Hanoi University of Science and Technology

The School of Information and Communication Technology

**A red text on a black background

Description automatically generated**

**IT3100E-141177**

**Object-oriented Programming**

**Mini-Project**

**Interactive simulation of composition of forces**

**[Group 12]**

Tran Duc Tuan Kien 20214908

Pham Tuan Kiet 20214909

Tran Le My Linh 20210535

Bui Khanh Linh 20214910

Ha Noi, July 2023

Contents

[1. Introduction 3](#_Toc140343600)

[2. Assignment of member 4](#_Toc140343601)

[3. Mini project description 5](#_Toc140343602)

[3.1. Project overview 5](#_Toc140343603)

[3.2. Project requirement 5](#_Toc140343604)

[3.3. Use case diagram and explanation 6](#_Toc140343605)

[3.3.1 Use case diagram 6](#_Toc140343606)

[3.3.2 Explanation 6](#_Toc140343607)

[4. Design 8](#_Toc140343608)

[4.1. General class diagram 8](#_Toc140343609)

[4.2. Detailed class diagram of each package 9](#_Toc140343610)

[4.2.1 Controller 9](#_Toc140343611)

[4.2.2 Model 16](#_Toc140343612)

[4.3. View 21](#_Toc140343613)

[Appendix A 23](#_Toc140343614)

[Table 23](#_Toc140343615)

[Figure 23](#_Toc140343616)

[5. Reference 24](#_Toc140343617)

# Introduction

In this project, we will explore Newton's laws of motion by examining the behavior of two objects: the cube and the cylinder. These objects have distinct properties and respond differently to external forces.

The cube is a three-dimensional object with equal side lengths. When a force is applied to a cube, it affects its center of mass. The direction and magnitude of the force determine the cube's resulting motion. Friction also plays a role in its movement, with the coefficient of friction determining the resistance to sliding.

The cylinder, on the other hand, is characterized by a circular cross-section and height. When a force is applied to a cylinder, it affects its center of mass. Similar to the cube, the direction and magnitude of the force determine the resulting motion. Friction between the cylinder and the surface affects its sliding motion.

To implement this simulation, we will employ object-oriented programming (OOP) techniques. OOP allows us to encapsulate the properties and behaviors of the cube and cylinder within separate classes, ensuring their internal workings are hidden and can only be accessed through well-defined interfaces. We will also utilize abstraction to extract common characteristics and behaviors into base classes, making code reuse and shared functionality implementation easier.

Polymorphism will enable us to handle different object types (cube and cylinder) through a common interface, facilitating uniform interaction regardless of their specific implementations. Inheritance will be used to create derived classes for the cube and cylinder, inheriting shared characteristics and behaviors from base classes, organizing code, and allowing object-specific customization.

By employing OOP techniques, we can develop a modular and adaptable simulation that accurately represents the interaction between forces and objects according to Newton's laws of motion. This project aims to deepen our understanding of physics principles and showcase the practical applications of object-oriented programming.

Get ready to dive into the fascinating world of motion and object interaction!

# Assignment of member

In this section, we will announce the work each team member has done by listing the tasks, classes, and methods in which they are mainly involved.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tran Duc Tuan Kien**  20214908 | **Pham Tuan Kiet**  20214909 | **Tran Le My Linh**  20210535 | **Bui Khanh Linh**  20214910 |
| Report  Slide  Controller Package  View  Controller Diagram  Logic + Prototype | Report  Slide  Controller Package  View  Use Case diagram  Video Demo | Report  Slide  Model Package  Model diagram  Test Model | Report  Slide  Controller Package  View  General diagram |
| ForceSimulation.java  SliderController.java  BackgroundController.java  slider.fxml  background.fxml | SceneController.java  CheckboxController  scene.fxml  checkbox.fxml | MainObject.java  CubeBox.java  Cylinder.java  Surface.java  RotatingObject.java  Model.java | InfoController.java  ForceController.java  RoadController.java  info.fxml  force.fxml  road.fxml |

Table 2‑1. Assignment of member

# Mini project description

## Project overview

* Create a simple interactive simulation application to demonstrate Newton's laws of motion.
* Utilize Version Control (specifically GitHub) for effective collaboration and sharing among team members.
* Create use-case and class diagrams to guide the development process.
* Apply object-oriented programming concepts such as Inheritance, Polymorphism, Abstraction and Encapsulation.
* Clearly explain the project ideas and reasons behind their selection in the report and presentation.

## Project requirement

As we mentioned before, we need to create an interactive simulation application to demonstrate Newton's laws of motion with specific requirements:

* The GUI should be the same as in the reference, with objects, sky and surfaces. However, we will design the same graphical interface but will be more artistic.
* Users can control the main object, the surface, and an actor who applies a horizontal force on the main object's center of mass to observe its motion.
* Start the application by setting up the main object (cube or cylinder) on the surface and specifying relevant parameters (side length/mass for a cube, radius/mass for a cylinder).
* During simulation, users can control the actor's force by adjusting its length and direction.
* Users can modify static and kinetic friction coefficients of the surface.
* Display statistics related to forces, mass, velocity, acceleration, and position of the main object.
* Allow users to pause, continue, and reset the simulation.
* Recalculate the main object's statistics at each time interval based on the provided formula for physical force impact.
* Users can change the main object once chosen, and modifying parameters of the current object is initially allowed.
* If time permits, implement the ability to reset and choose a different main object.

## Use case diagram and explanation

### Use case diagram

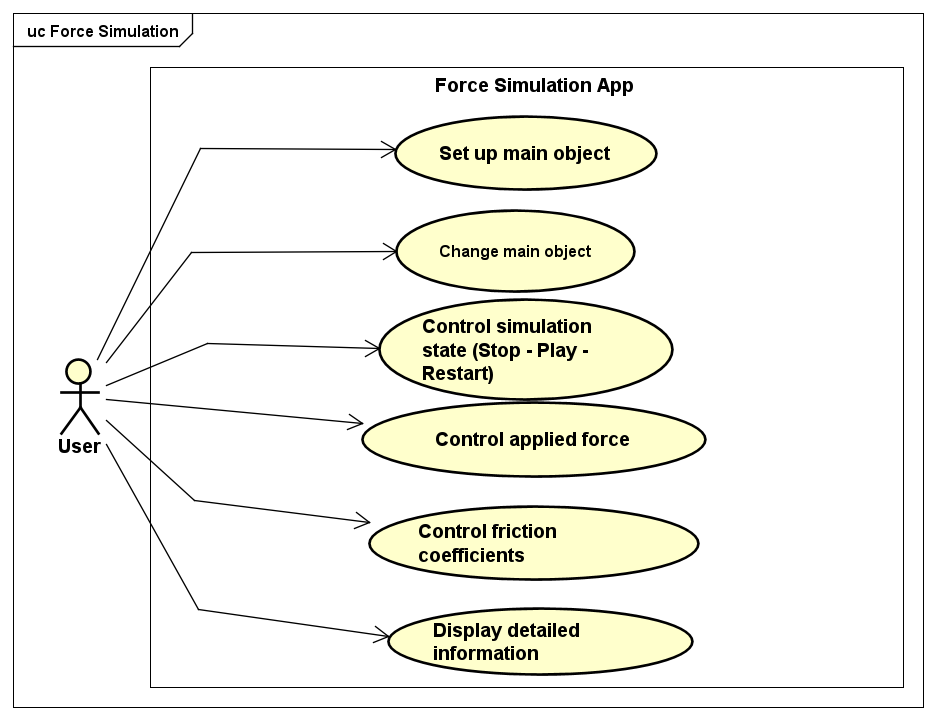


Figure 1. Use case diagram.

### Explanation

#### Set up main object

* User chooses the type of the object (cylinder or cube) by clicking on that object, and then passes the value for object’s mass and size by filling in the corresponding blank.

#### Change main object

#### User can change click another object on the bar and reset the object.

#### Control simulation state (Stop – Play – Restart)

#### Users can modify the simulation state by interacting with the control buttons. The buttons available for controlling the simulation state include "Stop", "Play", and "Restart".

* "Play" button: By pressing the "Play" button, the user can start or resume the simulation. This button initiates the simulation process or continues it if it was previously paused.
* "Stop" Button: The "Stop" button allows the user to pause the simulation. When the simulation is running, pressing this button will temporarily halt the progress and hold the current state.
* "Restart" Button: The "Restart" button is used to restart the project or simulation. When pressed, it resets the simulation to its initial state, clearing any previous progress or changes made during the simulation.

#### Control applied force

* Users can modify the applied force by adjusting the slider.
* The program will update the applied force on the object, resulting in changes to the net force, acceleration, and vector width.
* Users will see the object move faster or slower, with the force vector width reflecting the changes. Labels for acceleration and force values will be updated.

#### Control friction coefficients

* Just as with controlling the applied force, the user can adjust the coefficients of friction by adjusting the slider.
* After the coefficients of friction have been modified, the program will update the statistics, such as changing the friction force and adjusting the width of the vector representing it.
* The user will see that the object moves faster or slower, and the values displayed on the labels representing acceleration and force will change accordingly based on the current coefficients of friction.

#### Display detailed information

* Users have the option to display detailed information about the forces and the object by selecting checkboxes. The checkboxes correspond to specific label attributes.
* If a checkbox is selected, the program will display the values of the corresponding label attributes. This allows users to observe and track the detailed information related to the forces and the object.
* Users can choose to show or hide the detailed information by selecting or deselecting the checkboxes respectively. This provides flexibility in displaying the desired level of detail in the simulation.

# Design

## General class diagram

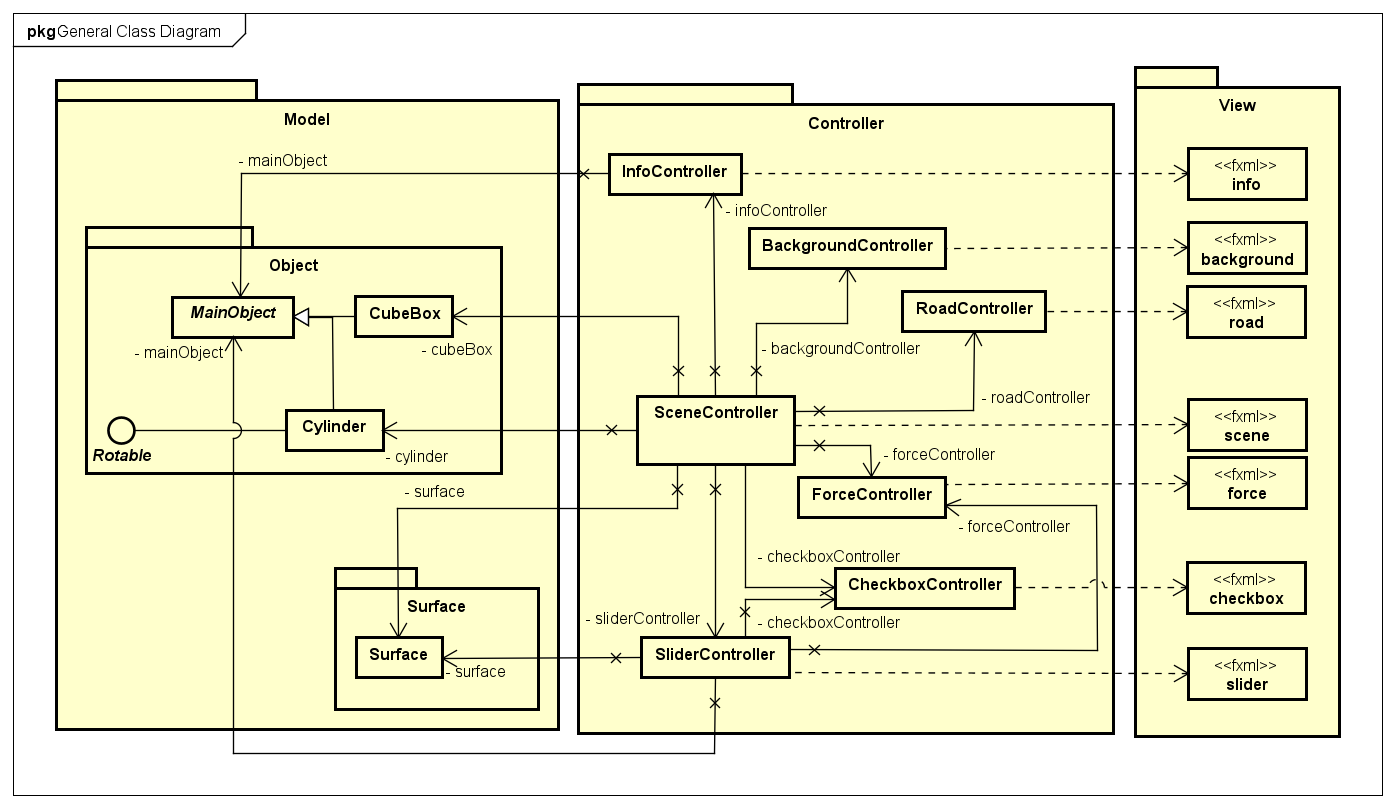


Figure 2. General class diagram.

|  |  |  |
| --- | --- | --- |
| **Package Name** | **Class Name** | **Description** |
| dsai.forcesimulation  .Model.Object | MainObject | An abstract class representing a main object (Cube/Cylinder) | |
| CubeBox | A concrete class representing a cube-shaped object. | |
| Cylinder | A concrete class representing a cylinder-shaped object. | |
| dsai.forcesimulation  .Model.Surface | Surface | A class representing the surface on which objects interact. |
| dsai.forcesimulation  .Controller | BackgroundController.java | A class controls background animation | |
| CheckboxController.java | A class controls appearance of information. | |
| ForceController.java | A concrete class controls magnitude of vector of forces | |
| InfoController.java | A class controls attributes of objects. | |
| RoadController.java | A concrete class controls movement of object (road) | |
| SceneController.java | A class controls main scene | |
| SliderController.java | A class controls slider corresponding to the applied force | |

Table 4‑1. General class diagram information

## Detailed class diagram of each package

### Controller

A screenshot of a computer

Description automatically generated

Figure 3. Controller diagram detail.

#### Road Controller

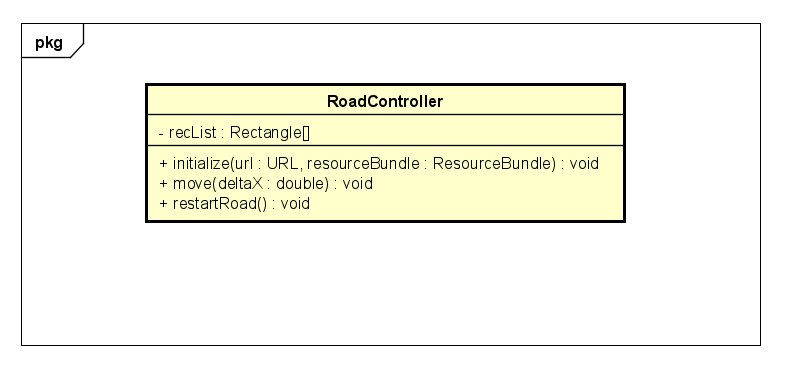


Figure 4. RoadController diagram detail.

This class controls the movement of the road creating the feeling that the object is moving.

* Attributes:

recList: this array contains Rectangle from fxml file.

* Methods:

initialize(): initializes the *recList* array with the values of the 12 Rectangle objects defined in the FXML file.

move(double deltaX): takes a double value as an argument and moves each rectangle in the *recList* array by that amount along the x-axis. If a rectangle moves beyond the left or right limits of the screen, its position is reset to the opposite side of the screen.

restartRoad(): resets the position of each rectangle in the *recList* array to its initial position.

#### Background Controller

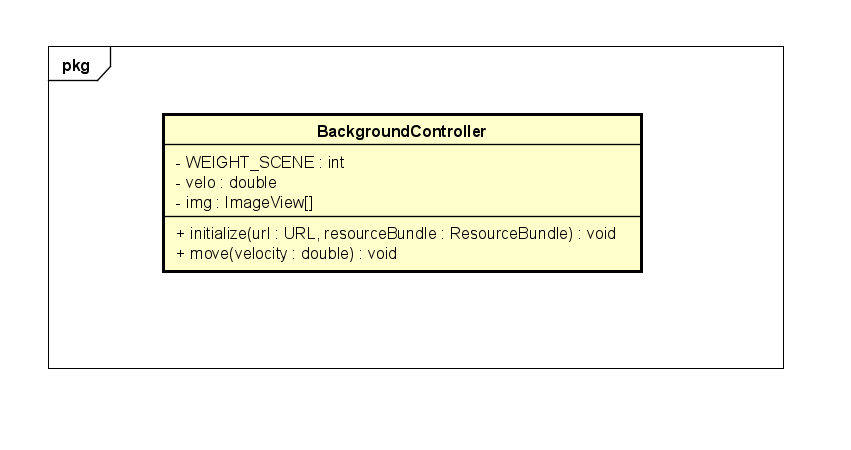


Figure 5. BackgroundController diagram detail.

This class is a utility for our model – which controller the movement of background.

* Attributes:

velo: this attribute contains velocity of background.

img: this attribute is used to contain array of image of background.

*WEIGHT\_SCENE*: this is the weight of scene.

* Methods:

initialize(): this method is called to initialize a controller after its root element has been completely processed.

move(): this method is used to move the image of background.

#### Checkbox Controller

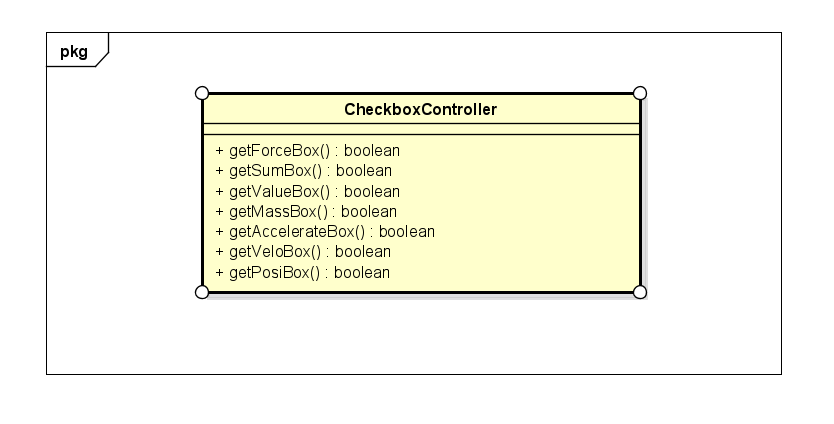


Figure 6. CheckboxController diagram detail.

The class encapsulates the state of the checkboxes within the class and provides public methods to access that state. The class provides a set of getter methods for each of the checkboxes: *getForceBox, getSumBox, getValueBox, getMassBox, getAccelerateBox, getVeloBox, getPosiBox*. It returns a boolean value indicating whether the corresponding checkbox is selected or not, to determine which checkboxes are checked by the user.

#### Info Controller

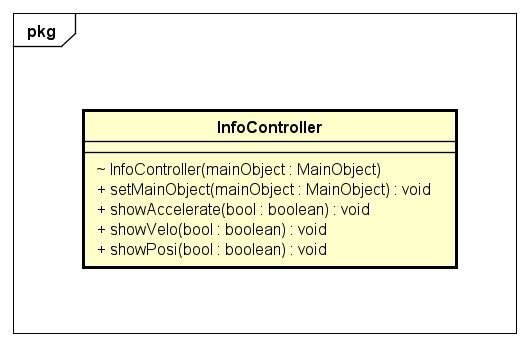


Figure 7. InfoController diagram detail.

This class displays information about a **MainObject** instance’s acceleration, velocity, and position. It has a constructor, a setter method, and three methods for displaying information about a **MainObject** instance.

* Methods:

InfoController(): takes a **MainObject** instance as an argument and sets the *mainObject* field to that value.

setMainObject(): takes a **MainObject** instance as an argument and sets the *mainObject* field to that value.

ShowAccelerate(), showVelo(), and showPosi(): take a boolean value as an argument. If the argument is true, the methods set the text of the corresponding labels to display information. If the mainObject is an instance of the Cylinder class, additional information is displayed.

#### Force Controller

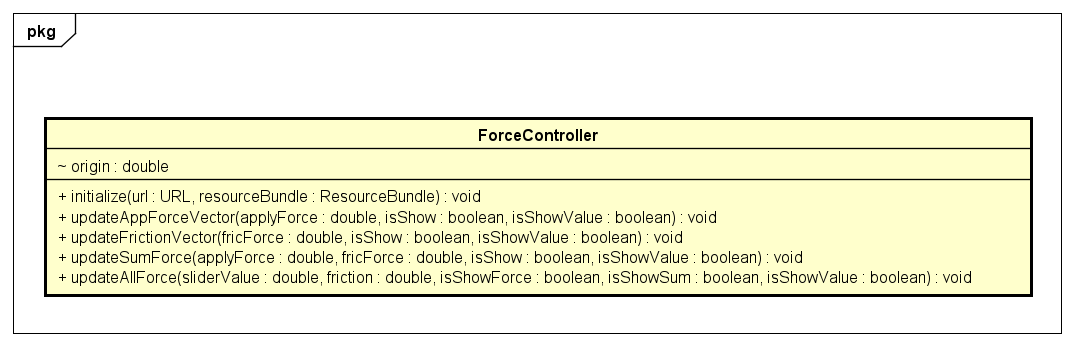


Figure 8. ForceController diagram detail.

This class updates the display of applied force, friction force, and sum force vectors. These methods update the width and position of the **ImageView** fields that represent the vectors and set the text of the corresponding **Label** fields to display their magnitudes.

* Attributes:

origin: a double value that represents the original layout x-coordinate of the *negaAppForce* **ImageView** within its parent’s coordinate system. This value is used in the *updateAppForceVector*, *updateFrictionVector* and *updateSumForce* methods to update the position of the negative force vectors along the x-axis based on their original position and the magnitude of the force.

* Methods:

initialize(): sets the value of the origin field to the layout x-coordinate of the *negaAppForce* **ImageView**.

updateAppForceVector(), updateFrictionVector(), and updateSumForce(): take a double value and two boolean values as arguments; update the display of the applied force, friction force, and sum force vectors, respectively. The double value represents the magnitude of the force, while the first boolean value determines whether or not the vector is visible. The second boolean value determines whether or not the corresponding label is visible.

updateAllForce(): takes four double values and three boolean values as arguments. It calls the updateAppForceVector, updateFrictionVector, and updateSumForce methods to update the display of all three force vectors.

#### Slider Controller

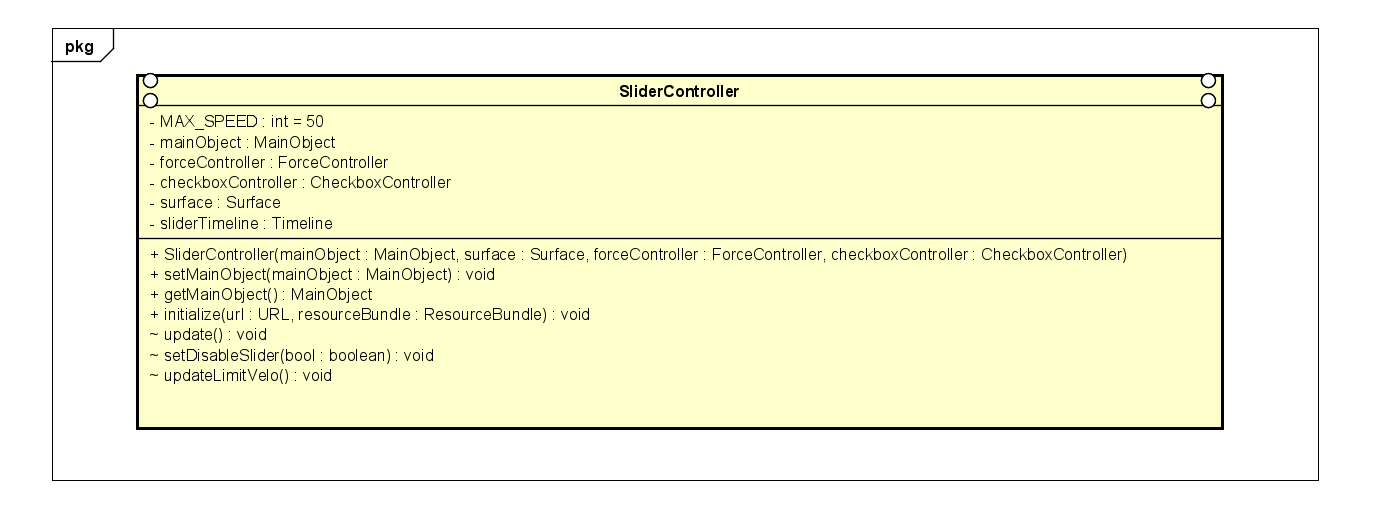


Figure 9. SliderController diagram detail.

Its class is used to control slider which is associate which the applied force by user.

* Attributes:

*MAX\_SPEED*: this attribute contains max speed that object can reach.

mainObject: this attribute is an instance of MainObject to change movement of object that is in road.

forceController: this attribute is an instance of ForceController to controller vector of force

checkboxController: this attribute is an instance of CheckboxController to decide when we show information.

surface: this attribute is an instance of Surface to get information about surface.

sliderTimeline: this attribute is an instance of Timeline to create timeline when applied force is change.

* Methods:

SliderController(mainObject: MainObject, surface: Surface, forceController: ForceController, checkboxController: CheckboxController): this is contructor of this class.

setMainObject(mainObject: MainObject): this method is used to change object running in road.

getMainObject(): this method is used to get object that is running in the road.

initialize(): this method is called to initialize a controller after its root element has been completely processed.

update(): this method is used to get applied force then update attribute of object.

setDisableSlider(): it is used to set slider be disable when there is no object in road.

updateLimitVelo(): this method is used to set applied of force equals 0 when velocity of object exceeds the max velocity.

#### Scene ControllerA screenshot of a computer program Description automatically generated

Figure 10. SceneController diagram detail.

This class serves as the central controller for the force simulation application. It arranges the interaction between different UI components, manages the state of objects in the simulation, and handles user input and actions.

* **Attributes:**

*MAX\_MASS*: A constant integer representing the maximum mass value.

*MAX\_SIDE*: A constant integer representing the maximum side length value.

originalDuration: A Duration object representing the original duration of an animation.

cubeBox: An instance of the CubeBox class representing a cube-shaped object.

cylinder: An instance of the Cylinder class representing a cylinder-shaped object.

rotation: An instance of the Rotate class representing the rotation transformation applied to a circle.

sliderController: An instance of the SliderController class responsible for controlling sliders in the UI.

infoController: An instance of the InfoController class responsible for managing information display in the UI.

checkboxController: An instance of the CheckboxController class responsible for managing checkboxes in the UI.

backgroundController: An instance of the BackgroundController class responsible for managing the background in the UI.

forceController: An instance of the ForceController class responsible for managing forces in the UI.

surface: An instance of the Surface class representing the surface on which objects move.

roadController: An instance of the RoadController class responsible for managing the road in the UI.

Various FXML-annotated attributes representing UI elements such as panes, buttons, shapes, sliders, and text fields.

* **Methods:**

Initialize(): This method is called when the FXML file is loaded. It initializes various components and sets up event handlers for the UI elements.

btnStopPressed(): Event handler method for the "Stop" button. It stops or plays the animation timeline based on its current state.

btnRestartPressed(): Event handler method for the "Restart" button. It resets the simulation and restarts the animation timeline.

setBox: Event handler method for setting the properties of a cube-shaped object. It prompts the user for input using an alert dialog.

setCylinder: Event handler method for setting the properties of a cylinder-shaped object. It prompts the user for input using an alert dialog.

cylinderInput: Helper method for handling user input when setting cylinder properties.

cubicBoxInput: Helper method for handling user input when setting cube box properties.

loadRoadPane: Helper method for loading the road pane and its associated controller from an FXML file.

loadVectorPane: Helper method for loading the vector pane and its associated controller from an FXML file.

loadCheckBoxPane: Helper method for loading the checkbox pane and its associated controller from an FXML file.

loadSliderPane: Helper method for loading the slider pane and its associated controller from an FXML file.

loadInfoPane: Helper method for loading the info pane and its associated controller from an FXML file.

loadBackgroundPane: Helper method for loading the background pane and its associated controller from an FXML file.

### Model

A diagram of a computer

Description automatically generated

Figure 11. Model diagram simple.

* **Encapsulation**

The code uses encapsulation to protect an object's properties. It achieves this by using getter and setter methods to access and update the private properties. By encapsulating these properties, the code ensures controlled access and validates the data before modifying their values.

* **Inheritance**

In Java, we define a subclass using the keyword "extends", and we used it in the Object package as follows:

1. **public** **class** CubeBox **extends** MainObject{.....}
2. **public** **class** Cylinder **extends** MainObject{.....}

The superclass **`MainObject`** defines common attributes and behaviors for objects in the simulation. The subclasses **`CubeBox`** and **`Cylinder`** inherit from MainObject and extend its functionality to suit their specific characteristics. This eliminates redundancy by defining shared properties like mass, position, velocity, and acceleration in the superclass. The subclasses override methods like **calculateFriction()** to customize their behavior based on their specific characteristics. Inheritance promotes code reuse and modularity, allowing for more efficient and organized code.

* **Abstraction**

The MainObject class being declared as an abstract class demonstrates abstraction. This allows defining abstract methods that subclasses need to implement. The calculateFriction method serves as an example of an abstract method, where subclasses must provide a specific implementation. Using abstraction helps define a common interface and provides extensibility for subclasses.

1. **public** **abstract** **class** MainObject {
2. **…**
3. **public** **abstract double** calculateFriction(**double** appliedForce, **double** staticCoeffient, **double** kineticCoefficient);}

* **Polymorphism**

Polymorphism allows objects of different classes to be treated interchangeably through a shared interface. (“Polymorphism in OOPs- Logicmojo”). In the provided code, the calculateFriction method implemented in the CubeBox and Cylinder subclasses exemplifies polymorphism. Despite having the same method name, each subclass provides its own implementation, tailored to its specific characteristics and requirements. This flexibility enables objects of different classes to be used interchangeably within the context of calculateFriction, promoting code reusability and adaptability.

* **Interface**

Cylinder can implement the RotatingObject interface and must provide the complete definition for the methods in the interface. Implementing an interface allows other classes to use the methods defined in the interface without concerning themselves with the specific implementation details of each class. This enhances flexibility and the ability to extend the system.

#### Object

A screenshot of a computer

Description automatically generated

Figure 12. Object diagram detail.

| **Class name** | **Class usage** | **Attributes** | **Method** | **OOP technique** |
| --- | --- | --- | --- | --- |
| MainObject | Base abstract class for simulation objects | - side  - mass  - position  - velocity  - acceleration | - getter & setter for attributes  - resetObject()  - updateAttribute()  - normalForce()  - calculateAcceleration()  - calculateFriction() | Inheritance  Encapsulation  Abstraction  Polymorphism |
| CubeBox | Represents a cube-shaped object in the simulation | Inherits attributes from MainObject | - calculateFriction() | Inheritance  Polymorphism |
| Cylinder | Represents a cylinder-shaped object in the simulation | Inherits attributes from MainObject  - gamma  - theta  - omega | - getter & setter for attributes  - resetObject()  - calculateFriction()  - updateAttribute()  - calculateGamma() | Inheritance  Encapsulation  Polymorphism  Interface |

Table 4‑2.Object detail.

* **MainObject class:**

It is an abstract class that serves as the superclass for CubeBox and Cylinder.

It contains properties such as side, mass, position, velocity, and acceleration.

It provides methods to get and set the values of these properties.

It includes a **`resetObject`** method to reset the acceleration, velocity, and position of the main object.

The **`updateAttribute`** method calculates the acceleration, velocity, and position of the main object based on the acceleration.

It also has an abstract method **`calculateFriction`**, which is implemented in the subclasses.

It includes helper methods to calculate normal force.

A screenshot of a computer

Description automatically generated

Figure 13. CubeBox diagram detail.

* **CubeBox class:**

It extends the MainObject class and represents a cube-shaped object..

It overrides the **`calculateFriction`** method to calculate the friction based on different conditions.

A screenshot of a computer

Description automatically generated

Figure 14. Cylinder diagram detail.

* **Cylinder class:**

It extends the MainObject class and represents a cylinder-shaped object.

It introduces additional properties such as gamma, theta and omega.

It overrides the **`calculateFriction`** method to calculate the friction based on different conditions.

The **`calculateGamma()`** method calculates the angular acceleration of the cylinder.

The **`updateAttribute()`** method calculates the angular attributes of the cylinder based on the applied force, gamma and updates them.

The class implements the **`RotatingObject`** interface, which defines the necessary properties and methods related to rotational motion.

* **RotatingObject interface**

A screenshot of a computer

Description automatically generated

Figure 15. RotatingObject diagram detail.

The RotatingObject interface provides a contract for classes that represent objects with rotational motion. By implementing this interface, a class guarantees that it will provide the necessary properties and methods for handling rotational motion.

#### Surface

A computer code with black text

Description automatically generated

Figure 16. Surface diagram detail.

| **Class name** | **Class usage** | **Attributes** | **Method** | **OOP technique** |
| --- | --- | --- | --- | --- |
| Surface | Represents the surface on which objects move in the simulation | - staticCoefficient  - kineticCoefficient | - setStaticCoefficient()  - setKineticCoefficient() | Encapsulation |

* **Surface class:**

It represents the surface on which the main object moves.

It includes properties such as staticCoefficient and kineticCoefficient.

It provides methods to get and set the values of these properties.

## View

In this project, there are many components so we break them into indepent part to control easily. Each part has fxml file and controller to control it. We have 7 fxml file corresponding to 7 controller:

* background.fxml: this is anchor pane containing 2 image of background that can move when object runs.
* checkbox.fxml: this is anchor pane which contains check box. Each check box can be selected to show information about object.
* force.fxml: this file contains anchor pane. There are 6 image of vector corresponding to each force that can be stretch based on magnitude of corresponding force
* info.fxml: this is anchor pane containing labels which contain information of acceleration, velocity, …. object.
* road.fxml: this is anchor pane. There are many rectangle in this anchor pane which are blocks in the road.
* scene.fxml: this is anchor pane which is our scene. There are many nested anchor pane in it to load graphics of other components. There also two sliders to control coefficients of friction. Buttons in this anchor pane to control state of scene.
* slider.fxml: this anchor pane contain slider to control applied force .

# Appendix A

List of table or figure we insert

## Table

[Table 2‑1. Assignment of member 4](#_Toc140343471)

[Table 4‑1. General class diagram information 8](#_Toc140343472)

[Table 4‑2.Object detail. 18](#_Toc140343473)

## Figure

[Figure 1. Use case diagram. 6](#_Toc140343491)

[Figure 2. General class diagram. 8](#_Toc140343492)

[Figure 3. Controller diagram detail. 9](#_Toc140343493)

[Figure 4. RoadController diagram detail. 9](#_Toc140343494)

[Figure 5. BackgroundController diagram detail. 10](#_Toc140343495)

[Figure 6. CheckboxController diagram detail. 11](#_Toc140343496)

[Figure 7. InfoController diagram detail. 11](#_Toc140343497)

[Figure 8. ForceController diagram detail. 12](#_Toc140343498)

[Figure 9. SliderController diagram detail. 13](#_Toc140343499)

[Figure 10. SceneController diagram detail. 14](#_Toc140343500)

[Figure 11. Model diagram simple. 16](#_Toc140343501)

[Figure 12. Object diagram detail. 17](#_Toc140343502)

[Figure 13. CubeBox diagram detail. 19](#_Toc140343503)

[Figure 14. Cylinder diagram detail. 19](#_Toc140343504)

[Figure 15. RotatingObject diagram detail. 20](#_Toc140343505)

[Figure 16. Surface diagram detail. 21](#_Toc140343506)

# Reference