Project Plan: Simulator for the X2 Insulin Pump

Video Walkthrough

https://www.youtube.com/watch?v=aARcW3aMKM0

Team Members

Daniel B. Solomon
Bradley Nguyen
Shane Sibley-Felstead
Sabateesh Sivakumar

Software Development Methodologies

We're doing the incremental development method rather than the scrum approach. It is characterized by being more structured, ensuring each step is completed comprehensively before moving on to the next step. We can afford this because the specs are clearly laid out ahead of time. However, some parts must be left in a rough draft state on the first go-around, such as the UML diagrams.

Team Responsibilities

Responsibility	Description	Complete by	Status
Create project plan	This table	March 19th	Done.
Create use cases, including use case diagram	Makes all use case with the team for approval (done by all)	March 25th	Done.
Create traceability matrix	Iterate on this over time.	March 25th	Done.
Create UML Class diagram rough draft	Include textual explanation of design decisions. It's understood that design will likely change during implementation, hence why these are considered rough drafts.	March 25th	Done.
Create UML Sequence diagrams rough draft	^ Ditto.	March 25th	Done.

Create State machine diagrams rough draft	^ Ditto.	March 25th	Done.
Sketch GUI	Use Figma to collaborate regarding the design of UI.	March 25th	Done.
Establish coding standards	To ensure our code is homogenous. For example: using camelCase.	April 5th	Done.
* Backend Implementation	Implement designs. Deviate from design when necessary.	April 5th	Done.
* Frontend Implementation	^ Ditto.	April 5th	Done.
Merge modular pieces of the program	The program will be coded with modularity in mind. We work on our sections separately in our own branches, then merge them at the end.	April 5th	Done.
Review coding styling	Ensure comments are informative and up-to-date. Ensure variables are self-documenting where applicable.	April 6th	Done.
Extensively test code	Ensure it's working and all requirements are met.	April 6th	Done.
Update UML diagrams	For any updates to design made during the design phase, update the UML diagrams accordingly.	April 7th	Done.
Record Video	A video recording, running through the simulation with the UML sequence diagrams	April 8th	Done.
Submit project	Each member must submit the project.	April 8th	Done.
Do design-code review (interview)	Each member is interviewed individually.	After April 8th (scheduled individually)	WIP.

* = Done simultaneously by separate members.

Use Case docs

USE CASE 2: Getting Started

Primary Actor:
Patient/User

Stakeholders:
Tandem

Pre-conditions:
Device is currently powered off and properly set up.
A USB cable is available for charging if needed.

Success guarantee:
Device is turned on and accessible.

Main success scenario:
1. Charge device using provided USB cable. Battery indicator shows progress.
2. Turn the pump on by holding the power button until the startup sequence is complete.
3. Display touchscreen interface that enables users to navigate between various screens easily.

Extensions:
3a.
3a1. If user has PIN-based lock screen set up, do not display the interface. Instead, display the lock screen. If user inputs correct PIN code, open as normal.
3b.
3b1. User can put the device to sleep via the option menu. Sleep mode should turn the display off while preserving the current session.
3c.
3c1. Display warning when battery is critically low. Suggest user connect the device to a charger.

USE CASE 3: Setting Insulin Delivery Settings

Primary Actor:
Patient/User

Stakeholders:
Tandem

Pre-conditions:
Device is properly set up and powered on.

Success guarantee:
Personal profiles are properly managed according to the user's different daily needs.

Main success scenario:
1. User navigates to the personal profiles section.
2. User creates and names new profile, including critical settings such as basal rates, carbohydrate ratios, correction factors, target glucose levels

Extensions:
1a.
    1a1. User selects a profile to create.
1b.
    1b1. User selects a profile to update.
1c.
    1c1. User selects a profile to delete.

USE CASE 4: Setting Manual Bolus

Primary Actor:
Patient/User

Stakeholders:
Tandem

Pre-conditions:
Device is properly set up and powered on.

Success guarantee:
Correct bolus amount is calculated.

Main success scenario:
1. User accesses the bolus calculator from the home screen or bolus button.
2. User inputs their current blood glucose level and carbohydrate intake manually or pull the values from CGM.
3. Device calculates and suggests an appropriate dose based on settings like insulin sensitivity and target glucose levels

Extensions:
3a.

3a1. Users can override the recommendation by manually adjusting the dose suggested.

USE CASE 5: Starting, Stopping, or Resume Bolus

Primary Actor:
Patient/User

Stakeholders:
Tandem

Pre-conditions:
Device is properly set up, powered on, and bolus settings are set.

Success guarantee:
Bolus is delivered.

Main success scenario:
1. User selects a basal rate from an active personal profile or manually configures it.
2. When insulin is low, begin delivering. System continuously delivers insulin at a specified rate.
3. Stop delivery of insulin either manually or automatically if CGM detects glucose levels below 3.9 mmol/L.

Extensions:
2a.
2a1. If interrupted, stop insulin delivery.
2b.
2b1. If occlusion is detected, display an alert, suggesting user to check the infusion site.
2c.
2c1. During critical failures, the system suspends insulin delivery to prevent incorrect dosing.
3a.
3a1. Resume insulin delivery once the CGM data indicates that glucose levels have stabilized.
3b.
3b1. If CGM is disconnected, triggers alert and informs users of corrective actions.
3c.
3c1. For severe errors like pump shutdowns, provide guidance on how to restart or contacting support if necessary

USE CASE 6: Displaying Information and History

Primary Actor:
Patient/User

Stakeholders:
Tandem

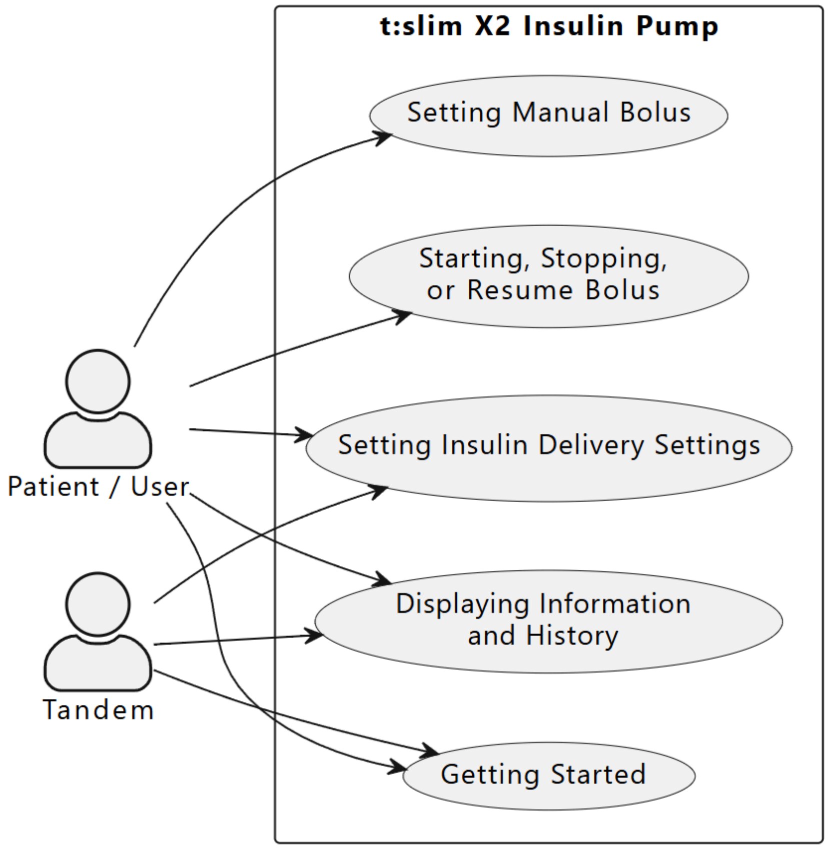
Pre-conditions:
Device is powered on.

Success guarantee:
User access detailed records of insulin delivery and system events.

Main success scenario:
1. User selects History from the home screen.
2. System displays log information such as basal rates, bolus injections, insulin duration, and correction factors

Extensions:
—

Use Case Diagram


t:slim X2 Insulin Pump
Setting Manual Bolus
Starting, Stopping, or Resume Bolus
Setting Insulin Delivery Settings
Displaying Information and History
Getting Started
Patient / User
Tandem

Traceability Matrix

ID	Requirements	Related Use Case	Fulfilled By	Test	Description

1	Home screen is the central hub where users can monitor insulin delivery, battery life, and CGM data.	-	HomeScreen, MainWindow, PumpController	Run the simulator. Observe the initial Home Screen displays battery, insulin, glucose data, IOB, and navigation buttons (Bolus, Options).	MainWindow manages screens and displays HomeScreen. HomeScreen UI (setupUi) shows indicators (batteryLabel, insulinLabel, glucoseLabel, iobValueLabel) and buttons. PumpController supplies data via signals/updates.
2	Home includes a bolus button, that directs to the bolus calculator, and an options button which provides access to insulin delivery settings, alerts and system configurations.	-	HomeScreen (UI), MainWindow (Navigation/Signals)	Run simulator. Click the "BOLUS" button, verify navigation to Bolus Screen. Click the gear ("⚙") Options button, verify navigation to Options Screen.	HomeScreen contains bolusButton and optionsButton. MainWindow::connectSignals connects their clicked signals to MainWindow::showBolusScreen and MainWindow::showOptionsScreen respectively.
3	Return to the home screen by tapping the Tandem logo from any screen within the pump.	-	MainWindow::setupUi (Logo creation/connection), MainWindow::showHomeScreen	Navigate away from Home Screen (e.g., Options). Click the Tandem logo at the top. Verify navigation back to the Home Screen.	MainWindow::setupUi creates the logo QLabel, installs an event filter, and connects its click event (via lambda) to MainWindow::showHomeScreen.
4	Real-time battery indicator at top left corner of display.	-	HomeScreen::setupUi, HomeScreen::updateBatteryLevel	Run simulator. Observe the battery indicator (bars/percentage) top-left. Use Test Panel to change battery level and observe the indicator update visually (color/bars/text).	HomeScreen::setupUi creates batteryWidget (batteryBars, batteryLabel) in the status bar. HomeScreen::updateBatteryLevel updates visuals based on PumpController data.

5	Show remaining insulin in the 300 unit cartridge at the top right corner.	-	HomeScreen::setupUi, HomeScreen::updateInsulinRemaining	Run simulator. Observe the insulin indicator (bars/units) top-right. Use Test Panel to change insulin level and observe the indicator update visually (color/bars/text).	HomeScreen::setupUi creates insulinWidget (insulinBars, insulinLabel) in the status bar. HomeScreen::updateInsulinRemaining updates visuals based on PumpController data.
6	Insulin on Board (IOB) indicator shows how much insulin is still active in the body from previous bolus injections.	-	HomeScreen::setupUi, HomeScreen::updateInsulinOnBoard, InsulinModel::calculateIOB	Run simulator. Observe the "INSULIN ON BOARD" value. Deliver a bolus (Bolus Screen/Test Panel). Observe the IOB value update over subsequent minutes.	HomeScreen::setupUi creates iobValueLabel. HomeScreen::updateInsulinOnBoard updates it. PumpController gets IOB from InsulinModel, which calculates it based on bolus history and decay (calculateIOB, updateIOB).
7	Allow users to quickly return to the home screen by tapping the Tandem logo from any screen within the pump.	-	MainWindow::setupUi (Logo creation/connection), MainWindow::showHomeScreen	Navigate away from Home Screen. Click the Tandem logo. Verify return to Home Screen.	MainWindow::setupUi creates and connects the logo QLabel click event to MainWindow::showHomeScreen.
8	The pump can be charged using a USB cable, with the battery indicator showing its progress.	2	PumpController::startCharging, PumpModel (State), HomeScreen::updateBatteryLevel (UI)	Use Test Panel (or menu if added) to simulate plugging in USB. Observe battery indicator showing charging status (e.g., icon change - not implemented) and level increasing.	PumpController::startCharging simulates charging, updating PumpModel. Internal timer increments PumpModel battery level. HomeScreen::updateBatteryLevel reflects changes. (Actual USB detection/icon not implemented).

9	The pump can be turned on by holding the power button until the startup sequence is complete.	2	MainWindow::powerOn, PumpController::startPump (Note: Button hold not simulated, but logic exists)	Run simulator (auto-starts after 500ms). Or modify code to require manual start. When off, click power ('X') button. Verify UI activates and simulation starts.	MainWindow::handlePowerButtonPressed calls powerOn when off. powerOn calls PumpController::startPump. (Hold action not simulated).
10	The pump can be turned off or put to sleep via the options menu.	2	MainWindow::handlePowerButtonPressed, MainWindow::simulatePowerOff, PumpController::stopPump	Click power ('X') button while on. Select "Power Off", verify shutdown sequence/stop. Select "Sleep", verify screen dims temporarily.	MainWindow::handlePowerButtonPressed shows options. "Power Off" calls simulatePowerOff -> powerOff -> PumpController::stopPump. "Sleep" dims opacity temporarily.
11	The pump features a touch screen interface that enables users to navigate between different screens.	2	Qt Framework (QPushButton, etc.), MainWindow (Screen management), Various Screen classes (UI)	Click various buttons (Bolus, Options, Back, Profile items, etc.). Verify navigation occurs between HomeScreen, BolusScreen, OptionsScreen, etc.	Application uses Qt widgets. MainWindow manages QStackedWidget. Button signals are connected to MainWindow slots which change the displayed screen.
12	A PIN-based lock screen can be set up for security and to prevent accidental inputs.	2	PinLockScreen, PinSettingsScreen, MainWindow::checkPinLock	Navigate Options -> Security Settings. Enable PIN, set PIN. Lock pump. Attempt access. Verify PinLockScreen appears. Enter correct/incorrect PIN, verify results (access/denial/lockout).	PinSettingsScreen manages settings. MainWindow::checkPinLock shows PinLockScreen if enabled/locked. PinLockScreen verifies entered PIN hash against stored hash.
13	The pump allows users to create, name, and manage new profiles that cater to different needs.	3	ProfileScreen, ProfileModel, PumpController (Profile access)	Navigate Options -> Personal Profiles. Click "New". Enter valid settings. Save. Verify profile appears in list.	ProfileScreen UI for creation. PumpController::createProfile -> ProfileModel::createProfile adds to data. ProfileScreen::loadProfiles refreshes view.

14	User can enter critical settings for a profile like basal rates, carbohydrate ratios, correction factors, and target glucose levels.	3	ProfileScreen (UI), ProfileModel (Storage)	Navigate Options -> Personal Profiles -> New/Edit. Verify fields exist for Basal Rate, Carb Ratio, Correction Factor, Target Glucose. Enter valid values.	ProfileScreen::setupUi creates the edit form with input widgets (QDoubleSpinBox, etc.) for profile parameters.
15	The pump provides a functionality to review/update existing profiles.	3	ProfileScreen, ProfileModel::updateProfile	Navigate Options -> Personal Profiles. Select profile. Click "Edit". Modify settings. Save. Re-edit and verify changes were saved.	ProfileScreen displays list, populates form on edit. PumpController::updateProfile -> ProfileModel::updateProfile modifies stored data.
16	Allow deletion of profiles that are no longer needed.	3	ProfileScreen, ProfileModel::deleteProfile	Navigate Options -> Personal Profiles. Create profile. Select it. Delete & Confirm. Verify removal. Try deleting "Default" or active profile, verify disallowed.	ProfileScreen "Delete" button triggers checks (not Default/active) & confirmation. PumpController::deleteProfile -> ProfileModel::deleteProfile removes data.
17	Users can initiate a manual bolus by accessing the bolus calculator from the homescreen or bolus button.	4	HomeScreen, MainWindow::showBolusScreen, BolusScreen	On Home Screen, click "BOLUS" button. Verify Bolus Screen appears.	HomeScreen's bolusButton signal connected via MainWindow to showBolusScreen, displaying BolusScreen.
18	The pump allows users to manually enter their current blood glucose level and carbohydrate intake.	4	BolusScreen (glucoseSpinBox, carbsSpinBox)	Navigate to Bolus Screen. Verify input fields exist for "Current Glucose" and "Carbohydrates". Enter numerical values.	BolusScreen::setupUi creates glucoseSpinBox and carbsSpinBox for manual entry.

19	The bolus calculator suggests an appropriate dose based on the programmed settings like insulin sensitivity and target glucose levels.	4	BolusScreen::calculateSuggestedBolus, BolusController (Calculations), ProfileModel (Settings)	Navigate to Bolus Screen. Enter glucose (e.g., 12.0) and carbs (e.g., 30g). Observe "Suggested Bolus" value update based on active profile settings.	BolusScreen calls calculateSuggestedBolus on input change. Method uses profile settings (Carb Ratio, Correction Factor, Target) from ProfileModel via PumpController to calculate and display suggestion.
25	Determining how much bolus to provide is done through real-time data reading by the Control IQ technology.	-	ControlIQAlgorithm::calculateBasalAdjustment, PumpController::runControlIQ (Note: Primarily adjusts Basal)	Run simulation with Control-IQ on. Monitor glucose. Observe Control-IQ status/history for automatic basal adjustments. (Code doesn't implement auto-bolus).	ControlIQAlgorithm calculates basal adjustments based on glucose/trend. PumpController::runControlIQ executes this. (Current code does not implement automatic boluses from Control-IQ).
20	The system can pull the blood glucose and carbohydrate data from the CGM.	4	PumpController (Uses GlucoseModel), BolusScreen (Uses controller data)	Navigate to Bolus Screen. Observe "Current Glucose" field pre-populated with latest CGM reading. (Carb data is manual entry).	BolusScreen::updateCurrentValues gets current glucose from PumpController (sourced from GlucoseModel) to pre-fill the field. Carb data requires manual input via carbsSpinBox.
21	Users can override the suggested dose by manually adjusting the bolus amount.	4	BolusScreen (bolusSpinBox UI allows editing)	Navigate to Bolus Screen. Let suggestion calculate. Manually change "Bolus Amount" value. Deliver & Confirm. Verify manual amount is delivered.	BolusScreen's bolusSpinBox allows direct editing by user. on_deliverButton_clicked uses the value from this spin box, overriding any suggestion.

22	The pump supports extended boluses where insulin delivery is spread over a longer period of time.	-	BolusScreen (UI), PumpController::deliverBolus, InsulinModel::deliverBolus	Navigate to Bolus Screen. Enter amount. Check "Extended Bolus". Set duration. Deliver & Confirm. Monitor IOB/history over duration.	BolusScreen UI allows setting extended/duration. PumpController::deliverBolus passes parameters to InsulinModel, which manages timed delivery using extendedBolusTimer.
23	The pump supports quick boluses for immediate correction of high glucose.	-	BolusScreen (UI), PumpController::deliverBolus, InsulinModel::deliverBolus (Standard bolus)	Navigate to Bolus Screen. Enter amount. Leave "Extended Bolus" unchecked. Deliver & Confirm. Verify immediate (simulated ~1s) delivery.	Standard bolus function. BolusScreen takes input, PumpController::deliverBolus calls InsulinModel with extended=false. InsulinModel uses QTimer::singleShot for simulated immediate delivery.
24	Users can cancel or stop a bolus mid-delivery if needed, ensuring flexibility during treatment.	-	PumpController::cancelBolus, InsulinModel::cancelBolus, TestPanel (Test button)	Start an extended bolus. Use Test Panel's "Cancel Active Bolus" button. Confirm. Verify bolus stops and history reflects partial delivery.	TestPanel button calls PumpController::cancelBolus -> InsulinModel::cancelBolus. Model stops timer, calculates delivered amount, updates history, emits bolusCancelled.
26	The simulation should allow for generating, reading and graphing bolus data.	-	HistoryScreen, GraphView, InsulinModel (History), DataStorage, main.cpp (Sample data)	Deliver boluses. Navigate to History screen. Select "Insulin" or "Graph" tab. Set date range. Observe bolus events in table/graph .	InsulinModel records boluses. DataStorage saves/loads. main.cpp adds sample data. HistoryScreen displays table. GraphView renders graph.

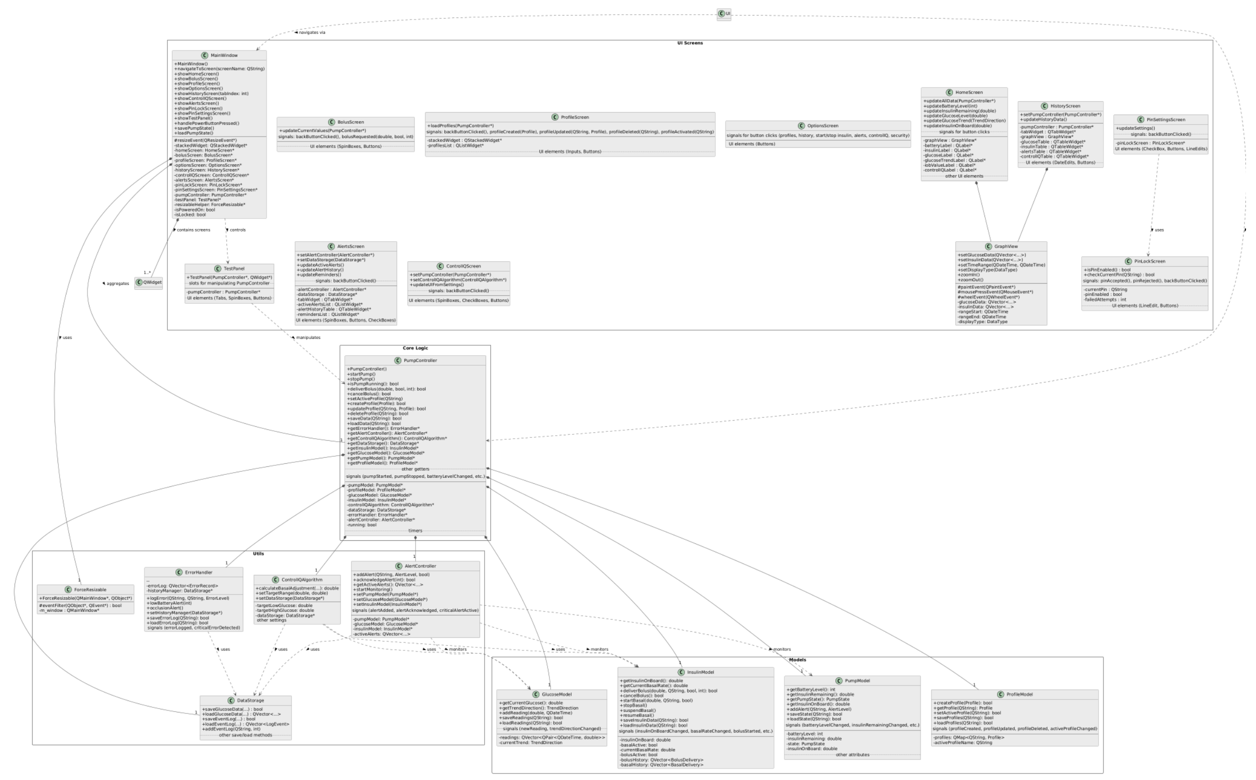
27	Users can start an insulin delivery by selecting a basal rate from their active personal profile or manually configure it through the options menu.	5	OptionsScreen, PumpController::startPump, InsulinModel::startBasal, ProfileModel	Navigate to Options. Click "Start Insulin". Confirm. Verify basal starts at active profile rate. Or, power on pump, verify basal starts automatically.	OptionsScreen button triggers PumpController::startPump (via MainWindow connection). startPump gets active profile from ProfileModel and calls InsulinModel::startBasal.
28	Once confirmed by the user, the pump can continuously deliver insulin at the specified basal rate, unless interrupted.	5	PumpController (Timers), InsulinModel (State management)	Start insulin. Monitor status (IOB, insulin remaining) over time. Verify basal delivery continues without intervention.	InsulinModel basalActive state is true. PumpController::updateBasalConsumption timer deducts insulin based on currentBasalRate until stopped/suspended.
29	Stopping insulin delivery can occur either manually or automatically when Control IQ technology detects low glucose levels (below 3.9 mmol/L).	5	OptionsScreen, PumpController::stopPump, ControlIQAlgorithm, InsulinModel::suspendBasal	Manual: Navigate Options -> Stop Insulin -> Confirm. Verify stop. Auto: Enable Control-IQ. Force glucose < 3.9 mmol/L (Test Panel). Observe basal suspension.	Manual: OptionsScreen -> PumpController::stopPump -> InsulinModel::stopBasal. Auto: ControlIQAlgorithm logic detects low, InsulinModel::suspendBasal is called.
30	Users are able to manually stop insulin delivery.	5	OptionsScreen, PumpController::stopPump, InsulinModel::stopBasal	Navigate to Options. Click "Stop Insulin". Confirm. Verify delivery stops.	OptionsScreen button triggers PumpController::stopPump (via MainWindow connection), which calls InsulinModel::stopBasal.

31	When insulin delivery stops, the event must be logged for future references.	-	InsulinModel::endCurrentBasalSegment, DataStorage, ErrorHandler	Stop insulin delivery. Navigate to History -> Alerts or Control-IQ tab. Verify a stop/suspension event is logged. Check log files.	InsulinModel records end of basal segment in basalHistory. ErrorHandler logs suspension/stop events (triggered by Control-IQ or manual action), stored by DataStorage.
32	When resuming insulin delivery it involves restoring the previous basal rate or switching to an updated profile when glucose levels stabilize.	-	InsulinModel::resumeBasal, PumpController::startPump	Stop insulin. Click "Start Insulin" (Options). Verify basal resumes at previous rate. OR Stop insulin, change active profile, Start insulin. Verify new profile rate is used.	InsulinModel::resumeBasal uses stored pre-suspend rate/profile. PumpController::startPump (after full stop) uses the current active profile.
33	Logs information such as basal rates, bolus injections, insulin duration, and correction factors.	6	InsulinModel (bolusHistory, basalHistory), DataStorage, HistoryScreen	Perform actions (change basal, deliver bolus types). Navigate History -> Insulin tab. Verify events (basal segments, boluses w/ duration/status) are logged. (Correction factor logged via profile).	InsulinModel maintains basalHistory and bolusHistory. DataStorage persists it. HistoryScreen displays it. Profile settings (incl. correction factor) are separate in ProfileModel.
34	When accessing the current status screen, users can view recent events like the time and amount of the last bolus, changes in basal rates, or alerts triggered	-	HomeScreen (IOB, Control-IQ), HistoryScreen, AlertsScreen	Deliver bolus, observe IOB on HomeScreen. Change basal (profile/Control-IQ), observe controlIQLabel. Trigger alert, check AlertsScreen. For full details, check HistoryScreen.	HomeScreen shows IOB & Control-IQ status. Does not show last bolus time/amount or recent basal changes directly. Requires navigation to HistoryScreen / AlertsScreen for detailed recent events. (Partially met).

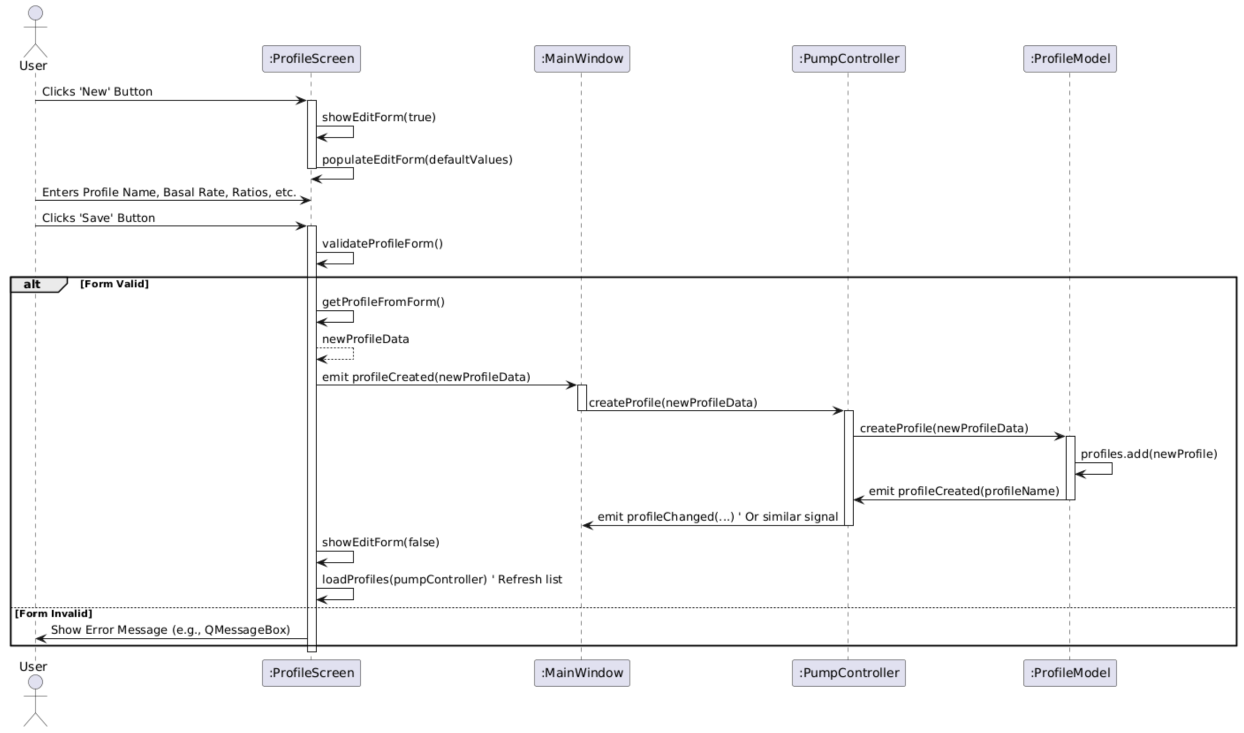
	by CGM readings.				
35	System can detect a low battery and trigger a warning that prompts the user to charge the pump.	2	PumpController::checkLowBattery, PumpModel, ErrorHandler::lowBatteryAlert, AlertController	Use Test Panel to set battery to 15%. Observe "Battery low" alert. Set to 4%. Observe "BATTERY CRITICALLY LOW" alert.	PumpController timer checks PumpModel level via checkLowBattery, calls ErrorHandler::lowBatteryAlert which logs alert, potentially triggering UI via AlertController.
36	System can detect low insulin levels and trigger an alert.	5	PumpController::checkLowInsulin, PumpModel, ErrorHandler::lowInsulinAlert, AlertController	Use Test Panel to set insulin to 40u. Observe "Insulin low" alert. Set to 8u. Observe "INSULIN CRITICALLY LOW" alert.	PumpController checks PumpModel level via checkLowInsulin, calls ErrorHandler::lowInsulinAlert which logs alert, potentially triggering UI via AlertController.
37	System can detect a CGM disconnection and alert the user for a corrective action.	5	AlertController::checkMiscAlerts, ErrorHandler::cgmDisconnectedAlert, GlucoseModel	Stop glucose updates (e.g., pause simulation timer). Wait >10 mins. Observe "CGM data gap" / "CGM SIGNAL LOST" alert.	AlertController timer calls checkMiscAlerts, checks GlucoseModel last reading time. If gap > threshold, logs alert via ErrorHandler.
38	Occlusion alert suggests checking the infusion site for blockages.	5	PumpController::checkForOcclusion, ErrorHandler::occlusionAlert	Run simulation. Wait for random occlusion trigger or force via Test Panel. Observe "OCCLUSION DETECTED" alert with specific message.	PumpController timer simulates check via checkForOcclusion, calls ErrorHandler::occlusionAlert which logs the specific critical message.

39	The pump's error-handling system automatically suspends insulin delivery during critical failures to prevent incorrect dosing.	5	ErrorHandler (Detects critical), PumpController, ControlIQAlgorithm, InsulinModel (Suspension logic)	Trigger critical failure (Occlusion, Critical Low Glucose via Control-IQ). Verify basal delivery stops/suspends (check history/status).	Specific logic handles suspension: Occlusion in PumpController::checkForOcclusion, Low Glucose in ControlIQAlgorithm, potentially others via ErrorHandler::criticalErrorDetected signal connection.
40	Logs errors and malfunctions allowing users and healthcare providers to troubleshoot issues effectively.	-	ErrorHandler::logError, DataStorage, HistoryScreen, AlertsScreen	Trigger various warnings/errors. Navigate History -> Alerts or Alerts -> History tab. Verify logs exist. Check error_log.json file.	Components call ErrorHandler::logError. ErrorHandler adds to errorLog and saves via DataStorage. HistoryScreen/AlertsScreen display the log.
41	In case of severe issues, the system provides guidance on restarting or contacting support if necessary.	5	ErrorHandler::provideTroubleshootingGuidance, ErrorHandler::contactSupportPrompt	Trigger critical error (e.g., Occlusion). Observe alert. Check event/error log for logged guidance/support prompt message. (UI display of guidance not implemented).	ErrorHandler methods log guidance/prompt messages. UI would need to connect to ErrorHandler signals (e.g., criticalErrorDetected) to display this information prominently to the user. (Logging implemented).

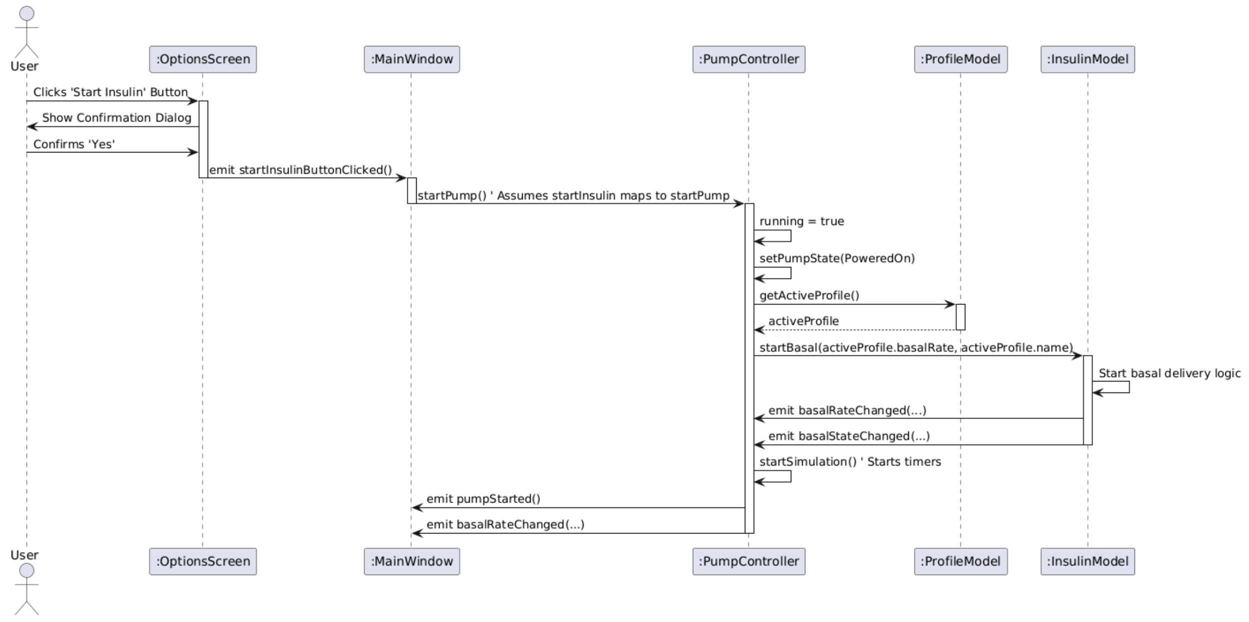
UML Class Diagram


UI
navigates via
UI Screens
MainWindow
+MainWindow()
+navigateToScreen(screenName: QString)
+showHomeScreen()
+showBolusScreen()
+showProfileScreen()
+showOptionsScreen()
+showHistoryScreen(tabIndex: int)
+showControlIQScreen()
+showAlertsScreen()
+showPinLockScreen()
+showPinSettingsScreen()
+showTestPanel()
+handlePowerButtonPressed()
+savePumpState()
+loadPumpState()
#resizeEvent(QResizeEvent*)
-stackedWidget: QStackedWidget*
-homeScreen: HomeScreen*
-bolusScreen: BolusScreen*
-profileScreen: ProfileScreen*
-optionsScreen: OptionsScreen*
-historyScreen: HistoryScreen*
-controlIQScreen: ControlIQScreen*
-alertsScreen: AlertsScreen*
-pinLockScreen: PinLockScreen*
-pinSettingsScreen: PinSettingsScreen*
-pumpController: PumpController*
-testPanel: TestPanel*
-resizableHelper: ForceResizable*
-isPoweredOn: bool
-isLocked: bool
BolusScreen
+updateCurrentValues(PumpController*)
signals: backButtonClicked(), bolusRequested(double, bool, int)
UI elements (SpinBoxes, Buttons)
ProfileScreen
+loadProfiles(PumpController*)
signals: backButtonClicked(), profileCreated(Profile), profileUpdated(QString, Profile), profileDeleted(QString), profileActivated(QString)
-stackedWidget : QStackedWidget*
-profilesList : QListWidget*
UI elements (Inputs, Buttons)
OptionsScreen
signals for button clicks (profiles, history, start/stop insulin, alerts, controlIQ, security)
UI elements (Buttons)
HomeScreen
+updateAllData(PumpController*)
+updateBatteryLevel(int)
+updateInsulinRemaining(double)
+updateGlucoseLevel(double)
+updateGlucoseTrend(TrendDirection)
+updateInsulinOnBoard(double)
signals for button clicks
-graphView : GraphView*
-batteryLabel : QLabel*
-insulinLabel : QLabel*
-glucoseLabel : QLabel*
-glucoseTrendLabel : QLabel*
-iobValueLabel : QLabel*
-controlIQLabel : QLabel*
other UI elements
HistoryScreen
+setPumpController(PumpController*)
+updateHistoryData()
-pumpController : PumpController*
-tabWidget : QTabWidget*
-graphView : GraphView*
-glucoseTable : QTableWidget*
-insulinTable : QTableWidget*
-alertsTable : QTableWidget*
-controlIQTable : QTableWidget*
UI elements (DateEdits, Buttons)
PinSettingsScreen
+updateSettings()
signals: backButtonClicked()
-pinLockScreen : PinLockScreen*
UI elements (CheckBox, Buttons, LineEdits)
uses
contains screens
controls
1..*
QWidget
aggregates
TestPanel
+TestPanel(PumpController*, QWidget*)
slots for manipulating PumpController
-pumpController : PumpController*
UI elements (Tabs, SpinBoxes, Buttons)
AlertsScreen
+setAlertController(AlertController*)
+setDataStorage(DataStorage*)
+updateActiveAlerts()
+updateAlertHistory()
+updateReminders()
signals: backButtonClicked()
-alertController : AlertController*
-dataStorage : DataStorage*
-tabWidget : QTabWidget*
-activeAlertsList : QListWidget*
-alertHistoryTable : QTableWidget*
-remindersList : QListWidget*
UI elements (SpinBoxes, Buttons, CheckBoxes)
ControlIQScreen
+setPumpController(PumpController*)
+setControlIQAlgorithm(ControlIQAlgorithm*)
+updateUIFromSettings()
signals: backButtonClicked()
UI elements (SpinBoxes, CheckBoxes, Buttons)
GraphView
+setGlucoseData(QVector<...>)
+setInsulinData(QVector<...>)
+setTimeRange(QDateTime, QDateTime)
+setDisplayType(DataType)
+zoomIn()
+zoomOut()
#paintEvent(QPaintEvent*)
#mousePressEvent(QMouseEvent*)
#wheelEvent(QWheelEvent*)
-glucoseData: QVector<...>
-insulinData: QVector<...>
-rangeStart: QDateTime
-rangeEnd: QDateTime
-displayType: DataType
PinLockScreen
+isPinEnabled() : bool
+checkCurrentPin(QString) : bool
signals: pinAccepted(), pinRejected(), backButtonClicked()
-currentPin : QString
-pinEnabled : bool
-failedAttempts : int
UI elements (LineEdit, Buttons)
uses
manipulates
Core Logic
PumpController
+PumpController()
+startPump()
+stopPump()
+isPumpRunning(): bool
+deliverBolus(double, bool, int): bool
+cancelBolus(): bool
+setActiveProfile(QString)
+createProfile(Profile): bool
+updateProfile(QString, Profile): bool
+deleteProfile(QString): bool
+saveData(QString): bool
+loadData(QString): bool
+getErrorHandler(): ErrorHandler*
+getAlertController(): AlertController*
+getControlIQAlgorithm(): ControlIQAlgorithm*
+getDataStorage(): DataStorage*
+getInsulinModel(): InsulinModel*
+getGlucoseModel(): GlucoseModel*
+getPumpModel(): PumpModel*
+getProfileModel(): ProfileModel*
other getters
signals (pumpStarted, pumpStopped, batteryLevelChanged, etc.)
-pumpModel: PumpModel*
-profileModel: ProfileModel*
-glucoseModel: GlucoseModel*
-insulinModel: InsulinModel*
-controlIQAlgorithm: ControlIQAlgorithm*
-dataStorage: DataStorage*
-errorHandler: ErrorHandler*
-alertController: AlertController*
-running: bool
timers
1
Utils
ForceResizable
+ForceResizable(QMainWindow*, QObject*)
#eventFilter(QObject*, QEvent*) : bool
-m_window : QMainWindow*
ErrorHandler
--
-errorLog: QVector<ErrorRecord>
-historyManager: DataStorage*
+logError(QString, QString, ErrorLevel)
+lowBatteryAlert(int)
+occlusionAlert()
+setHistoryManager(DataStorage*)
+saveErrorLog(QString): bool
+loadErrorLog(QString): bool
signals (errorLogged, criticalErrorDetected)
ControlIQAlgorithm
+calculateBasalAdjustment(...): double
+setTargetRange(double, double)
+setDataStorage(DataStorage*)
-targetLowGlucose: double
-targetHighGlucose: double
-dataStorage: DataStorage*
other settings
AlertController
+addAlert(QString, AlertLevel, bool)
+acknowledgeAlert(int): bool
+getActiveAlerts(): QVector<...>
+startMonitoring()
+setPumpModel(PumpModel*)
+setGlucoseModel(GlucoseModel*)
+setInsulinModel(InsulinModel*)
signals (alertAdded, alertAcknowledged, criticalAlertActive)
-pumpModel: PumpModel*
-glucoseModel: GlucoseModel*
-insulinModel: InsulinModel*
-activeAlerts: QVector<...>
uses
uses
uses
uses
monitors
uses
monitors
monitors
DataStorage
+saveGlucoseData(...) : bool
+loadGlucoseData(...) : QVector<...>
+saveEventLog(...) : bool
+loadEventLog(...) : QVector<LogEvent>
+addEventLog(QString, int)
other save/load methods
Models
GlucoseModel
+getCurrentGlucose(): double
+getTrendDirection(): TrendDirection
+addReading(double, QDateTime)
+saveReadings(QString): bool
+loadReadings(QString): bool
signals (newReading, trendDirectionChanged)
-readings: QVector<QPair<QDateTime, double>>
-currentTrend: TrendDirection
InsulinModel
+getInsulinOnBoard(): double
+getCurrentBasalRate(): double
+deliverBolus(double, QString, bool, int): bool
+cancelBolus(): bool
+startBasal(double, QString, bool)
+stopBasal()
+suspendBasal()
+resumeBasal()
+saveInsulinData(QString): bool
+loadInsulinData(QString): bool
signals (insulinOnBoardChanged, basalRateChanged, bolusStarted, etc.)
-insulinOnBoard: double
-basalActive: bool
-currentBasalRate: double
-bolusActive: bool
-bolusHistory: QVector<BolusDelivery>
-basalHistory: QVector<BasalDelivery>
PumpModel
+getBatteryLevel(): int
+getInsulinRemaining(): double
+getPumpState(): PumpState
+getInsulinOnBoard(): double
+addAlert(QString, AlertLevel)
+saveState(QString): bool
+loadState(QString): bool
signals (batteryLevelChanged, insulinRemainingChanged, etc.)
-batteryLevel: int
-insulinRemaining: double
-state: PumpState
-insulinOnBoard: double
other attributes
ProfileModel
+createProfile(Profile): bool
+getProfile(QString): Profile
+setActiveProfile(QString): bool
+saveProfiles(QString): bool
+loadProfiles(QString): bool
signals (profileCreated, profileUpdated, profileDeleted, activeProfileChanged)
-profiles: QMap<QString, Profile>
-activeProfileName: QString

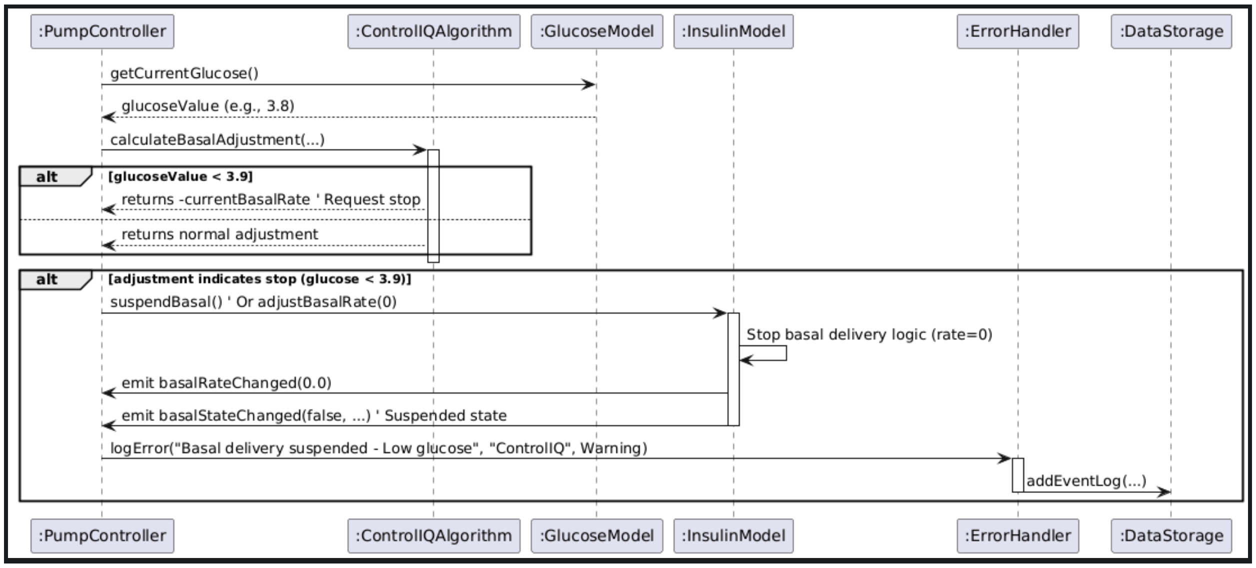
Sequence Diagram 1: User Creates a New Profile


User
:ProfileScreen
:MainWindow
:PumpController
:ProfileModel
Clicks 'New' Button
showEditForm(true)
populateEditForm(defaultValues)
Enters Profile Name, Basal Rate, Ratios, etc.
Clicks 'Save' Button
validateProfileForm()
alt
[Form Valid]
getProfileFromForm()
newProfileData
emit profileCreated(newProfileData)
createProfile(newProfileData)
createProfile(newProfileData)
profiles.add(newProfile)
emit profileCreated(profileName)
emit profileChanged(...) ' Or similar signal
showEditForm(false)
loadProfiles(pumpController) ' Refresh list
[Form Invalid]
Show Error Message (e.g., QMessageBox)

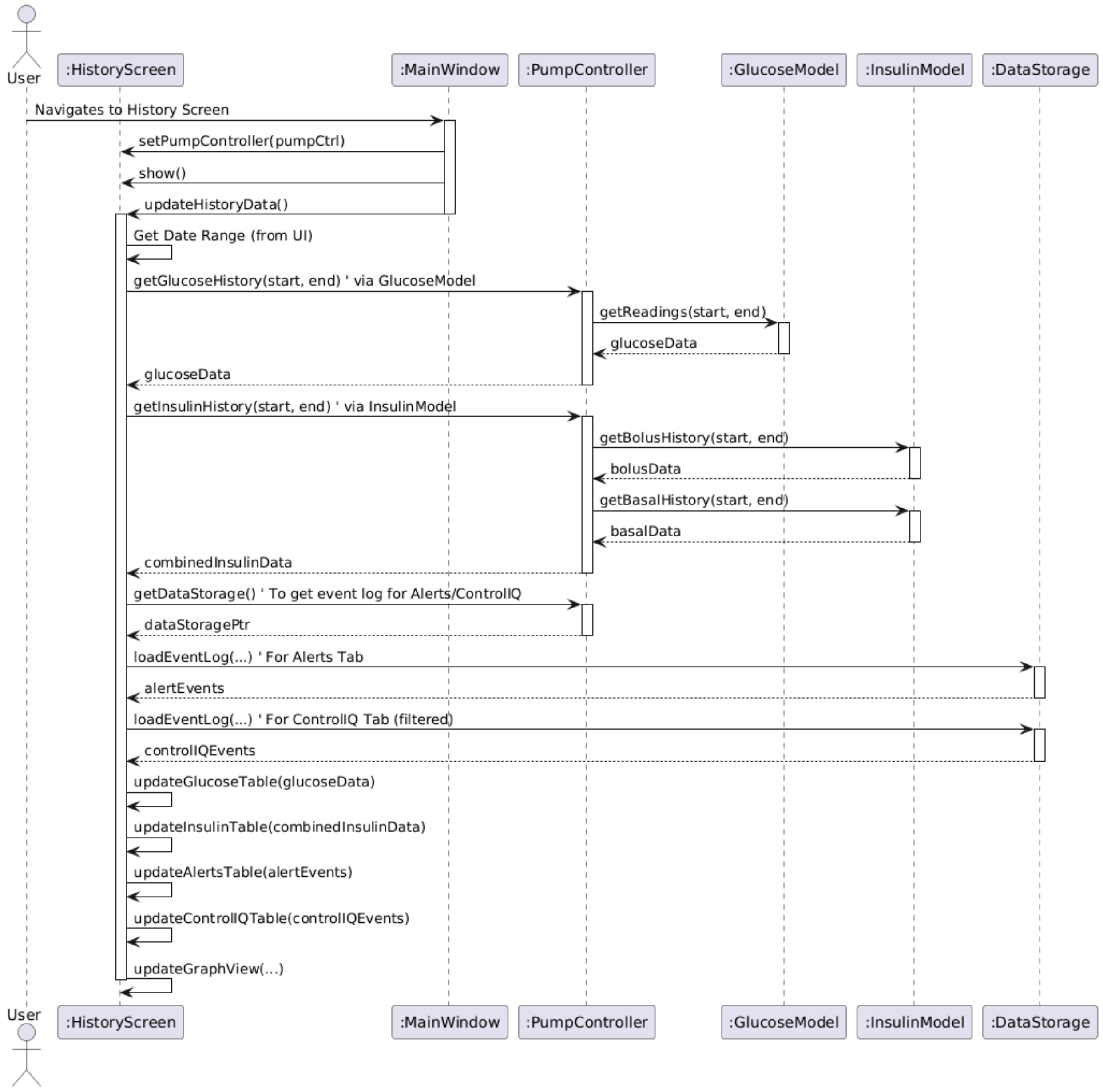
Sequence Diagram 2: User Starts Insulin Delivery


User
:OptionsScreen
:MainWindow
:PumpController
:ProfileModel
:InsulinModel
Clicks 'Start Insulin' Button
Show Confirmation Dialog
Confirms 'Yes'
emit startInsulinButtonClicked()
startPump() ' Assumes startInsulin maps to startPump
running = true
setPumpState(PoweredOn)
getActiveProfile()
activeProfile
startBasal(activeProfile.basalRate, activeProfile.name)
Start basal delivery logic
emit basalRateChanged(...)
emit basalStateChanged(...)
startSimulation() ' Starts timers
emit pumpStarted()
emit basalRateChanged(...)

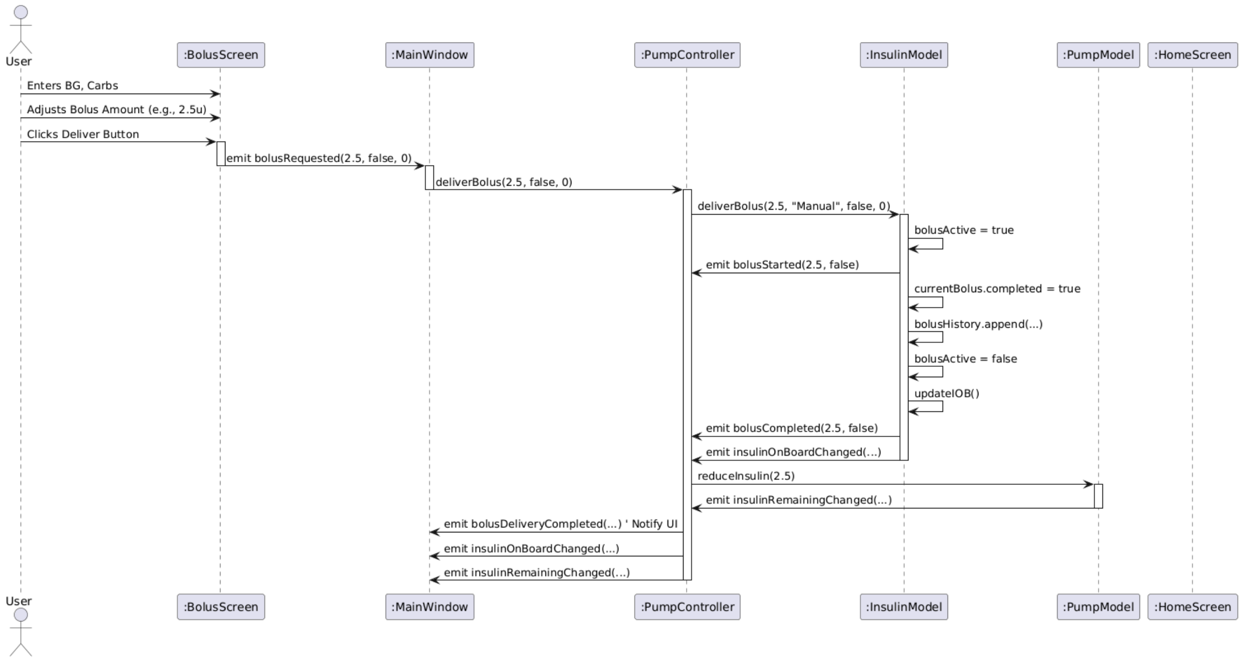
Sequence Diagram 3: Auto-Stop Basal Delivery (Low Glucose)


:PumpController
:ControlIQAlgorithm
:GlucoseModel
:InsulinModel
:ErrorHandler
:DataStorage
getCurrentGlucose()
glucoseValue (e.g., 3.8)
calculateBasalAdjustment(...)
alt
[glucoseValue < 3.9]
returns -currentBasalRate ' Request stop
returns normal adjustment
alt
[adjustment indicates stop (glucose < 3.9)]
suspendBasal() ' Or adjustBasalRate(0)
Stop basal delivery logic (rate=0)
emit basalRateChanged(0.0)
emit basalStateChanged(false, ...) ' Suspended state
logError("Basal delivery suspended - Low glucose", "ControlIQ", Warning)
addEventLog(...)
:PumpController
:ControlIQAlgorithm
:GlucoseModel
:InsulinModel
:ErrorHandler
:DataStorage

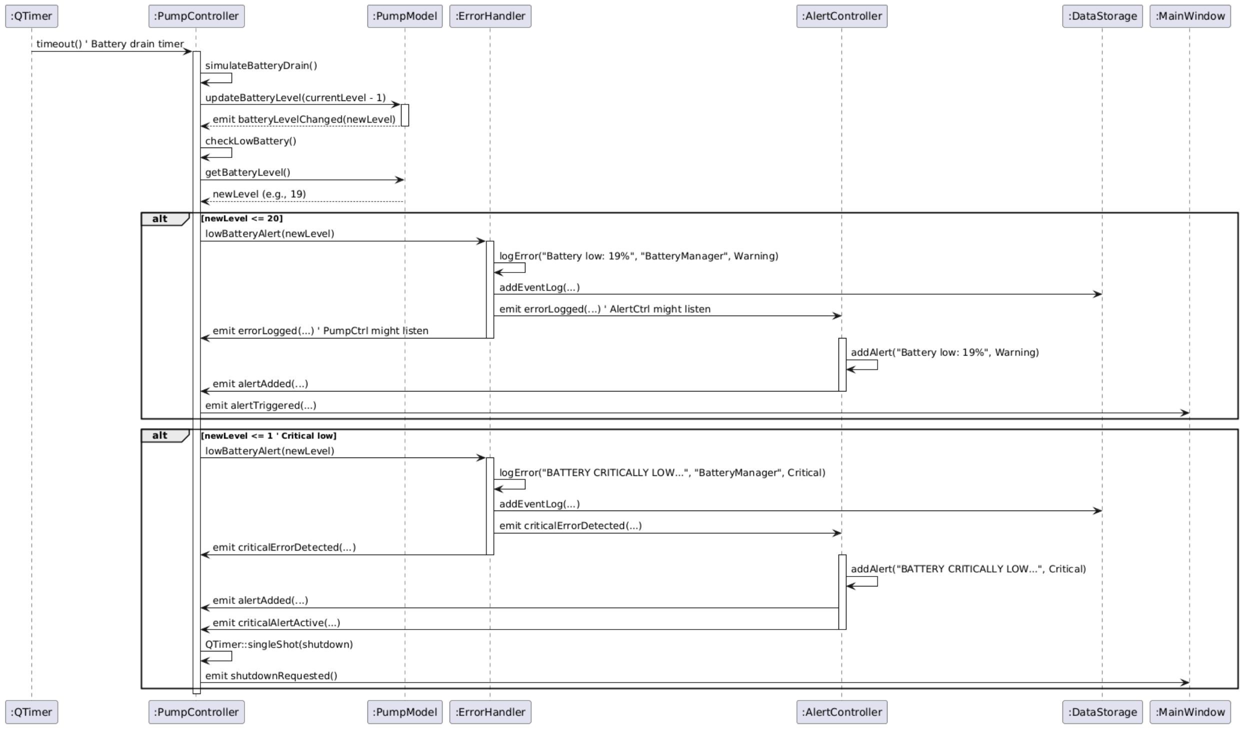
Sequence Diagram 4: User Views History


User
:HistoryScreen
:MainWindow
:PumpController
:GlucoseModel
:InsulinModel
:DataStorage
Navigates to History Screen
setPumpController(pumpCtrl)
show()
updateHistoryData()
Get Date Range (from UI)
getGlucoseHistory(start, end) ' via GlucoseModel
getReadings(start, end)
glucoseData
glucoseData
getInsulinHistory(start, end) ' via InsulinModel
getBolusHistory(start, end)
bolusData
getBasalHistory(start, end)
basalData
combinedInsulinData
getDataStorage() ' To get event log for Alerts/ControlIQ
dataStoragePtr
loadEventLog(...) ' For Alerts Tab
alertEvents
loadEventLog(...) ' For ControlIQ Tab (filtered)
controlIQEvents
updateGlucoseTable(glucoseData)
updateInsulinTable(combinedInsulinData)
updateAlertsTable(alertEvents)
updateControlIQTable(controlIQEvents)
updateGraphView(...)
User
:HistoryScreen
:MainWindow
:PumpController
:GlucoseModel
:InsulinModel
:DataStorage

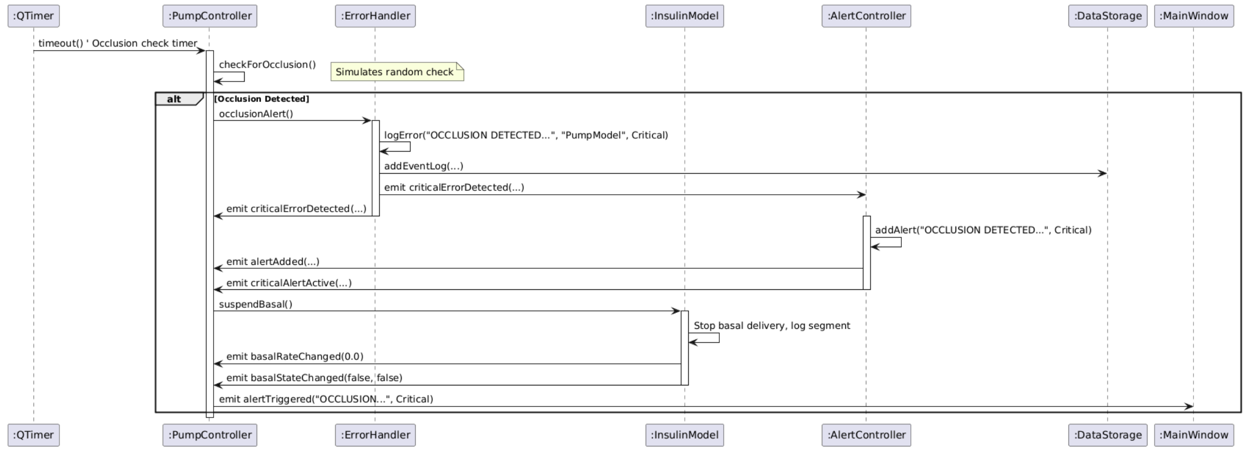
Sequence Diagram 5: Manual Bolus


User
:BolusScreen
:MainWindow
:PumpController
:InsulinModel
:PumpModel
:HomeScreen
Enters BG, Carbs
Adjusts Bolus Amount (e.g., 2.5u)
Clicks Deliver Button
emit bolusRequested(2.5, false, 0)
deliverBolus(2.5, false, 0)
deliverBolus(2.5, "Manual", false, 0)
bolusActive = true
emit bolusStarted(2.5, false)
currentBolus.completed = true
bolusHistory.append(...)
bolusActive = false
updateIOB()
emit bolusCompleted(2.5, false)
emit insulinOnBoardChanged(...)
reduceInsulin(2.5)
emit insulinRemainingChanged(...)
emit bolusDeliveryCompleted(...) ' Notify UI
emit insulinOnBoardChanged(...)
emit insulinRemainingChanged(...)
User
:BolusScreen
:MainWindow
:PumpController
:InsulinModel
:PumpModel
:HomeScreen

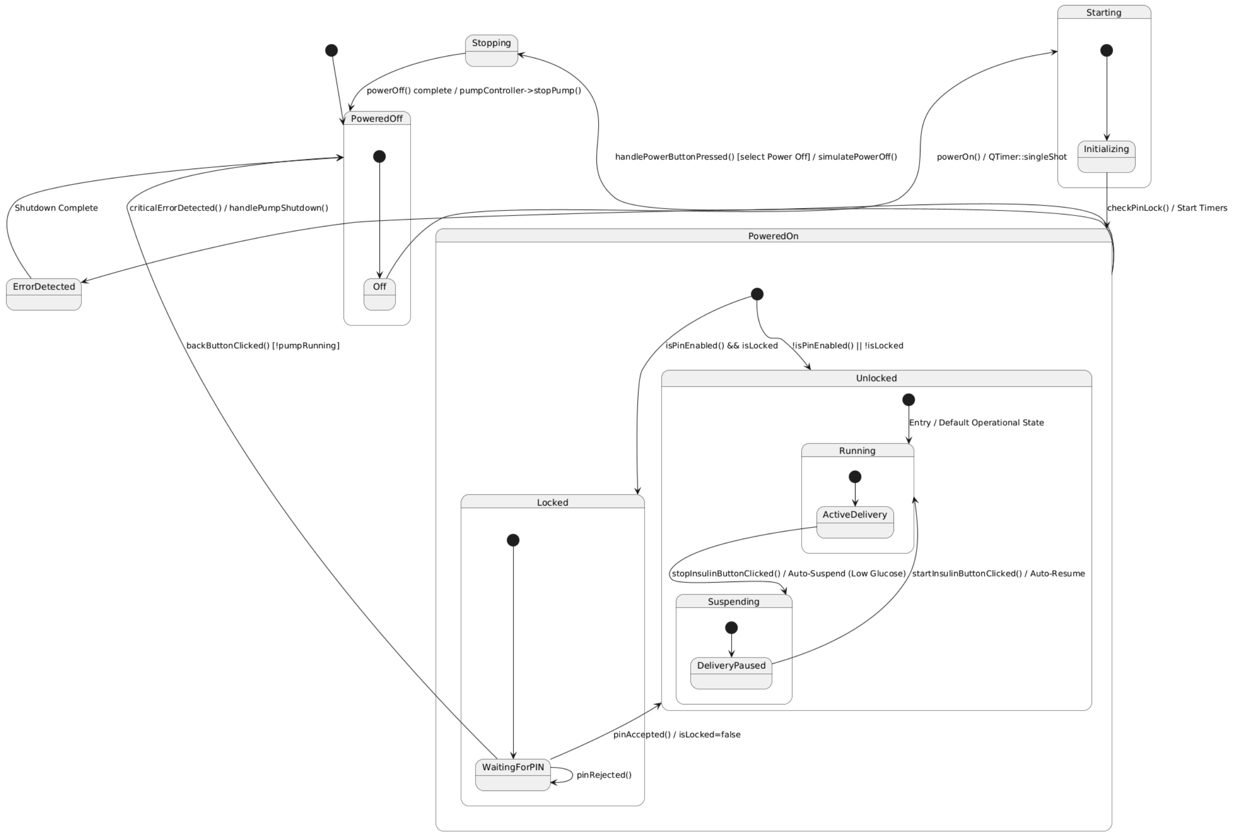
Sequence Diagram 6: Low Battery Alert


:QTimer
:PumpController
:PumpModel
:ErrorHandler
:AlertController
:DataStorage
:MainWindow
timeout() ' Battery drain timer
simulateBatteryDrain()
updateBatteryLevel(currentLevel - 1)
emit batteryLevelChanged(newLevel)
checkLowBattery()
getBatteryLevel()
newLevel (e.g., 19)
alt
[newLevel <= 20]
lowBatteryAlert(newLevel)
logError("Battery low: 19%", "BatteryManager", Warning)
addEventLog(...)
emit errorLogged(...) ' AlertCtrl might listen
emit errorLogged(...) ' PumpCtrl might listen
addAlert("Battery low: 19%", Warning)
emit alertAdded(...)
emit alertTriggered(...)
alt
[newLevel <= 1 ' Critical low]
lowBatteryAlert(newLevel)
logError("BATTERY CRITICALLY LOW...", "BatteryManager", Critical)
addEventLog(...)
emit criticalErrorDetected(...)
emit criticalErrorDetected(...)
addAlert("BATTERY CRITICALLY LOW...", Critical)
emit alertAdded(...)
emit criticalAlertActive(...)
QTimer::singleShot(shutdown)
emit shutdownRequested()
:QTimer
:PumpController
:PumpModel
:ErrorHandler
:AlertController
:DataStorage
:MainWindow

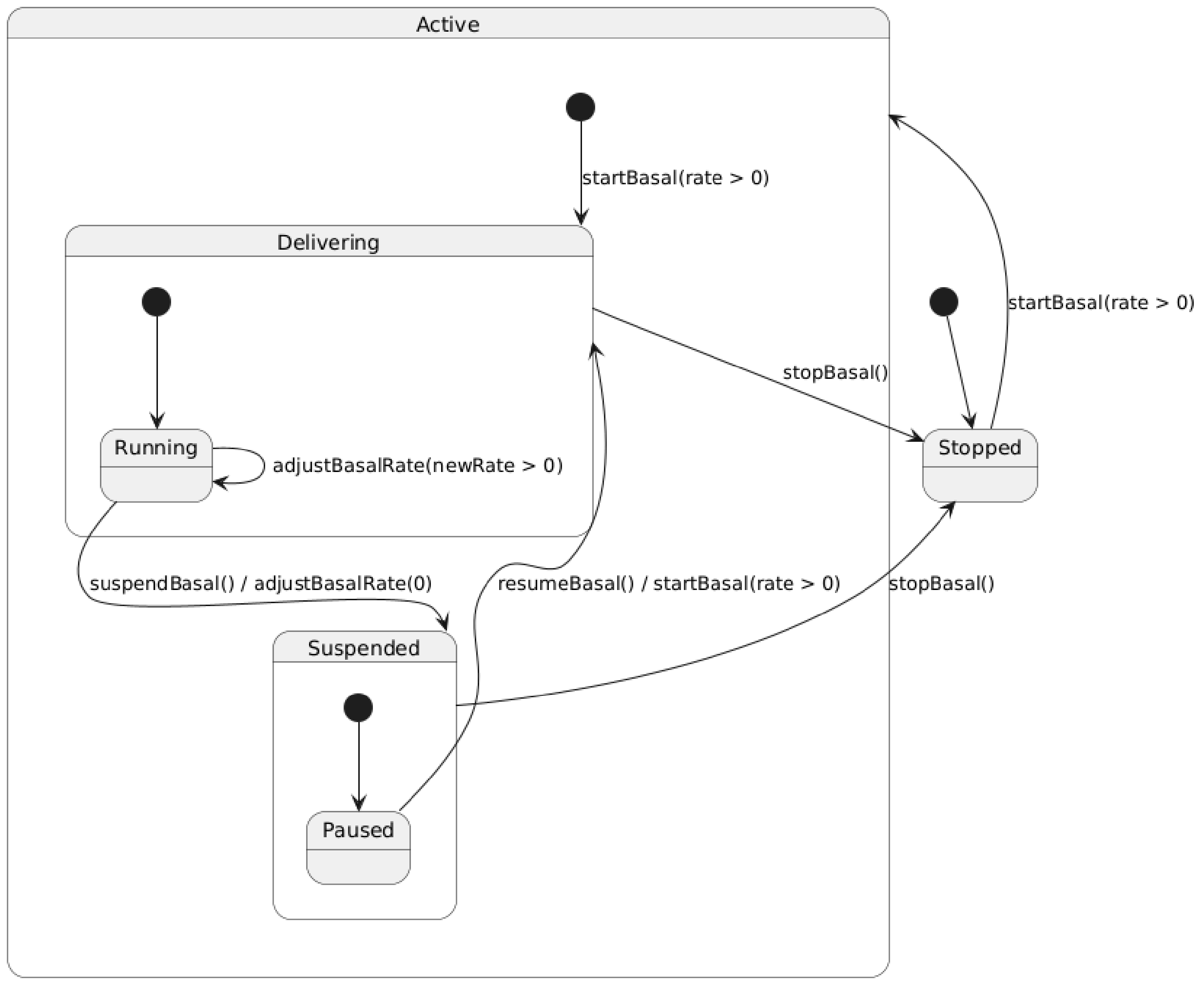
Sequence Diagram 7: Occlusion Detection and Handling


:QTimer
:PumpController
:ErrorHandler
:InsulinModel
:AlertController
:DataStorage
:MainWindow
timeout() ' Occlusion check timer
checkForOcclusion()
Simulates random check
alt
[Occlusion Detected]
occlusionAlert()
logError("OCCLUSION DETECTED...", "PumpModel", Critical)
addEventLog(...)
emit criticalErrorDetected(...)
emit criticalErrorDetected(...)
addAlert("OCCLUSION DETECTED...", Critical)
emit alertAdded(...)
emit criticalAlertActive(...)
suspendBasal()
Stop basal delivery, log segment
emit basalRateChanged(0.0)
emit basalStateChanged(false, false)
emit alertTriggered("OCCLUSION...", Critical)

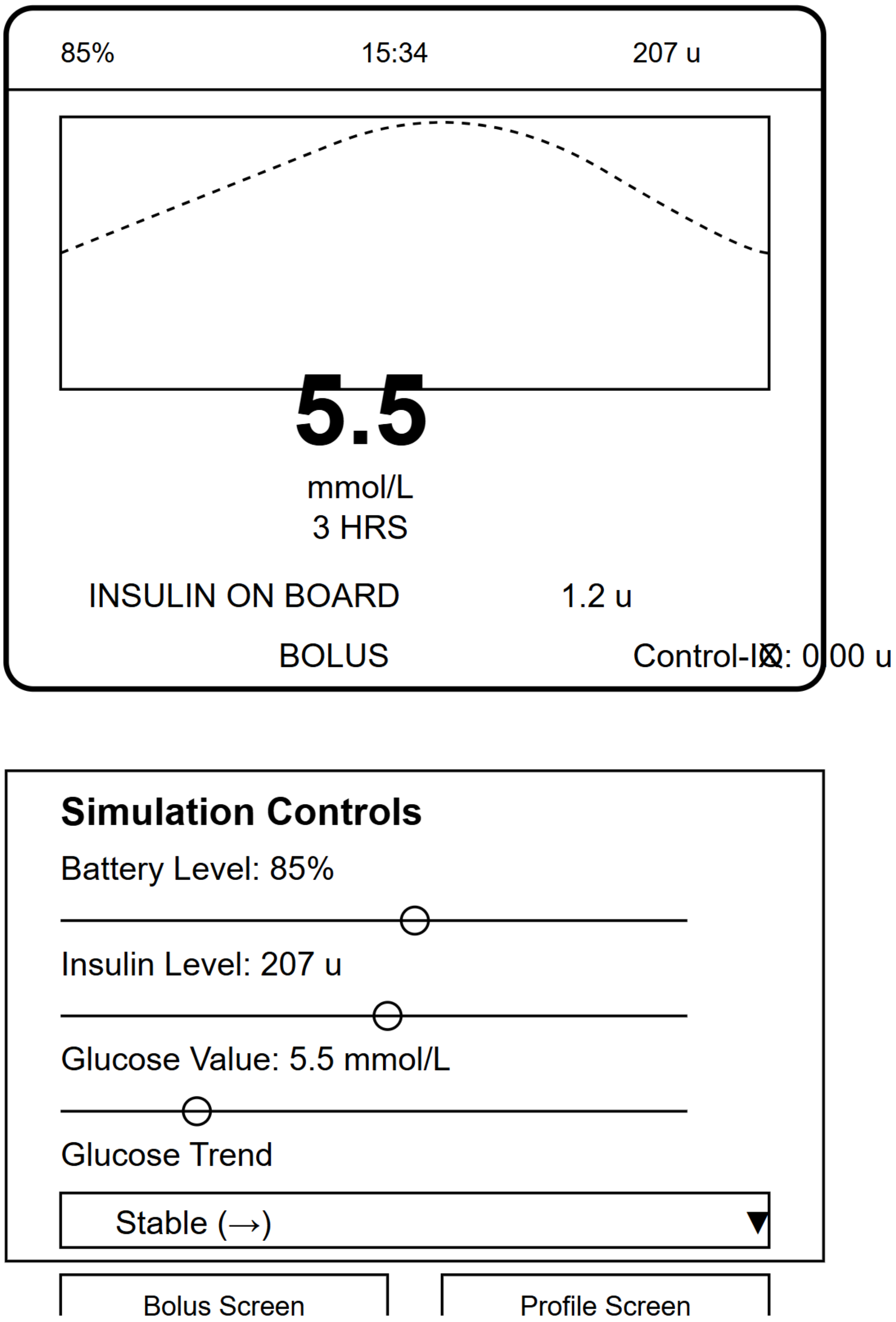
State Machine Diagram for PumpController (Operational State)


Stopping
Starting
PoweredOff
Initializing
Off
ErrorDetected
PoweredOn
Unlocked
Running
ActiveDelivery
Suspending
DeliveryPaused
Locked
WaitingForPIN
powerOff() complete / pumpController->stopPump()
handlePowerButtonPressed() [select Power Off] / simulatePowerOff()
powerOn() / QTimer::singleShot
checkPinLock() / Start Timers
Shutdown Complete
criticalErrorDetected() / handlePumpShutdown()
backButtonClicked() [!pumpRunning]
isPinEnabled() && isLocked
!isPinEnabled() || !isLocked
Entry / Default Operational State
stopInsulinButtonClicked() / Auto-Suspend (Low Glucose)
startInsulinButtonClicked() / Auto-Resume
pinAccepted() / isLocked=false
pinRejected()

State Machine Diagram for InsulinModel (Basal Delivery State)



GUI Sketch



85% 15:34 207 u

5.5

mmol/L

3 HRS

INSULIN ON BOARD 1.2 u

BOLUS Control-IQ: 0.00 u

Simulation Controls

Battery Level: 85%

Insulin Level: 207 u

Glucose Value: 5.5 mmol/L

Glucose Trend

Stable (→)

Textual Explanation of Design Decisions

1. MainWindow as the Central Interface

- **Decision**: The MainWindow class serves as the primary interface, managing the graphical user interface (GUI), processing user inputs, and coordinating with other components like BatteryManager and ProfileManager.
- **Rationale**: Centralizing the UI logic in MainWindow ensures a consistent and streamlined way to update the display and handle interactions. This design choice simplifies the coordination of the simulator's visual elements and user-driven events, making it the hub of the system.

2. Separation of Concerns

- **Decision**: Major functionalities—such as battery management (BatteryManager), user profiles (ProfileManager), and bolus calculations (BolusCalculator)—are split into separate classes.
- **Rationale**: By encapsulating each feature in its own class, the system becomes more modular, easier to maintain, and simpler to test. This approach allows developers to focus on individual components without affecting others, enhancing the overall robustness of the simulator.

3. Use of Aggregation Relationships

- **Decision**: MainWindow aggregates other classes like BatteryManager and ProfileManager using composition (denoted as o--> in UML diagrams).
- **Rationale**: This indicates that MainWindow owns and manages these components, reflecting the simulator's need to oversee and display various pump states. Aggregation provides a clear ownership structure, ensuring that these components are tightly integrated with the UI.

4. Dependency on CGMData for Insulin Delivery

- **Decision**: The InsulinDeliveryManager relies on CGMData (Continuous Glucose Monitoring data) to monitor glucose levels and adjust insulin delivery dynamically.
- **Rationale**: This dependency mimics real-world insulin pumps that integrate with CGM systems to respond to blood glucose changes. It ensures the simulator can adapt insulin delivery based on real-time data, a key feature for accuracy and safety.

5. Error Handling Mechanism

- **Decision**: An ErrorHandler class detects issues like low battery or low glucose levels, triggers alerts through MainWindow, and suspends insulin delivery via InsulinDeliveryManager during critical failures.

- **Rationale**: Centralizing error management in ErrorHandler ensures consistent and reliable responses to safety-critical events. This design prioritizes user safety by halting operations when necessary, such as during hardware malfunctions or health risks.

6. State Machine for InsulinDeliveryManager

- **Decision**: The InsulinDeliveryManager uses a state machine with states like **Idle**, **Delivering**, and **Stopped**, transitioning based on events such as starting delivery, low glucose detection, or manual stops.
- **Rationale**: This state-based approach models the dynamic behavior of insulin delivery accurately, allowing the system to handle interruptions and resumptions logically. It ensures the pump's operation aligns with real-world scenarios where delivery must pause or adjust.

7. BatteryManager State Machine

- **Decision**: The BatteryManager employs a state machine with states like **Discharged**, **Charging**, **Charged**, and **Discharging**, driven by charging events and battery level changes.
- **Rationale**: This design effectively simulates the battery's lifecycle, ensuring the simulator reflects realistic charging and discharging behavior. It provides a clear framework for managing power, a critical aspect of pump operation.

8. Sequence Diagrams for Key Scenarios

- **Decision**: Sequence diagrams illustrate interactions between components for normal operations (e.g., delivering a bolus) and safety scenarios (e.g., low glucose alerts).
- **Rationale**: These diagrams offer a detailed view of how components collaborate during key processes, clarifying the system's behavior. They are essential for understanding complex interactions that aren't immediately obvious from class definitions alone.

9. User Interaction via MainWindow

- **Decision**: All user inputs—such as button presses or dose confirmations—are processed through MainWindow, which delegates tasks to relevant components.
- **Rationale**: Centralizing user interaction in MainWindow keeps the UI responsive and ensures that actions are interpreted and executed correctly. This design provides a single entry point for user commands, simplifying the flow of control.

10. Data Logging and History

- **Decision**: A HistoryManager class logs events (e.g., insulin doses, errors) and supplies data for display via MainWindow.
- **Rationale**: Maintaining a detailed event history is vital for tracking treatment progress and diagnosing issues. This feature meets the simulator's requirement to record and review insulin delivery activities, enhancing its utility for users and developers.