Software design
Team project – Deliverable 2

Team number: 12

Team members:

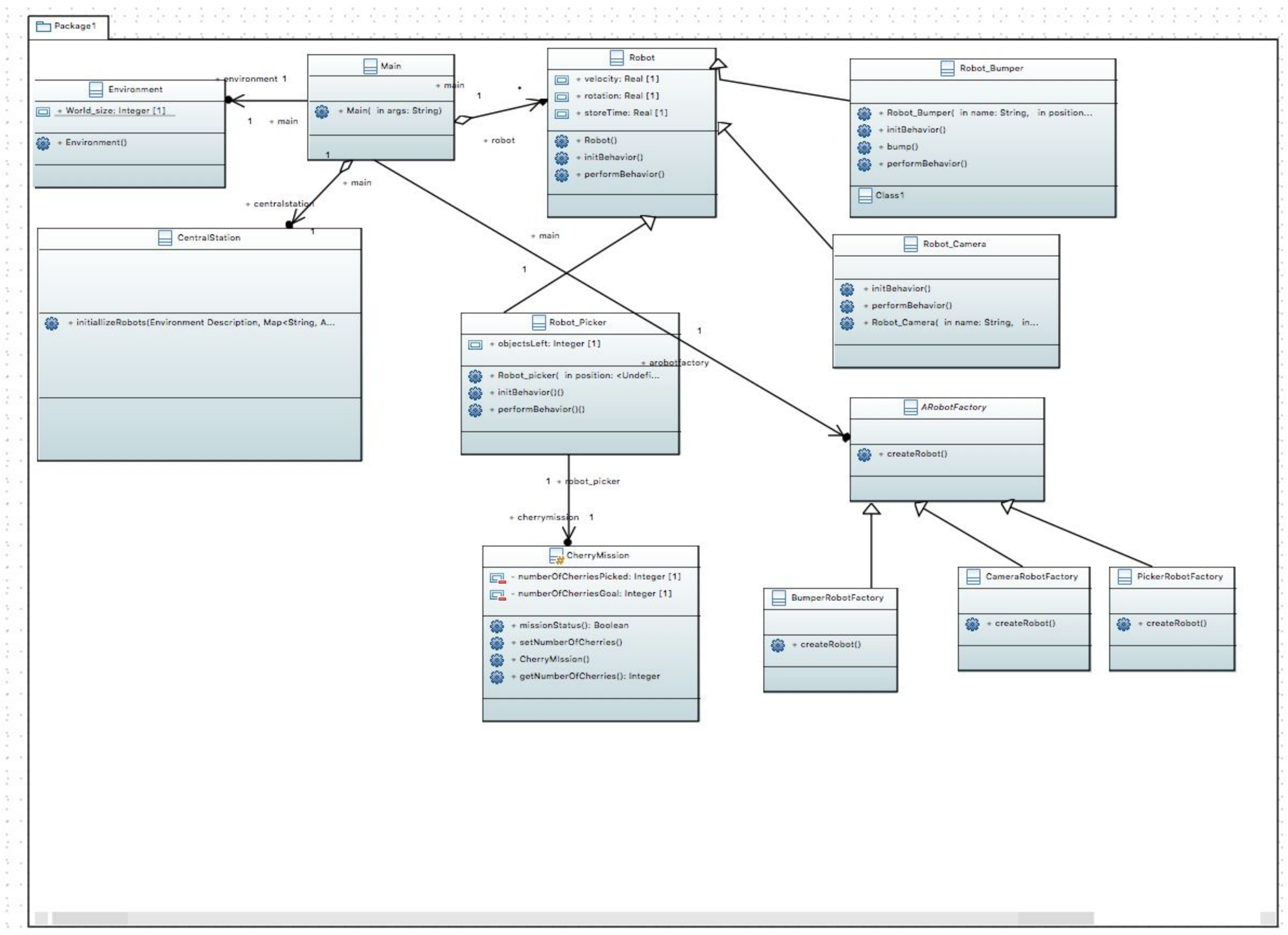
Name	Student Nr.	E-mail
Klaas Schoenmaker	2599604	klaasscho@gmail.com
Sara Real	2650919	saresa.3@outlook.es
Victoria Zimmermann	2607488	tzimms7@gmail.com
Mark Wiering	2621327	markwiering@gmail.com
Syed Zain Ul Abbas Moosvi	2548636	szain07@gmail.com
Joshua Caumont	2595695	joshua.caumont@gmail.com

Table of Contents

Table of Contents……………………………………………..2
1. Class diagram……………………………………………....3
2. Applied design patterns……………………………….... 8
3. Object diagram……………………………………………..10
4. Code generation remarks…………………………………..11
5. Implementation remarks…………………………………...11

Class diagram

Author(s): Zain, Sara, Joshua, Victoria, Klaas



Main

The **Main** class is responsible of creating instances for **CentralStation**, **EnvironmentDescription** and instances for every robot factory type**,** as well as updating the frames of the simulation.

Attributes :

Main does not contain any attributes.

Operation :

Main contains only the main method that creates the previously mentioned instances.

Association:

- ➔ Invokes **Environment** to create the environment and has access to its information
- ➔ Invokes **Centralstation** and has access to its functions and information

CentralStation

CentralStation creates all the robots using the robot factories and adds them to the environment

Attribute:

No attributes

Operations:

`initializeRobots()` - uses the member function .createRobot() of the factory objects to create the robots then adds them to the environment

Association:

- ➔ In association with the factory classes

CherryMission

CherryMission represents the mission assigned to the cherry picking robot. This robot has to pick cherries layed out randomly on the environment. **CherryMission** is called by the class **Robot_picker**. It implements attributes and methods useful for this mission.

Attributes:

`numberOfCherriesPicked:int`	integer storing the total number of cherries picked so far
`numberOfCherriesGoal:int`	integer storing the set number of cherries to be picked

Operation

`setNumberOfCherries()`	- sets numberOfCherriesGoal to a specified integer value.
`getNumberOfCherries()`	- gets the integer numberOfCherriesPicked value.
`missionStatus()`	- returns false if the goal is not set returns false if numberOfCherries Picked != numberOfCherriesGoal returns true if numberOfCherries Picked == numberOfCherriesGoal
`cherryPicked()`	- increments numberOfCherries Picked by 1.

Association:

- ➔ Gets invoked in the constructor of the and is a complement of CentralStation.
- ➔ *Picker* the instance of **Robot_picker** has access to the information in this class in order to determine how many cherries are left and when to terminate the object gathering process.

Robot_bumper

Robot_bumper is a class representing our bumper robot. It is an extension of the class **Robot**. The **Robot_bumper** class is supposed to move freely in the map, without any specific direction. He is equipped with a bumper sensor and a light. Everytime he encounters a wall or obstacle, he must communicate it and change the light mode (blinking or fixed light).

Attribute:
No attributes

Operation :

`initbehavior()`	- indicates the initial behavior
`performbehavior()`	- manages the movement of the robot and calls another method when it collides with an obstacle.
`bump()`	- manages the collision of the robot by sending a message and changing the lamp state.

Association

- ➔ **Robot_bumper** is in inheritance relation with **Robot** as it is a child of that Abstract class.

Robot_camera

Robot_camera is a class representing the camera robot. It is an extension of the class **Robot.**
The **Robot_camera** class's aim is to take pictures of the environment our robot is in. It has a camera sensor placed in the front part of the robot.

Attribute:
No attributes

Operation:

`initbehavior()`	- indicates the initial behavior
`performbehavior()`	- manages the movement and main behavior (taking pictures) of the robot.

Association:

- ➔ **Robot_camera** is in inheritance relation with **Robot** as it is a child of that Abstract class.
  .

Robot_picker

Robot_picker is the class of the robot responsible of picking up the cherries in our environment. Like all of the robot classes above, he is an extension of our abstract class **Robot.** He moves around and every time he encounters a cherry, he picks it. When he has taken all of them, he stops.
Attribute:

`objectsLeft:int`	integer representing the cherries left

Operation:

`initbehavior()`	- sets the initial behavior of the class
`performbehavior()`	- controls the movement of the robot and the cherry picking action.
`Initialize mission()`	- Set

Association:

- ➔ **Robot_picker** is in inheritance relation with **Robot** as it is a child of that Abstract class.
- ➔ **Robot_picker** is in association with the **CherryMission** instance *mission* and has access to the information in *mission.*

Environment

The **Environment** class represents the environment of our system.It is the class in which we create the Walls, Boxes and Arches. **Environment** class extends **EnvironmentDescription** which is the class including in Simbad Framework.

Attribute :
`world_size:int` integer indicating the size of our map

Operation :
`Environment()` - creates every element related to the map, the walls, boxes, arches…

Association :

- ➔ **Environment** has an association with the **Main** class.

Robot

The **Robot** class is an abstract superclass that is inherited by the other robot-related classes. The **Robot** class has a constructor, an attribute and two operations that are common to all of our robots. The most 'extensive' operation is the performBehavior() that defines the movement and rotation of the robots (in case they collapse).

Attribute :
`velocity:int` integer indicating the velocity of the robots

Operation :

`initBehavior()`	- indicates the initial behavior
`performBehavior()`	- indicates common behavior of all robots (movement and obstacle avoidance)

Association :

- ➔ **Robot_cam, Robot_bumper, Robot_picker** inherit from **Robo**t and are in a generalization with it.

ARobotFactory

The **ARobotFactory** is a class that acts as a factory method and creates a simple robot.

Attribute :
No attributes

Operation :
`createRobot(name, environment)` - empty operation (to be implemented by the other factories)

Association :

- ➔ Superclass of **CameraRobotFactory**, **BumperRobotFactory**, **PickerRobotFactory** which are in a generalization with it.

CameraRobotFactory

The **CameraRobotFactory** class is a factory method for our **Robot_camera** class. It simply implements the **ARobotFactory** in order to create our **Robot_camera**.

Attribute:
No attributes

Operation:
`createRobot(name, environment)` - creates a camera robot by implementing the method from **ARobotFactory**

Association:
- ➔ **CameraRobotFactory** implements the **ARobotFactory**.

BumperRobotFactory

BumperRobotFactory contains the factory method for **Robot_bumper**. It implements **ARobotFactory** in order to create **Robot_bumper**.

Attribute:
No attributes

Operation:
`createRobot(name, environment)` - creates a bumper robot by implementing the method from **ARobotFactory**

Association:
- ➔ **BumperRobotFactory** implements the **ARobotFactory**.

PickerRobotFactory

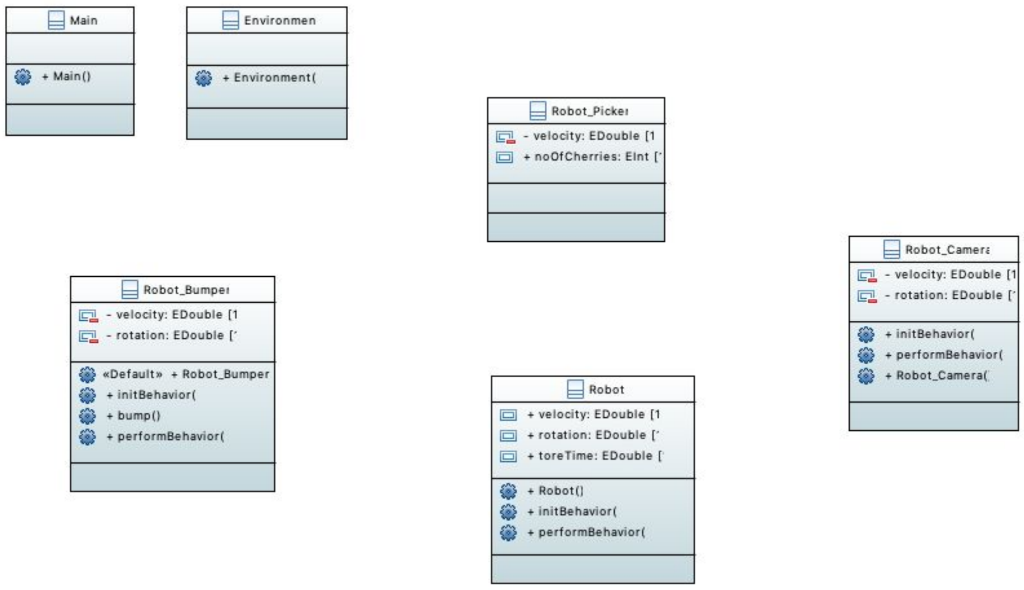
PickerRobotFactory contains the factory method for **Robot_picker.** It uses the method defined in **ARobotFactory** and implements it.

Attribute:
No attributes

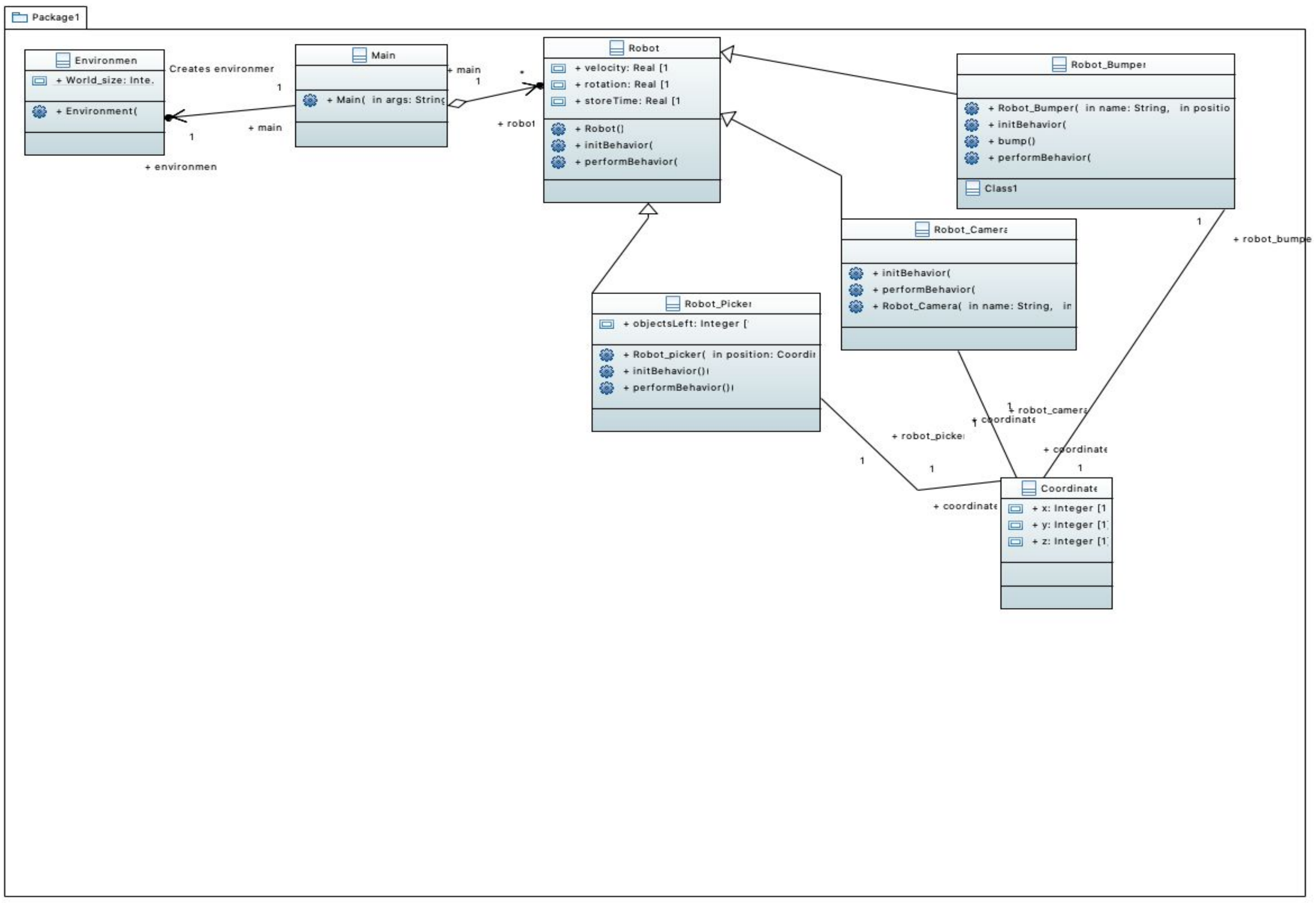
Operation:
`createRobot(name, environment)` - creates a cherry picker robot by implementing the method from **ARobotFactory**

Association:
- ➔ **PickerRobotFactory** implements the **ARobotFactory**.

Old Diagrams



This is our first attempt at the class diagram just showing the different classes we had in our implementation before extending it their attributes and their operations before but without their associations.



After a bit we came up with the associations between the classes but still struggled to find ways to find effective use of the design patterns.

Applied design patterns

Author(s): Joshua, Sara,

ID	DP1
Design pattern	Singleton
Problem	We need a central unit that creates all the robots such that it is easy to view the informations stored by the robots. And to easily assigns a mission to the cherry picking robots.
Solution	Making a Singleton **CentralStation** class that only has one instance in the system. Which creates robots and can initialize a mission for the *picker* robot
Intended use	**CentralStation** calls functions to create rovers -createCamera() -createBumper() -createPicker() **CentralStation** initializes a mission with -initializeMisssion() And passes the **cherryMission** instance to picker robots.
Constraints	Robots and **CherryMission** must not have references to **CentralStation** **CentralStation** must have references to Robots and **CherryMission**

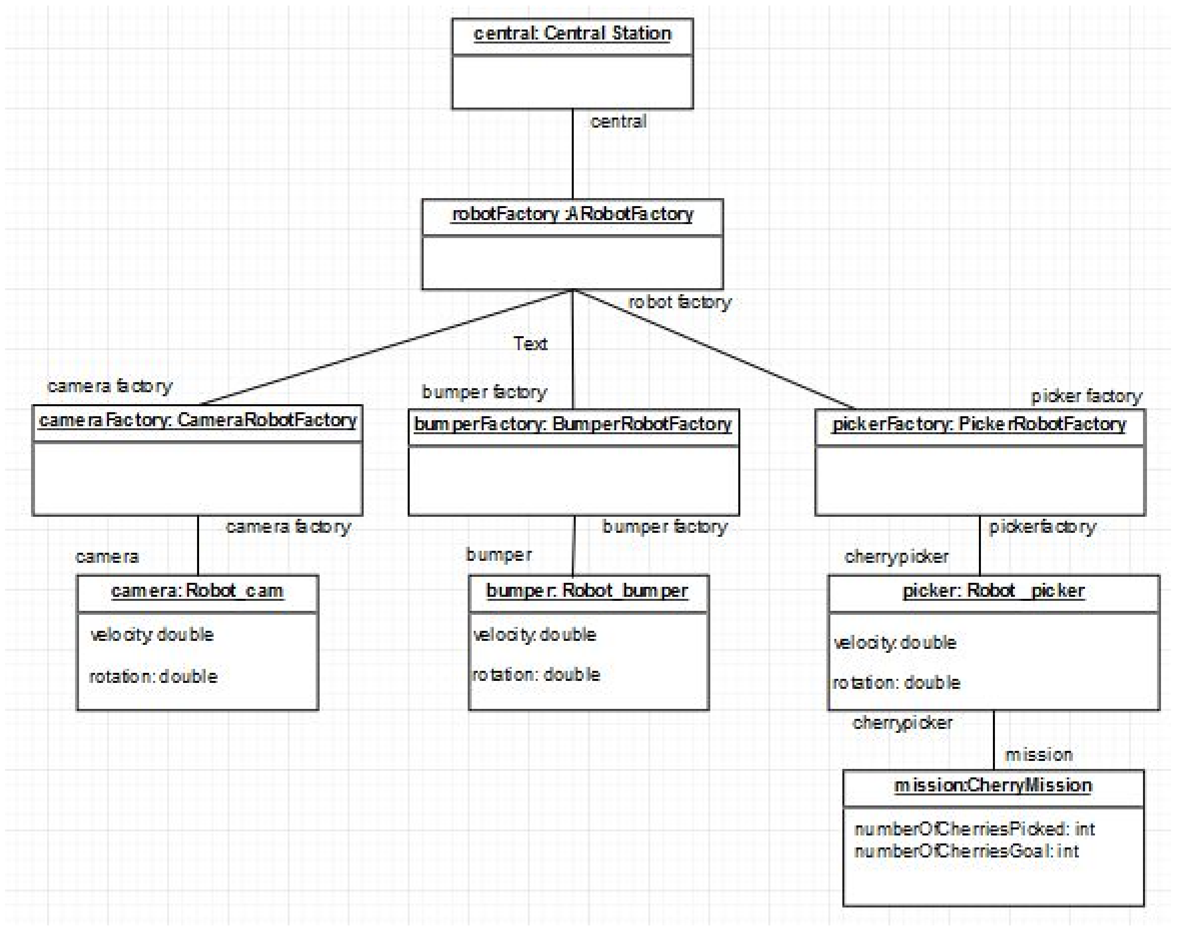
ID	DP2
Design pattern	Singleton
Problem	We wanted to define the mission and the elements within it in only one single centered class in order to avoid the problem of having two simultaneous mission and the system terminating when the wrong mission is complete.
Solution	Construct a class responsible of the creation of the different elements needed for the mission.
Intended use	When the **CherryMission** class is called from the **CentralStation** class, it creates important objects for the mission as well as some methods that help the robot.

Constraints	The cherries cannot be created there (programming constraint we couldn’t solve).

ID	DP3
Design pattern	Factory Method
Problem	There are different objects with some common characteristics
Solution	Build a **ARobotFactory** class that lays out the creation of the different robots and three different factory classes: **CameraRobotFactory**, **BumperRobotFactory** and **PickerRobotFactory** that implement the creation method of every robot.
Intended use	**ARobotFactory** uniforms creation methods used by all of the other ‘factories’.
Constraints	We could have had one single unique factory for the robots as well as common methods/attributes on the factory.

Object diagram

Author(s): Joshua, Sara, Victoria



Central Station Objects
Central is an object of the **CentralStation** class. It is associated with the robot factory object as it uses it in order to access and create the robots. Central is responsible of the initialization of the robots objects.

Factory Objects
robotFactory is an object of **ARobotFactory** class, it is associated with three other factory objects as they each implement code above in order to create the specific robot objects.

Robot objects
We have here three different robot objects: camera, bumper and cherry picker. All of them have different factory methods described above and each one of them have different objectives. Camera takes pictures of the environment, cherry picker picks up objects (cherries) and bumper goes through the environment signalising when he collides.
Cherry picker is also in association with the mission method as it calls it.

Mission object
The mission object is called by the cherry picker object. It represents the mission ‘CherryMission’ by laying out some necessary elements and methods of the Cherry Mission like the number of Cherries that were picked or the number of Cherries left.

Code generation remarks

Author(s): Zain

To generate the code we simply right click on package in the model explorer > Designer > Generate Java code and than follow the JDT dialogs that let us create a new JDT project ,where code is generated.

The code generated by Papyrus software had many comments we had to remove them. Papyrus also sometime acted weird when we trying to manipulate the properties of the class or attribute.

It is impossible to use various data types that are normally built in within the Java runtime environments. For those reasons, some of the types specified in the Class diagrams are not the ones we are actually using in the implementation since we had to use substitutes.

Implementation remarks

Author(s): Mark Wiering

On this deliverable we focused more on the class diagram and code generation rather than improving the implementation and the functioning of the robots. Due to this the implementation is still limited. For example we haven’t implemented a specific direction for the robots to move so the cherry picker robot can just move straight to a cherry. We also encountered some problems with papyrus and our git repository making us lose considerable amounts of time on issues not related to our system or our system model. With hindsight a better approach would have been to have a clearer idea of what our design patterns should be before implementing and perhaps a better way to implement missions for our system is one that could handle multiple missions one after the other.