Software Design

Sergio van Amerongen
0952200

Stefan Cloudt
0940775

Daan de Graaf
0956112

Robert van Lente
0953343

Tom Peters
0948730

Berrie Trippe
0948147

Responsible:
Stefan Cloudt
s.d.cloudt@student.tue.nl

March 11, 2016

1 Introduction

This document will describe the design decisions made while building the software, based on the UPPAAL model specified during the software specification phase. It will also include an early draft of the software that will control the machine. This software will be written in Java, which means that we are able to use the object oriented programming paradigm. The object oriented design of our program is described in a class diagram.

2 Object Oriented Design

We have chosen to implement the states using the object oriented programming paradigm. The state implementations follow directly from the finite state machine described in the software specification. The class diagram of figure 1 provides an overview of all states and auxiliary classes. In the diagram the '+' symbol indicates a publicly accessible method or property, '-' indicates a private accessible method or property, bold indicates a constant property or a method which cannot be overridden and underlined indicates a property or method which is accessible through the class

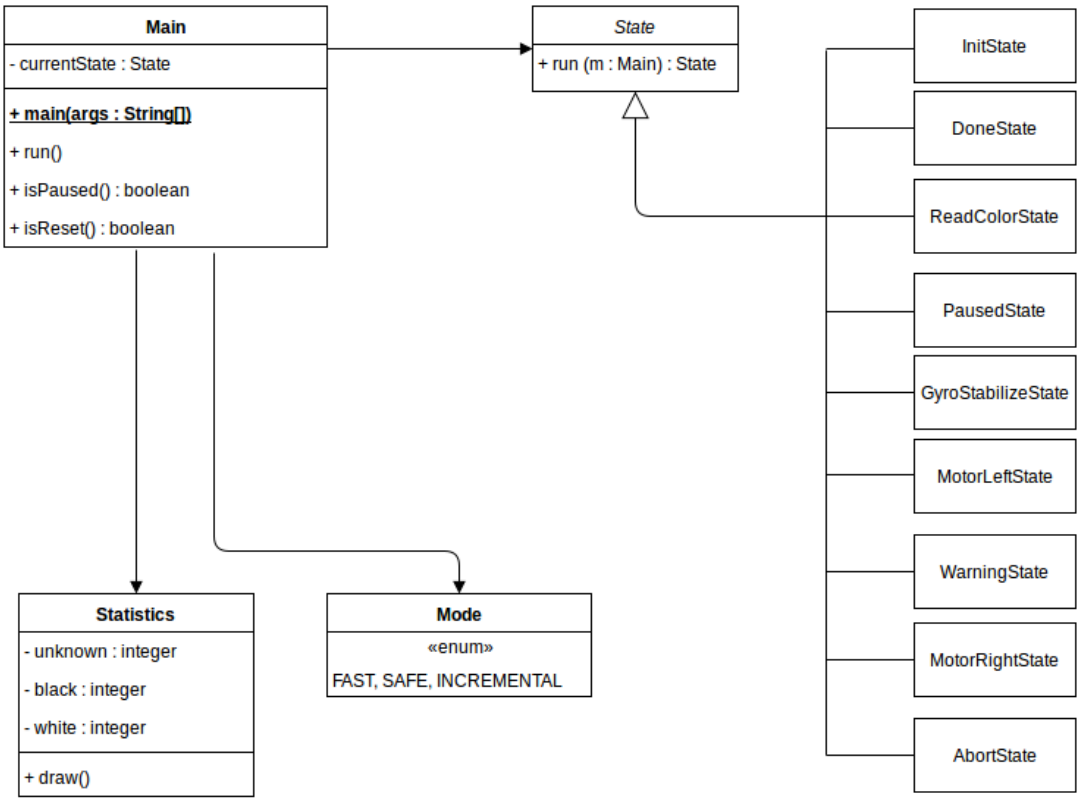

Figure 1: The class diagram

2.1 Main class

The main class handles button presses, updates the display and executes the 'run' method on the current state. It also exposes properties which provide access to the actuators and sensors. These are objects from the LeJOS API. The properties providing this access are motor, gyro, color, aButton (the abort button), spButton (the start/pause button) and rButton (the reset button). These properties are declared public to provide easy access, and these are declared final to make sure the properties cant be changed.

Then we have two more properties, paused and reset. These two booleans correspond to the paused and reset flags of the software specification. These flags can be accessed by getter and setter methods.

Furthermore we have a property storing the statistics object of type Statistics. This is a final and global property.

Finally a property indicating the current mode of the system is provided. For this property a getter is provided.

2.1.1 Run method

The main method of the class Main is the method which is claled when the program is started. That main method calls the run method of the Main class. The run method contains the main loop of the algorithm.

```
1: while true do
2:     if Abort button pressed and current state is not abort state then
3:         Current state = new AbortState()
4:     else if Start/pause button is pressed then
5:         paused = true
6:     else if reset equals true then
7:         reset = true
8:     current state = current state.run(this)
```

2.2 Statistics class

The statistics class is meant to store three counters for unknown, black and white discs. Although it is not shown in the class diagram of 1, it does provide mehtods to modify these counters.

2.3 Display class

The display class has some methods to draw output on the LCD screen of the brick. It has a method for normal running, for errors and warnings and for the mode-pick menu.

2.4 Mode enumerable

The mode enumerable can be set to either fast, safe or incremental. It stores the current mode in the Main class, which can be referenced by all states for use in a transition guard.

2.5 Abstract class state

This abstract class is the base class for all states. It defines the *nextState* method and provides a default display showing the current state and the number of sorted discs by colour. Other states can also opt to override this method and implement a custom display, such as a warning. The *nextState* method returns the next state of the machine, which may also be the same state if none of the guards are met.

2.6 State implementation classes

Nearly all states in the Uppaal state machine have a counterpart in java, although some of them have been merged, as we do not have to transition to a state to send a signal.

2.6.1 WarningState class

In the finite automaton of the Software Specification there are multiple warning states. All these warning states have in common that they have one or more transitions to the warning state, then a warning state which displays the warning on the display of the Brick, and after that it goes with one transition without a guard to some other state. The WarningState class takes on construction of a new object a parameter of the type Warning and a parameter indicating to which state to go after the warning. The warning object contains information about the warning and the text to display on the display.

The finite automaton contains multiple warning states. All of these have in common that they are transitioned into from one of the operating states, then display a warning message and return to the operating state they were entered from. We have implemented this using one WarningState, to which we pass a the relevant warning type and the state to return to. The warning state will then display the warning and transition to the return state.

2.6.2 AbortState class

The abort state class differs in the sense that it has an additional parameter on construction indicating the type of error which occurred. The abort state object then uses the information inside that object to display the appropriate message.

The *AbortState* takes a parameter indicating the type of error to report. It overrides the *displayUpdate* method to display the error message until the reset button is pressed.

3 Arguing correctness

We need to argue three things to argue that the software design is correct and will work:

1. The states are implemented as in the state machine
2. The button flags are set as required
3. The abort button triggers the system to go to the abort state if it is not yet in the abort state

3.1 State implementation

The Main class holds a pointer to the current state. On initialization the current state is set to a new ModeSelectionState object. This is done in the constructor of the Main class. Therefore the system starts in the initial state just like the finite automaton of the Software Specification. Then the main loop iterates forever and sets the current state to the result of the method run of the current state object. The run method of a state object should as documented do its task, then check the guards of the transitions and after that do a transition if the guard is satisfied. Doing a transition implies returning a state object which is the state the system transitions to. Then the current state pointer is set to that returned state object and next iteration the run method corresponding to the next state is executed. When the run method of a state object behaves as described and implements the state, its guards and transitions correctly, then it follows that the system behaves as expected, because in that case it is the same as the state machine we already have proven.

3.2 Button flags

We want to argue that when the buttons are pressed, the flags are set in a short amount of time. The loop of the run method first checks if the abort button is pressed. Then if the abort button is not pressed or if the current state is the abort state then the start pause button should be checked. Since the abort button has a higher priority than the start pause button this is no problem. So the paused flag is set to true when the start paused button is pressed, when the abort button is not pressed or when the current state is the abort state. Then if all of these cases do not occur, then the reset flag needs to be set if the reset button is pressed. Therefore the loop sets the flags as required, given that the run method takes less time than it takes to press a button. If a user presses a button longer than the running time of the run method, then the flags are set correctly. Because the run method is very short, the time it takes to press a button will always exceed the time it takes to execute one iteration of the run method.

3.3 Abort button

The loop first checks if the abort button is pressed and it checks if the system is not in the abort state. Then, if this is the case, then the current state pointer is set to a new abort state object and the run method of that abort state object is run and therefore the system is in the

abort state. Again we have the constraint that the run method should run in less time than an average human presses a button. However, this isnt a problem as said before.

4 Conclusion

This document has provided a diagram depicting the class structure of our object oriented program. This diagram serves as the basis for the software that will be fully constructed in the software implementation phase. We also created an initial version of this software and proved its correctness in this document. This is an important step in finalizing the software that will control our machine.