System Implementation

(c. 618 words)

This README file is for the Python code main.py which is an implementation of the project design assignment from Week 7. Since then, while the code still attempts to follow the general object-oriented programming philosophy (Zeil, 2013), the class diagram has been rather overhauled; see [Class Diagram](#_Class_Diagram).

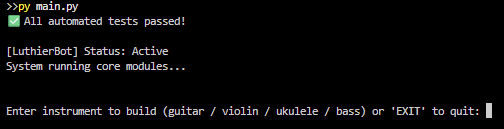
Some automated testing has been implemented via Python’s *assert* statement before the code runs, under the if \_\_name\_\_ == “\_\_main\_\_”: condition.

# Prerequisite

On Windows, check that Python is installed (version 3.13.5 was used for this .py file):

python --version

# Usage

1. Download the ZIP file, unzip it, and open a terminal here e.g. by right-clicking inside the folder.
   1. Alternatively, open a new terminal and navigate to the folder containing main.py.
2. Run the application in the terminal: py main.py
3. When prompted, type “guitar”, “violin”, “ukulele” or “bass” (or type “exit” to exit the program altogether):   
   
4. Suppose we enter “guitar”. The LuthierBot (“R”) will travel to individual spots around a workshop simulated with the following grid:   
   [ ] [ ] [ ] [ ] [S]

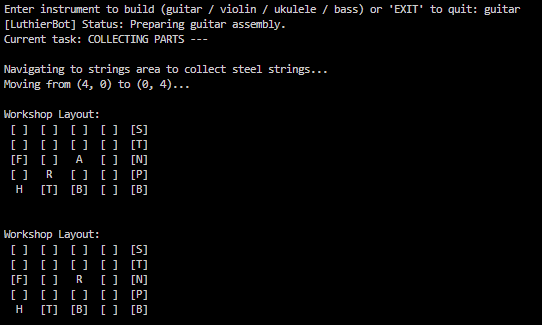
[ ] [ ] [ ] [ ] [T]

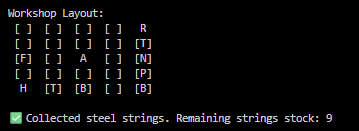
[F] [ ] A [ ] [N]

[ ] R [ ] [ ] [P]

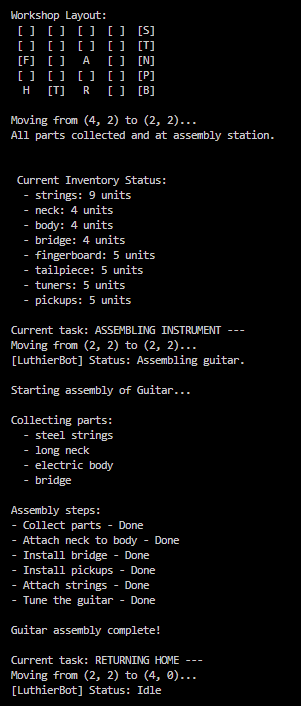
H [T] [B] [ ] [B]

[Here, H=Home, R=Robot, A=Assembly\_Station. All other letters inside square brackets refer to part boxes.]

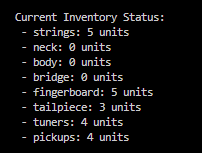


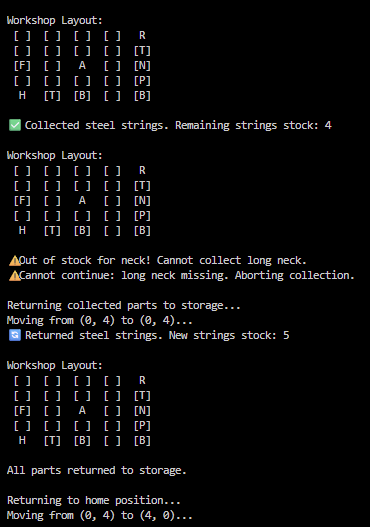
The LuthierBot will travel to the boxes around the workshop gathering the required parts to build this guitar, and update the inventory: 

1. If all components are successfully gathered, the LuthierBot will go to the Assembly Station in the centre. It will print a current status of the workshop’s inventory and begin assembling the guitar.



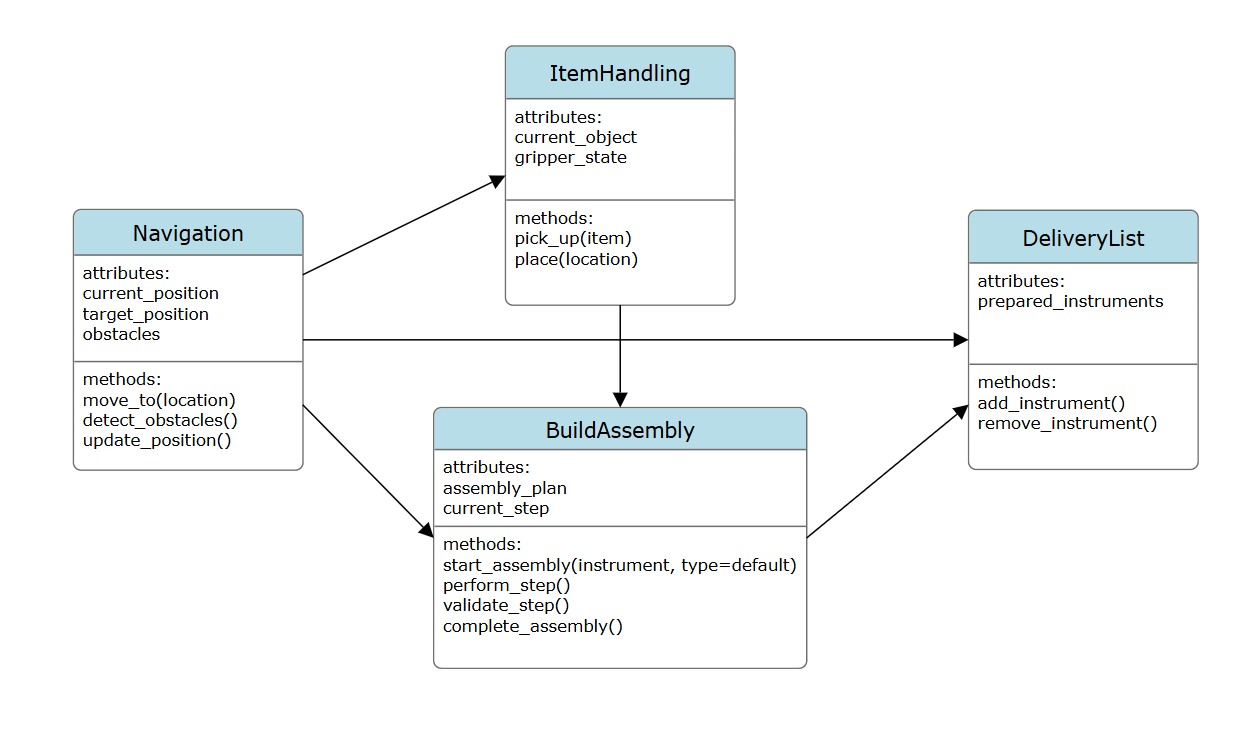
1. It will prompt the user to build a new instrument or exit the program.

As mentioned in Step 5 above, the LuthierBot maintains an inventory. If at some point it reports that there is no more of certain parts (see right), then when it is prompted to construct a new instrument with those missing parts, it will flag an error and return any parts collected:

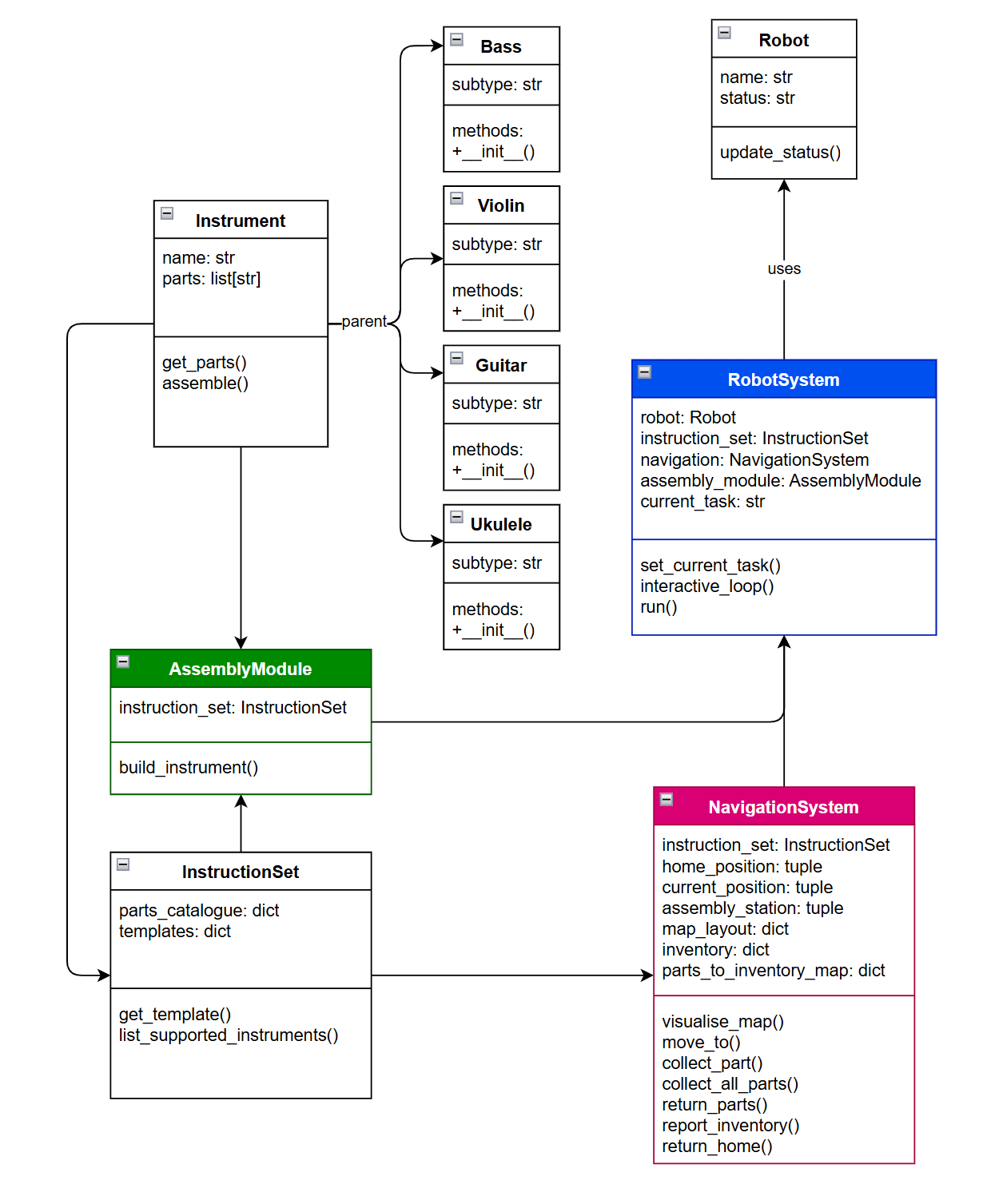


# Changes from the original proposal

## Class Diagram

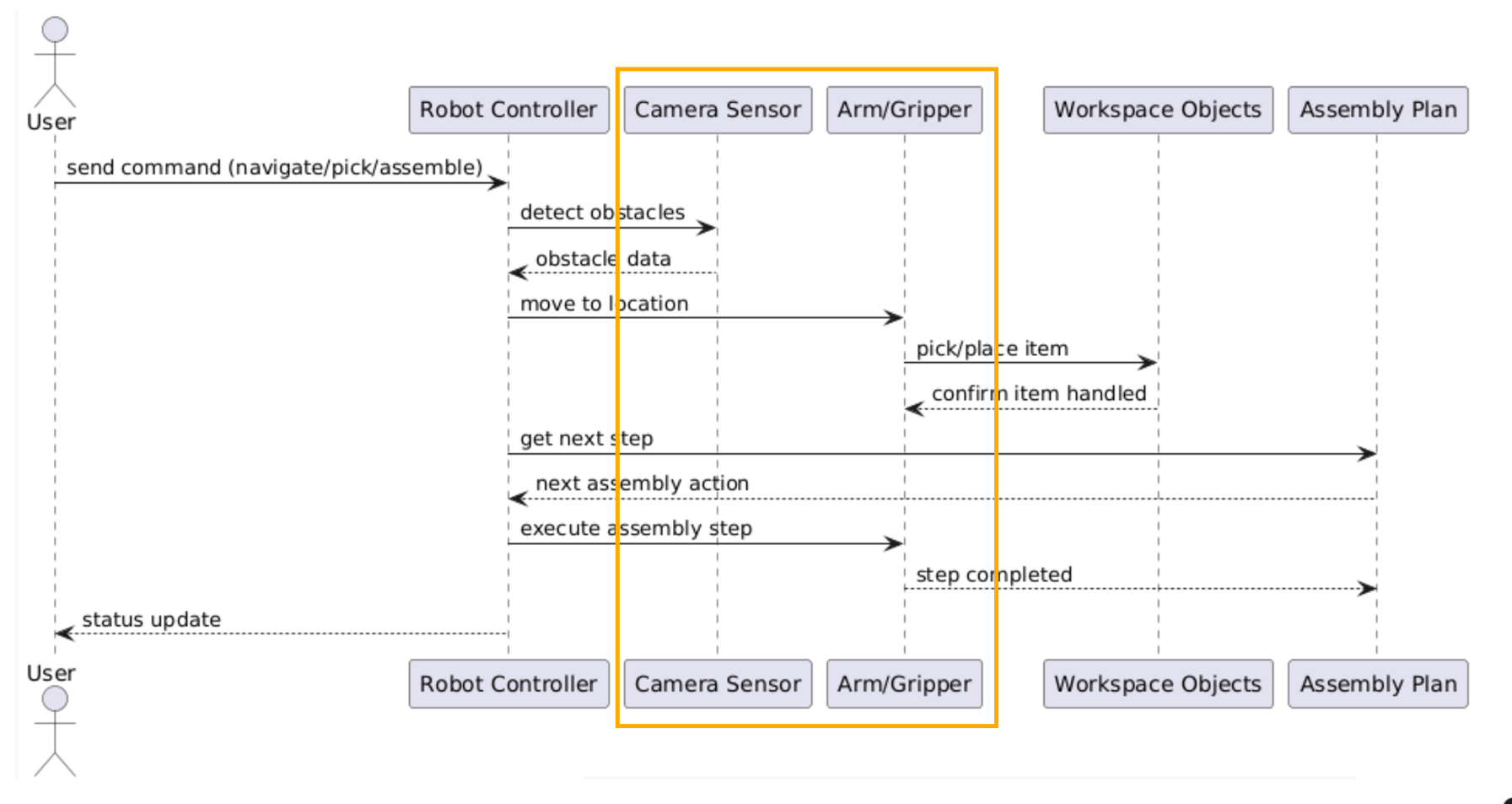
The original class diagram was presented as follows: 

For these to properly communicate with each other, however, more objects have had to be implemented in the system:



The three core functions from the proposal remain. *AssemblyModule* (formerly *BuildAssembly*) has been simplified – it only builds instruments when all parts are collected, removing redundant step validations. *ItemHandling* is now handled with *RobotSystem*, which now controls the overall workflow and decision-making management, further reducing redundancy via centralisation (Gollwitzer, 2024). *NavigationSystem* (formerly *Navigation*) is largely unaltered, although extra attributes and functions were added to reflect the robot’s status and its current step in an instruction set.

## Sequence Diagram



A *Sensor* class was originally planned but later dropped. While the idea was to have it detect obstacles dynamically, it proved difficult to integrate cleanly with the rest of the system without rewriting how navigation worked. For simplicity, the current version assumes the environment is static and predictable.

## Jupyturtle

This was rather difficult for me to code in, personally. While it worked fine in isolation, it kept either freezing or failing to sync with the robot’s actions when integrated into the wider program. The workaround for this was to integrate the map into the GUI via ASCII-diagramming instead, which was both easier to program and troubleshoot.

# If I had more time:

* Add more objects to manage the extra functions for *NavigationSystem*, so that the latter is not overcluttered.
* Implement a dedicated *Pathfinder* object, particularly for rerouting the robot if an obstacle stands in its path.
* Implement a persistent map and inventory list as separate files that the system then reads and makes alterations to.

# REFERENCES

Gollwitzer, Z (2024) *OOP Principles: What is Object Oriented Programming?* Available at: <https://www.fullstackfoundations.com/blog/oop-principles> [Accessed: 13th October 2025].

van Rossum, G. (2001) *PEP 8 – Style Guide for Python Code*, Python. Available at: <https://peps.python.org/pep-0008> [Accessed: 13th October 2025].

Zeil, S. J. (2013) *The Object-Oriented* Philosophy, Department of Computer Science, Old Dominion University. Available at: [https://www.cs.odu.edu/~zeil/cs330/f18/Public/themes](https://www.cs.odu.edu/~zeil/cs330/f18/Public/themes/) [Accessed: 12th October 2025].