute clock ticks, screen button presses, and keypad key presses to the current screen whenever they occur. Note that the UIController is an observer of the Clock, but the screens are not. This way only the current screen gets time notifications.

1.8.3 User Interface Layer Design Rationale

There are many alternative ways to implement the state machine that the UIController must realize. Among the alternatives considered were those with many specialized Screen sub-classes. For example, there might be a TimeSettingScreen class and an AllocationSettingScreen class. This alternative is more complicated but less flexible, so it was decided to give the Screen super-class a very general mechanism for handling events.

1.9 User Interface Layer Behavior

The user interface changes state based on user input. If each screen represents a state, then user interface states are captured by the dialog map in the SRS. The state diagram in Figure B-11-19 brings the dialog maps in the SRS together and adds internal actions to them.

Events B0 through B7 are presses of screen buttons 0 through 7. States in this diagram are represented by instances of the Screen class. The internal actions are handled either by the screen object or by command objects registered with the screen object.

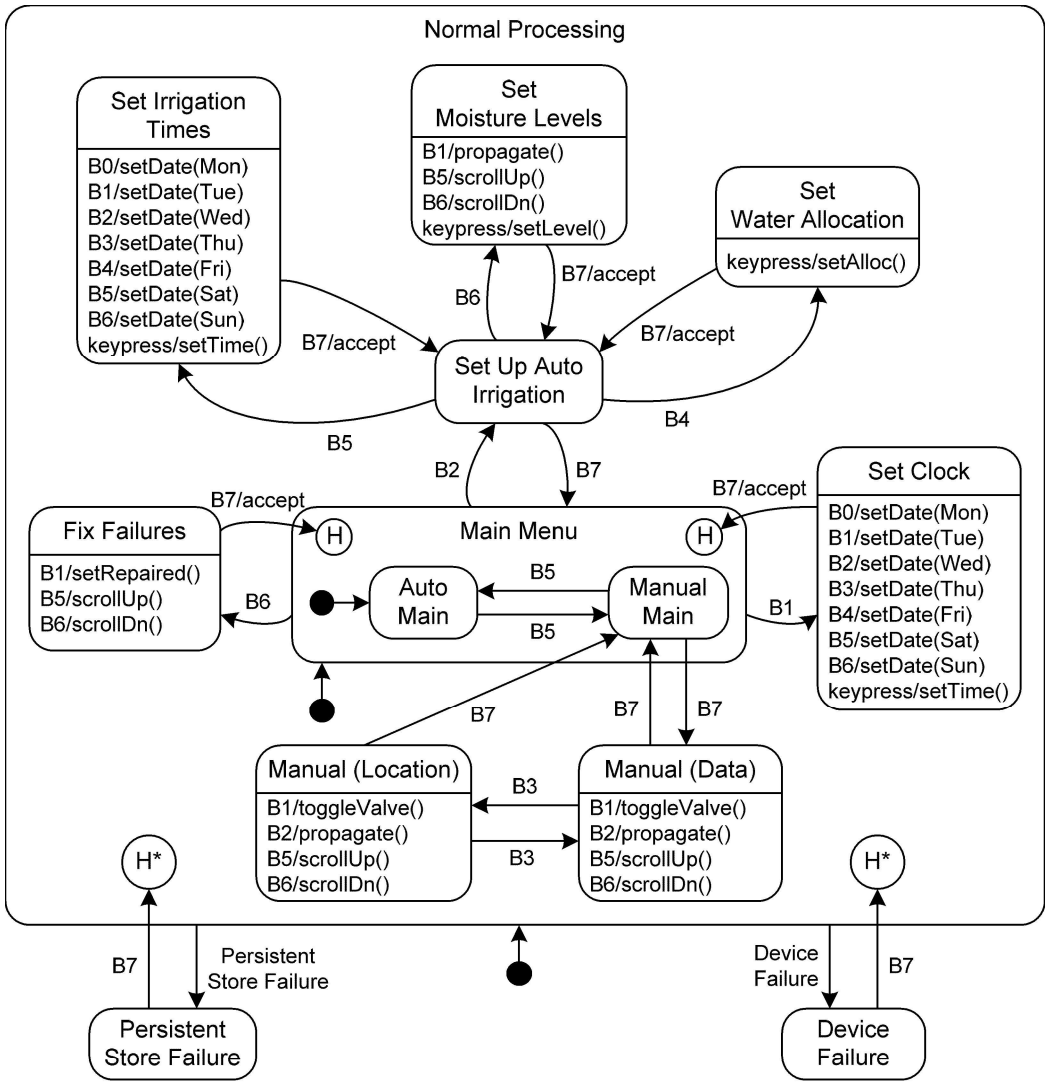

Figure B-11-19 User Interface States

The UIController simply passes keypad presses and time notifications on to the current screen. However, a screen button press may cause a state change, meaning the interaction is more complex. Figure B-11-20 models the interaction that occurs when a screen button is pressed.

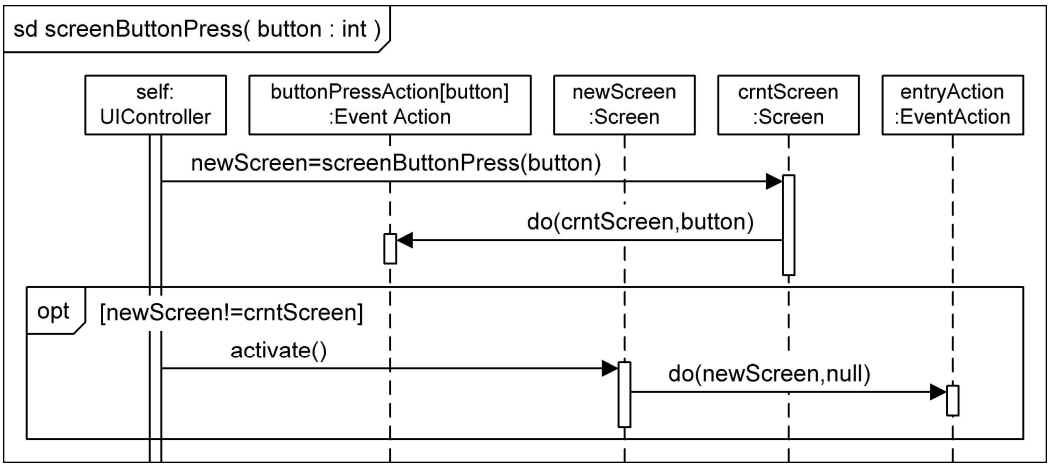

Figure B-11-20 UIController Screen Button Press Handling

2. Low-Level Design Models

There are few tricky low-level details about this program: The data structures and algorithms are all quite straightforward, and no advanced language features are required, though some can be used to make the code smaller. Consequently, only the module packaging and a few operation specifications are supplied in this section.

2.1 Packaging

Java packages are used to implement the design packages for each layer. This keeps the code structure very similar to the static design structure, making it easy to find the code that corresponds to each part of the design. Specifically, the following Java packages are used:

`startup`—Startup-layer code modules.

`ui`—User Interface-layer code modules.

`irrigation`—Irrigation-layer code modules.

`device`—Device Interface-layer code modules.

`device.sim`—Code for virtual devices for simulated hardware.

`device.real`—Code for virtual devices for real hardware.

`simulation`—Simulation-layer code modules.

`util`—Utility code modules (such as widely used exceptions and enumerations).

3. Mapping Between Design Models

The virtual devices in the Device Interface layer use simulated hardware in the Simulation layer. Table B-11-21 indicates the mapping between virtual devices and simulated devices.

Device Interface Layer (device)	Virtual Device (device.sim)	Simulated Device (simulation)
DisplayDevice	SimDisplayDevice	SimDisplay
ScreenButtonDevice	SimScreenButtonDevice	SimDisplay
KeypadDevice	SimKeypadDevice	SimKeypad
StorageDevice	SimStorageDevice	SimStore
ClockDevice	SimClockDevice	SimTime
SensorDevice	SimSensorDevice	SimSensor
ValveDevice	SimValveDevice	SimValve

Table B-11-21 Virtual Device to Simulated Device Mapping

There are two items to be noted about this mapping:

1. Both the SimDisplayDevice and the SimScreenButtonDevice virtual devices use the SimDisplay simulated device. It is easier to implement the GUI realizing the display and the screen buttons together as a single class, so this single simulated device provides two logically distinct services.
2. The SimClockDevice relies on SimTime, which is not a simulated piece of hardware. SimTime keeps track of time in the simulated world. It would have been possible to create a SimClock entity, but it would only have mediated the communication between SimTime and SimClockDevice, so it was not used.

4. Detailed Design Rationale

The prevailing principles driving this design are the following:

- Adhere to the access constraints imposed by the layered architecture to maintain its integrity.
- Hide information, make coherent modules, and decouple modules as much as possible to make the program as changeable as possible.
- Take advantage of the conceptual model to make design models reflect the problem domain as closely as possible.
- Design to interfaces as much as possible.
- Take advantage of standard mid-level design patterns to solve design problems and make the design easy to understand.

These principles are used to help generate, evaluate, and select design alternatives. For example, the Irrigation layer's design is based largely on the AquaLush conceptual model. The notification-driven behavior of the entire program is based on the Observer pattern, though it is implicit in the architecture as well.

A different set of guiding principles might have produced a different detailed design. For example, reuse is not emphasized in this design. Had it been, the Tokenizer class in the Startup layer would probably be based on the `java.util.StreamTokenizer` class rather than being written from scratch.

The only violation of architectural constraints is that the SimTimeControl uses the Irrigator to obtain the irrigation time and day. Much thought was given to ways to implement this feature without violating architectural principles, but it is impossible to do so: The SimTimeControl cannot work properly without data generated in higher layers of the program. Recognizing this, the next question was how to minimize the impact of the violation. One alternative considered was to make the SimTimeControl an observer of a higher-layer object such as a UI Screen or the Irrigator object. However, this would have required making these objects into subjects for no other reason.

It was realized that the SimTimeContol only needs a reference to the Irrigator, which already provides query functions for the data it needs, and that the AquaLush applet can query the Configurer to get an Irrigator object reference to pass to the Simulation object. This requires only a few lines of code in the applet, an extra query function in the Configurer, and extra functions in Simulation and SimEnvironment. This appears to be the simplest and least intrusive way to provide the data that SimTimeControl needs to realize its requirements.