**System Design Document**

**For**

**EGR101 Simulation**

Emily Connearney

Luke Crump

Vivian Dang

Keely Mashburn

Daniel Khalil

|  |  |  |
| --- | --- | --- |
| Version/Author | Edits | Date |
| 3.0/Daniel Khalil, Vivian Dang, Luke Crump, Emily Connearney | Completed main portions of document | 12/4/2021 |
| 3.1/Luke Crump | Edited section 1.2.1 & 1.2.2 | 2/3/2022 |
| 3.2/Luke Crump | Added style changes and touch ups to all sections. | 3/6/2022 |
| 4.0/Luke Crump |  | 4/4/2022 |

TABLE OF cONTENTS

[1 INTRODUCTION 3](#_Toc97458023)

[1.1 Purpose and Scope 3](#_Toc97458024)

[1.2 Project Executive Summary 3](#_Toc97458025)

[1.2.1 System Overview 3](#_Toc97458026)

[1.2.2 Design Constraints 4](#_Toc97458027)

[1.2.3 Future Contingencies 5](#_Toc97458028)

[1.3 Document Organization 5](#_Toc97458029)

[2 SYSTEM ARCHITECTURE 5](#_Toc97458030)

[2.1 System Software Architecture 5](#_Toc97458031)

[2.1.1 Bot Simulation 6](#_Toc97458032)

[2.1.2 Bot Customization 6](#_Toc97458033)

[2.1.3 Bot Wiring 6](#_Toc97458034)

[2.1.4 Arduino IDE 6](#_Toc97458035)

[2.1.5 Arduino Emulation 7](#_Toc97458036)

[2.2 Internal Communications Architecture 7](#_Toc97458037)

[2.2.1 Bot Component Customization 7](#_Toc97458038)

[2.2.2 Bot Wiring Interface 8](#_Toc97458039)

[2.2.3 Bot Simulation 8](#_Toc97458040)

[3 HUMAN-MACHINE INTERFACE 8](#_Toc97458041)

[3.1 Inputs 8](#_Toc97458042)

[3.1.1 Arduino IDE: 9](#_Toc97458043)

[3.1.2 Bot Design GUI 10](#_Toc97458044)

[3.1.3 If the user is selecting Components: 10](#_Toc97458045)

[3.2 Outputs 11](#_Toc97458046)

[3.2.1 Arduino IDE: 11](#_Toc97458047)

[3.2.2 Bot Design GUI 12](#_Toc97458048)

[4 DETAILED DESIGN 12](#_Toc97458049)

[4.1 Software Detailed Design 12](#_Toc97458050)

[4.1.1 Emulation 13](#_Toc97458051)

[4.1.2 Emulation Class Descriptions 13](#_Toc97458052)

[4.1.3 Display 16](#_Toc97458053)

[4.2 Internal Communications Detailed Design 16](#_Toc97458054)

[5 SYSTEM INTEGRITY CONTROLS 16](#_Toc97458055)

System Design Document

# INTRODUCTION

## Purpose and Scope

This document describes the system requirements, operating environment, system and sub-system architecture, files and database design, input formats, layouts, human-machine interfaces, detailed design, processing logic, and external interfaces for the EGR 101 project.

## Project Executive Summary

This section provides a description of the EGR101 Simulation project from a management perspective and an overview of the framework within which the conceptual system design was prepared.

### System Overview

The EGR101 Simulation Software will be used in the Engineering 101 Course to simulate the coursework done with the Parallax BOE-Bots Robot Kits. The three major systems implemented in the project are the working Arduino Emulator and IDE, the robot design/wiring menu, and the 3D Unity Environment to test code and component changes. The Arduino emulator will allow the students to program the Boe-Bot. The design/wiring menu will allow the students to hook up components and wire components to the board. The 3D Unity Environment will allow the user to watch the Boe-Bot complete the course. This product is designed to supplement learning for remote students in the course EGR101.

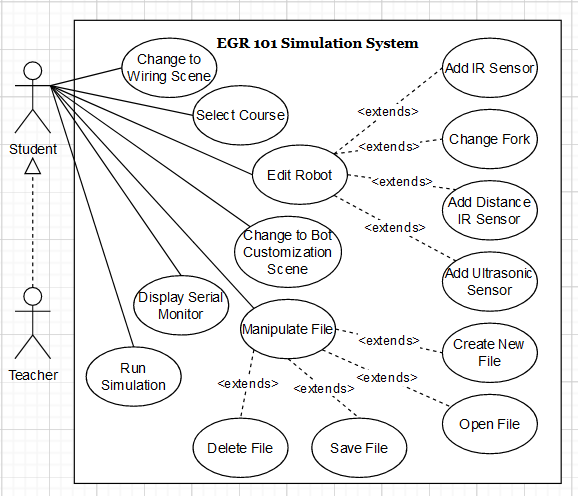


Figure 1. Use Case diagram showing the high-level uses of the system.

### Design Constraints

There are two limitations applied to this project: accessibility and scope. One accessibility limitation is that the simulation must work on both OSX and Windows 10 Operating Systems. The last limitation is the product must work on low-end PCs so students with weak laptops can run the simulation. Emulation of a sandbox-like system like Arduino could be difficult within the timeframe given to complete this project. The optimal solution to this problem would be to only simulate components used within the course.

### Future Contingencies

|  |  |
| --- | --- |
| Contingencies | Workarounds |
| User has unauthorized access to critical internal objects and can modify said objects through the Arduino IDE. | N/A |
| Performance may suffer due to the nature of 3D simulation. | N/A |
| Parallax BOE-Bot, the company that produces the original robots that EGR101 utilized, may decide that our current models are too similar to theirs and will not allow us to use them. | N/A |

## Document Organization

This document is divided into six parts with each part being further divided into sub-sections as needed. The first section of the document is the Introduction, which is then divided into the following subsections: Purpose and Scope, Project Executive Summary (which has its own subsections detailing the System Overview, Design Constraints, and Future Contingencies), Document Organization, Project References, and Glossary.

The second section of the document dives into the System Architecture with its subsections being the System Hardware Architecture, System Software Architecture, and Internal Communication Architecture. The third section of the document details the Human-Machine Interface, with its subsections focusing on the inputs and outputs of the system. The fifth and final section talks about System Integrity Controls; this section does not have any subsections.

# SYSTEM ARCHITECTURE

This section describes an overview of the software architecture for the EGR101 Simulation system and subsystems.

## System Software Architecture

The EGR-101 Simulation Software package is built in Java and C#, utilizing the external software packages RichTextFX, JavaFX, and Unity Engine. This project contains four different individual subsystems.

These include:  
 - Bot Simulation

* Bot Customization/Bot Wiring
* Arduino IDE
* Arduino Emulation

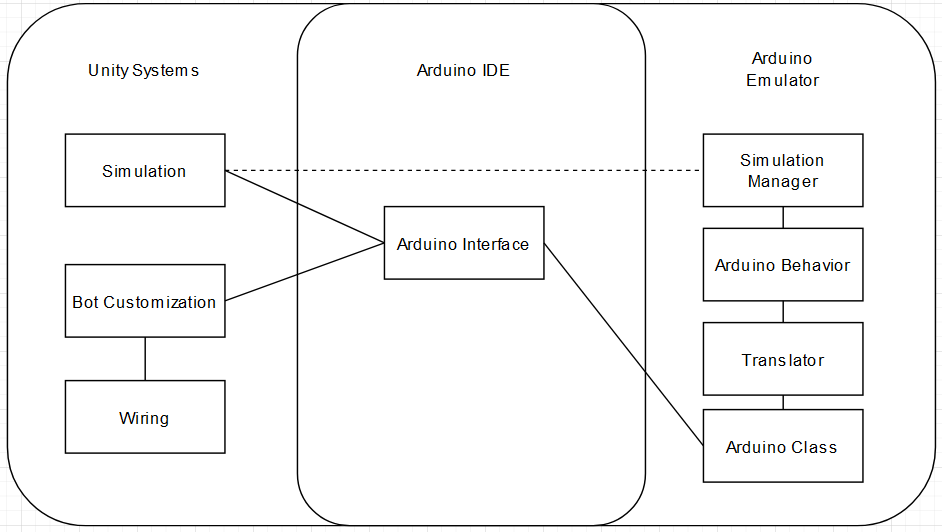


Figure 2. Software Architecture overview.

All of which are displayed using Unity Engine: Bot Simulation, Bot Customization, and Bot Wiring. The other two are programmed in Java: Arduino IDE, Arduino Emulation.

### Bot Simulation

The Bot Simulation module is started from the Arduino IDE if (and only if) there has been code built in the Arduino, and the execute button is selected from the Arduino IDE. A secondary window is displayed as a pre-compiled Unity executable is ran given information of the bot from the java application. The Bot Simulation is a 3D rendered animation of the bot following instructions from the previously coded virtual Arduino and connected components. The bot will follow the directions supplied by the Arduino Emulation via the simulation manager TCP connection between both application threads. Information about each component is sent to the Bot Simulation window updating the bot’s current directions per packet received.

### Bot Customization

The Bot Customization module is started from the Arduino IDE if the Configuration Menu button is selected. This application window is built in Unity as a runnable executable. This menu can update preset components such as different forks, frames, and wheels. The menu also allows for the drag and drop of multiple sensors and general components from the menu to the virtual bot. Each time a change is committed in the menu, a TCP connection to the application manager will update the base Arduino configuration.

### Bot Wiring

The Bot Wiring module is started from the Bot Customization scene when the user changes the view of the camera. This menu allows the user to connect pins on previously added components. The user can click and drag connections between pins on the Arduino and pins on specific components. Connections are saved to a \*.txt document and will be read by the Arduino Emulator at a later time.

### Arduino IDE

This project is executed from a java executable which will display an in-house interpretation of Arduino IDE. The Arduino IDE allows the user to write, build, and or execute code (Actions like build would update the Arduino Emulator with written behavior if logically viable, and actions, like execute would run the Arduino Emulator’s behavior. The Arduino IDE will be the initial hub of communication between subsystems from the user’s point of view. There will be user interfaces accessed from this menu that allow the user to run simulations, build/update emulator code, and add/customize/wire components on the virtual bot.

### Arduino Emulation

The Arduino IDE will create a single instance of an Arduino Board Object which is constructed with an array of Analog and Digital IO pins, Ground pins, 5-Volt pins, and 3.3-Volt Pins. Using class methods (digitalWrite(), analogWrite()) from this object the Arduino class can write voltages to these pins. Pins are built with a doubly linked-list structure such that a pin has a next and a previous reference to other pins with a default null value for each attribute. This pin structure allows for voltage to be sent up the chain to each next node facilitated by the simulation manager. The Arduino Emulation has reference to an array list of abstract components that can represent real life components IE: LEDs, Sensors, Motors, etc. Each of these components have their own array of pins, and requirements to be powered whether that be through simultaneously holding pins with enough voltage, or access to ground.

## Internal Communications Architecture

The EGR101 Simulation Software handles communication through a Java Application Manager class. A Server Socket is opened from the Java application and used to communicate via TCP in 3 different scenarios.

### Bot Simulation

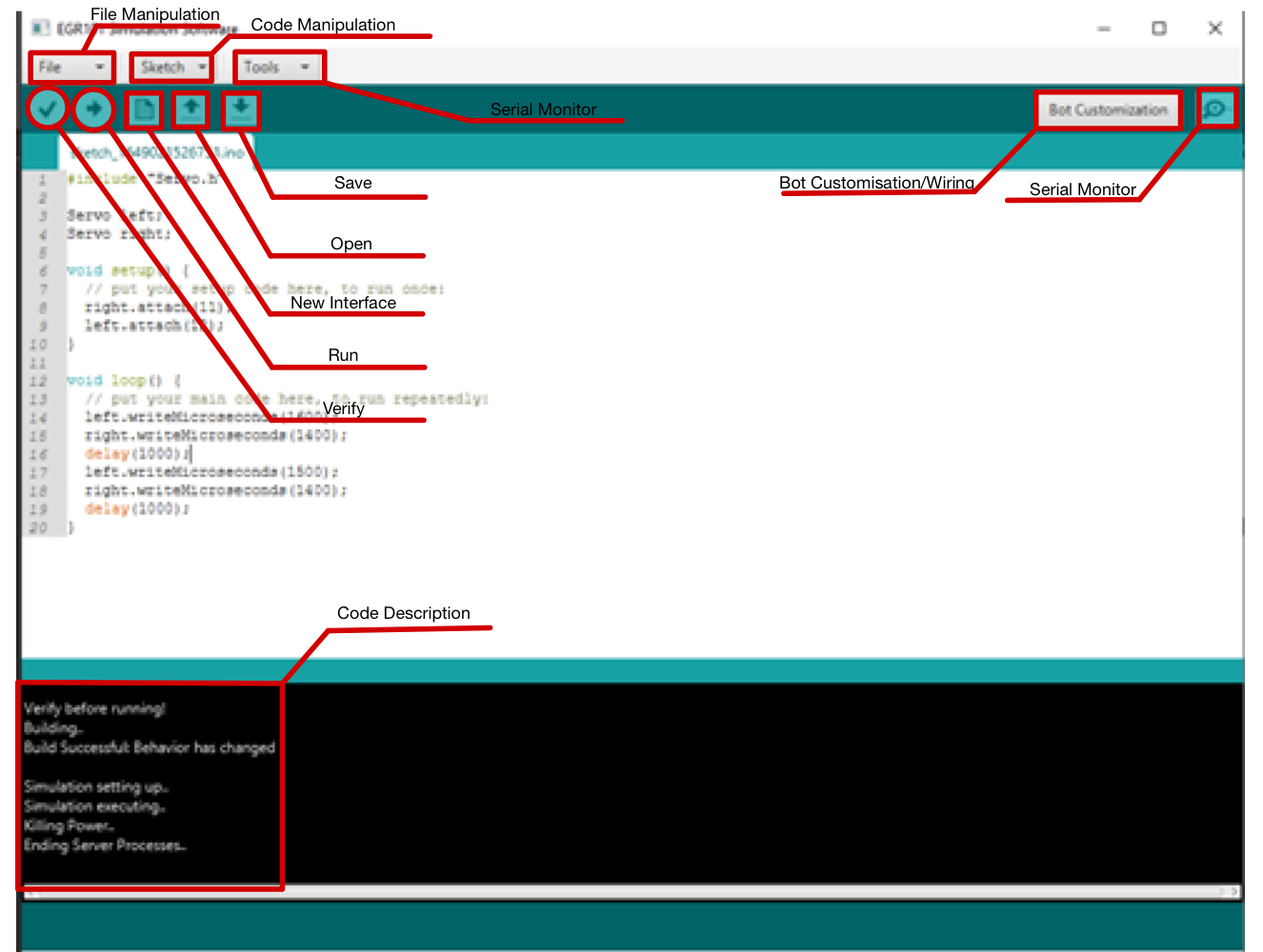
* Packets are sent from Bot Simulation to update the Arduino Emulator with Sensor Inputs that may update Arduino state.
* Packets are sent from the Arduino to communicate component behavior that shall be reflected in the bot simulation display.

# HUMAN-MACHINE INTERFACE

This section provides details of the inputs and the outputs as they relate to the user operating the EGR101 software.

## Inputs

User input will come in the form of interactions with the different UI elements depending on the interface. These will be broken up into these interface sections.



### Arduino IDE:

#### The user can input code into the text editor

#### The user can select the build button

#### The user can select the run button

#### The user can select buttons regarding save/save as/ open

#### The user can select the bot design button.

### Bot Design GUI

#### The user can select individual components from the bot by hovering over and clicking those components.

#### The user can select menus:

* 1. Presets
  2. Wheels
  3. Forks
  4. Chassis
  5. Components
  6. Sensors
  7. General

### If the user is selecting Components:

#### The user can drag and drop from the menu Items with pictures under either Sensor or General

#### The user can drag and drop over the chassis of the bot.

#### The user can drag a connection from the 3D pin to any component pin.



## Outputs

Outputs described in this section will occur after associated inputs. Refer to 3.1 for these inputs.

### Arduino IDE:

#### Code will generate as typed into the TextArea. Specific keywords will be highlighted based on the highlighting done in the actual Arduino IDE.

#### The build will occur if there are no syntax / logical errors present in the TextArea.

#### The run button will start the simulation and open an instance of Unity displaying the bot and moving it based on instructions and connections to components.

#### Save will save the current file if it has already been saved, Save As will save the current TextArea to a file location on the local system. Open will swap the code within the TextArea with one specified by the user.

#### The bot design button will display an instance of Unity that will allow the user to update component state and wiring state.

### Bot Design GUI

#### After the user has selected the component, the info panel will update with information related to the component selected.

#### Menus selected by the user will update a view of their dropdowns or (if a component or preset) will update the images within the toolbar with related components or presets.

#### An instance of the selected menu items will instantiate in 3 dimensions and be allowed to snap to the bot.

#### After releasing a drag and drop onto the bot, the component added will become a child of the bot object

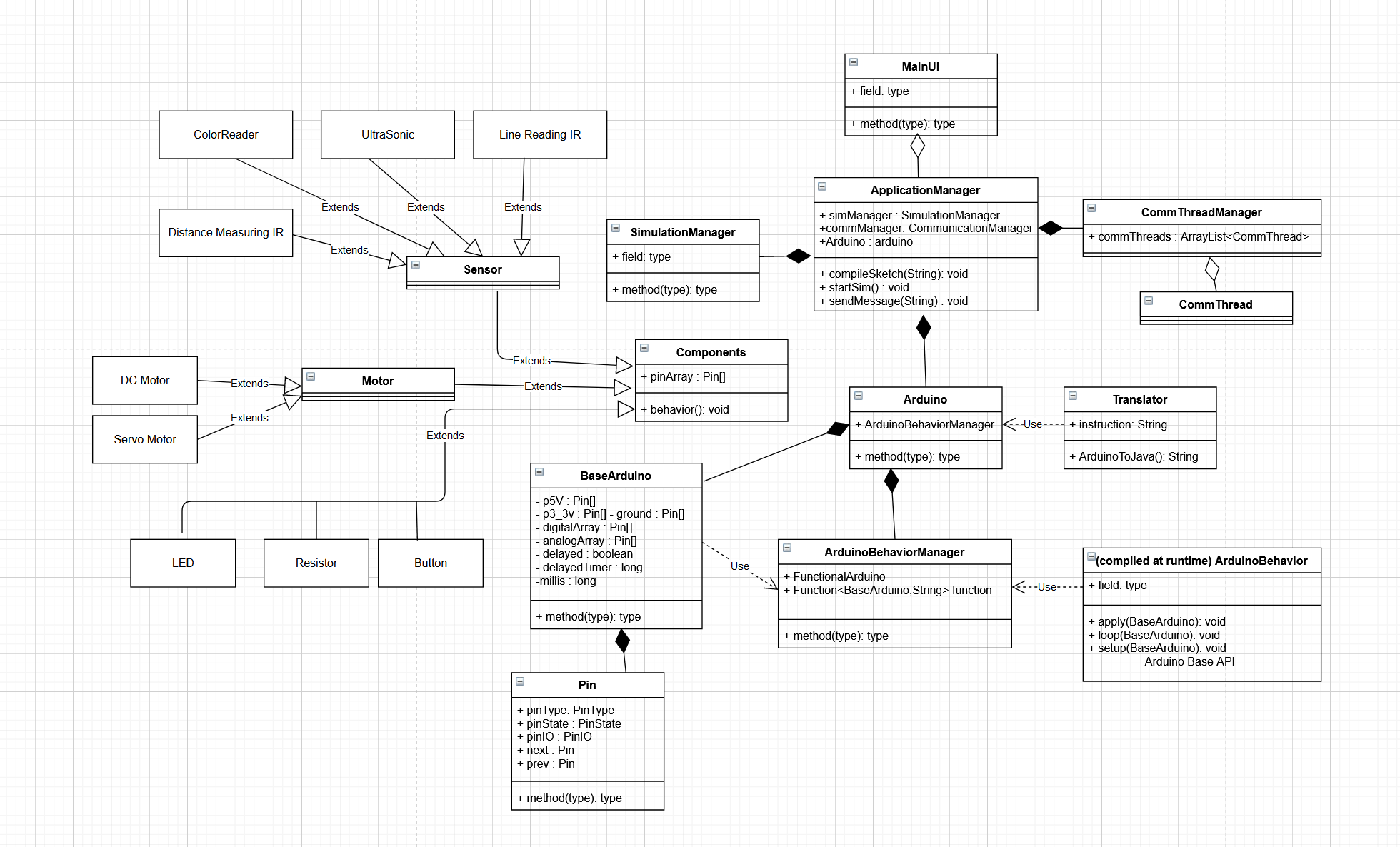
#### Dragging connections will display a line connecting the component pins or Arduino pins together.

# DETAILED DESIGN

This section contains detailed information about the software design of the EGR101 Simulation System.

## Software Detailed Design

This section will be broken up into two separate sections dealing with the outward facing display interface and the inner emulated system. The display section will contain a breakdown of the scripts and Java files relating to EGR101 Simulation display, bot design, wiring, and Arduino IDE. The emulation portion will describe the software of the simulated Arduino, its behaviors, and its simulation.



### Emulation

The figure above represents the UML diagram of the EGR101 Simulation Emulated Arduino code, and related management objects like the Application manager, and Simulation manager.

### Emulation Class Descriptions

**Pin** is a class utilized by BaseArduino and abstract Component classes. Pin contains 3 different ENUM types, pinType, pinState, and pinIO. Other attributes include resistance, voltage, and current double values, a reference to the next and previous Pin and finally a Boolean value that indicates whether the pin is local to Arduino. Pin objects are instantiated into arrays in the BaseArduino class. And are used like chains of connections between the local Arduino to components.

**BaseArduino** is a class that manages digital IO, analog IO, ground, 5V and 3.3V pins. Base Arduino allows for calls that can write voltages to digital IO and analog IO pins. The class contains methods similar to the base Arduino function calls. Functions like digitalRead(), digitalWrite(), pinMode(), analogRead() analogWrite(), etc. Are essential functions for the Arduino to send power to components connected to real life Arduino. This class acts as an object simulating a real life Arduino, as well as acting as the basic API for digital, analog IO manipulations, and time manipulations like delay(). These methods are used by the translated behavior class as an Application Interface to modify pinState and time at runtime. This class is first referenced by the Arduino class, which adds additional functionality to the BaseArduino class.

**ArduinoBehaviorManager** is a class that facilitates at-runtime-compilation of instructions given by the Arduino IDE. The ArduinoBehaviorManager compiles a constructed java class that extends the Function interface. This Function interface is what allows the Arduino class to reference the compiled user code. This class “compiles” the instructions presented as a string to the compile function by first compiling the string entered under the class name “ArduinoBehavior”, then setting a reference of the function to a class attribute. This attribute has an associated accessor and modifier in which the rest of the program can getFunction() to receive instructions that will modify the BaseArduino class.

**Arduino** is a class that contains references to BaseArduino, ArduinoBehaviorManager, and an ArrayList of components. This allows simulation manager or application manager to modify the Arduino’s components, connections, and behavior through the modification of this object. The methods supplied throughout the Arduino class deal with managing each of these attributes. There are methods associated with adding connections between pins, adding components to the ArrayList of components, calling setup() and loop() from the Function interface in ArduinoBehaviorManager, and finally compileSketch() which takes a parameter of Arduino code and utilizes the translate class to translate it to Java code, and modify the ArduinoBehaviorManager class with an updated Function.

**Serial** is a static class that mirrors the Arduino API Serial module. Methods include begin(), println(), print(), and flush(), which all contain the same functionality as Arduino API.  
  
**Servo** is an object that could be instantiated from the IDE that contains a reference to Base Arduino and can be attached to a pin to act as a servo motor. This class mirrors the Servo class within the Arduino API, so most of the methods throughout the code are similar. To list the ones synonymous to Arduino API: attach(), write(), writeMicroseconds(), read(), attached(), detach() are all utilized to modify the state of the Servo object. To utilize methods from the Servo class one must attach it to a pin (could be analog or digital). After doing so, the user could write angles or microseconds to that servo. This class only works if there is an instance of a component that is of type ContinuousServoMotor that is powered with 5V, and ground and IO pin must match the attached pin.

**Translator** is a class utilized by the **Arduino** class, that has an attribute called instructions. The constructor for Translator has a parameter that inputs a string of Arduino syntax and converts the text to a viable java class that extends the Function interface. The Apply Function implemented from the interface is defined by using both the setup and loop function depending on the number of times it is called. This class is only used to translate code to java and serves no other purpose.

**Component** is an abstract class used to have extensions that inherit its attributes and methods. Each component must inherit an array of pins, a (double) voltage limit, a (double) current drain, a state (String used to send information via TCP), and 2 Boolean values denoting whether the component is powered and grounded. It’s abstract methods include checkState(), and behavior() which is used within the Simulation Manager and Arduino.

**ContinuousServoMotor** is a class that extends component and defines functionality for a virtual Continuous Servo Motor. It has 3 pin connectors that are 5V, Ground, and IO. This component will only work if there is 5V, and ground connected and will provide a written angle based on whether the pin has been written an angle through the writeMicroseconds, or write command attached to an instance of Servo.

**Led** is a class that extends component and defines functionality for a virtual LED. It has 2 pin connectors that are [some voltage] and ground. This component will only work if there is a connection with a voltage less than 2.2V and above 1-V, and a ground connection.

**DistanceMeasuringIRSensor** is a class that extends component and defines functionality for a distance measuring IR sensor. It has 3 pin connectors with 5V ground, and an output pin. The sensor will not work if 5V and Ground hasn’t reached the component pins. This component will update state based on distance to voltage formula for distance measuring IR sensors.  
  
**LineReadingIRSensor** is a class that extends component and defines funcitnoaliy for a line reading IR sensor. It has 3 pin connectors with 5V ground, and an output pin. The sensor will not work if 5V and Ground hasn’t reached the component pins. This sensor will provide a voltage based off of the light level of the surface it is facing.  
  
**UltraSonicSensor** is a class that extends component and defines functionality for an ultrasonic sensor. It has 4 pin connectors with 5V, ground, input, and output pins, The sensor will not work if 5V and ground hasn’t reached the component pins. This sensor will return a time in microseconds for its return which can be dropped into a formula after receiving the signal.

**SimulationManager** is a class within the Base Arduino Emulation that facilitates component behavior, pin behavior, and overall simulation. Per iteration it updates each pin and its chain of connecting pins based on the local Arduino connections power. For example, if a 5V pin is connected in a chain of 3 pins, each pin in the chain will receive 5V. After updating pin power, the component state is updated. Components will display behavior based on their preconditions for working. (For example, LEDs need a sufficient voltage and any ground connection). After the component state has been updated, the component behavior is executed which will determine whether sensors read values, LEDs turn on, or Servos move. This class is essential in running the full simulation and connecting important parts of the project together.

**ApplicationManager** is a class that handles the full execution of the EGR101 Sim application. ApplicationManager has an instance of SimulationManager, Arduino, and ServerSockets. This interface is accessed through the MainUI to build and execute code.

### Display

There are 2 Unity Executables and 1 JavaFX executable that are accessed within the EGR101 Simulation project. The JavaFX Executable is a build of the Arduino IDE that contains an instance of the Arduino Emulator. The Arduino IDE contains a class called MainUI.

**MainUI** is a GUI class for the EGR101 Simulation IDE. This class has an instance of Application Manager and will call its methods based off User Input. This class runs syntax parsing to color words the same way the legacy Arduino IDE would and allows the user to Save/Save As/Open files. This program also allows the user to build code, execute code, or open a design/wiring interface to customize the virtual bot.

## Internal Communications Detailed Design

The internal communications of the EGR101 Sim project are handled within the ApplicationManager class.

ApplicationManager contains 3 ServerSockets that could simultaneously communicate on 3 different application threads. The Java application acts as the server within this communication and sends and receives information via DataInputStreams and DataOutputStreams from sockets setup in 3 different C# scripts within the Unity Engine. These connections occur once the User opens either Simulation, Bot Component Customization, or Bot Wiring Menus and sends packets through a TCP connection. Communications between running application threads follows the structure of section 2.2. The wiring interface sends a list of connections via packets, given each pin’s unique identifier. The bot customization also handles updates on emulation state with component unique identifiers. And finally, throughout Simulation, each component that is powered within the system is sent with a particular numerical state indicating the behavior of the component in 3D space. For example, Servo motors will send packets by splitting a string into this format:  
  
 ( componentIndex, componentType, writenAngle)

# SYSTEM INTEGRITY CONTROLS

This system does not handle sensitive information, although there are system integrity controls in place to limit control of the user to break program state (via compiling Arduino code through the IDE portion of the project). For the program interface to compile Arduino Code to Java code, a reference must be supplied to the generate Java code. This reference is the BaseArduino object. This means that the user could possibly re-instantiate the BaseArduino or set it to null. Since BaseArduino has multiple threads dependent on its function, this can cause internal harm to the application. Issues similar to this are present throughout the project. This leads to extremely strict access controls. If anything can be private it should be private.