PhysX:
A Text-Based Physics Engine

By: Samuel, Akhil, and Rishi

Introduction:

PhysX is a java application that allows users to create virtual 3-dimensional objects and model their behavior when forces are applied (e.g., gravity, friction, push, pull, lifting, dropping).

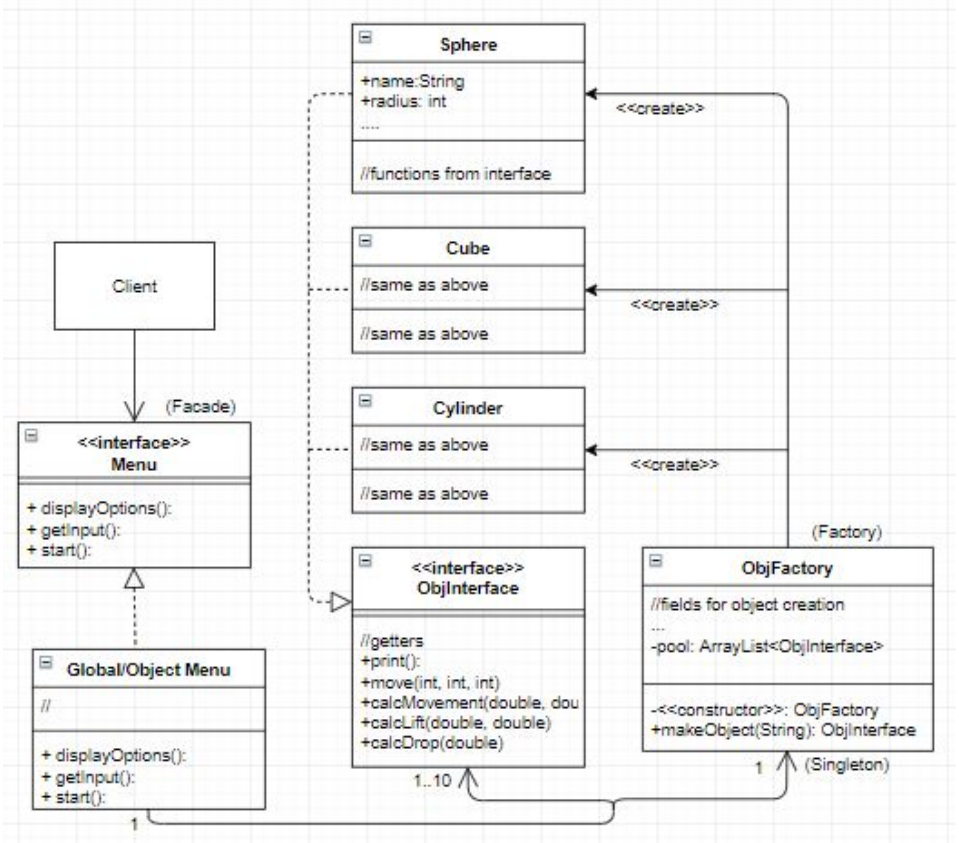
The application is text-based, displaying options and results to the user as the simulation runs.

Users are able to save their simulations and load/reuse them in the future using the .physX file extension.

System Design: Factory

Why?

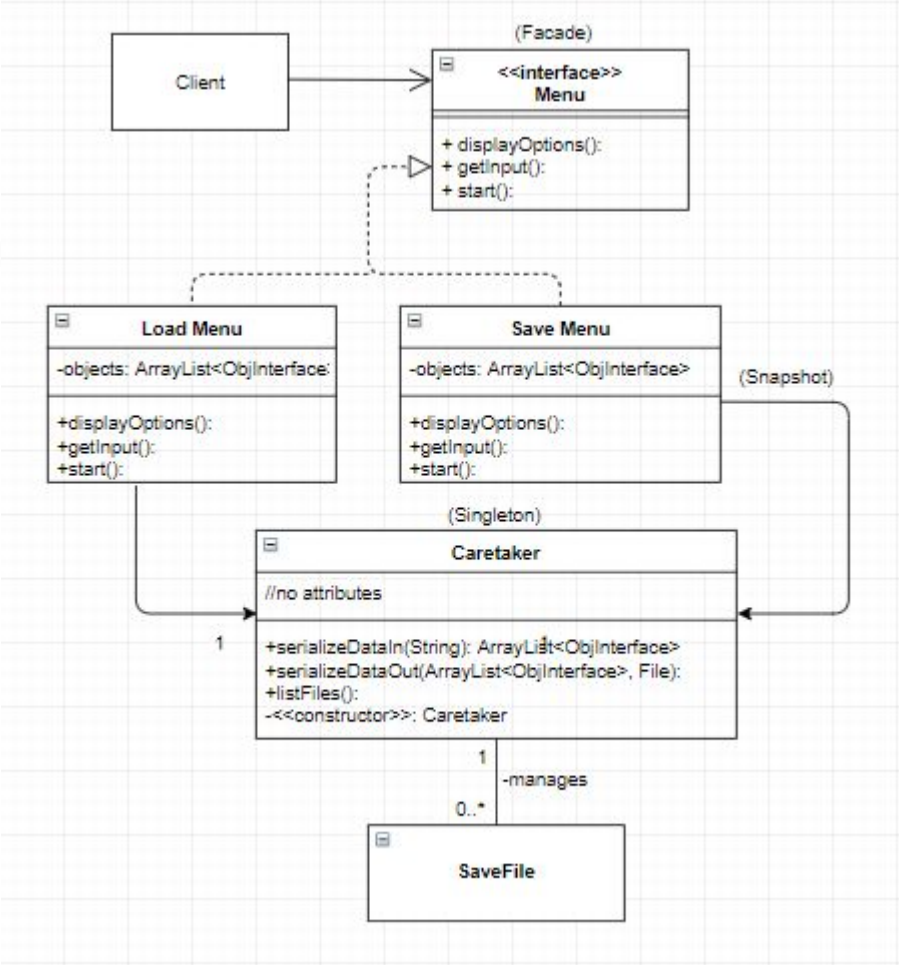
The Factory Design pattern allows us to dynamically create whole objects (Shapes) without requiring the client to understand the details of the objects' creation.



System Design: Snapshot

Why?

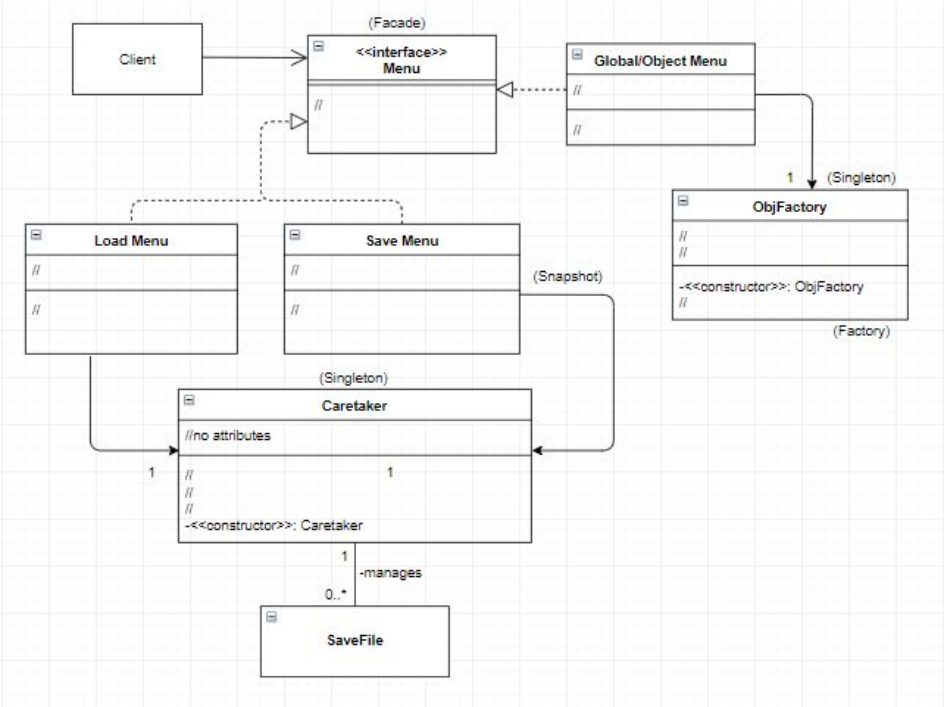
The Snapshot Design pattern allows us to store and retrieve the current state of our objects (Shapes) and facilitates the save/load system of the application.



System Design: Singleton

Why?

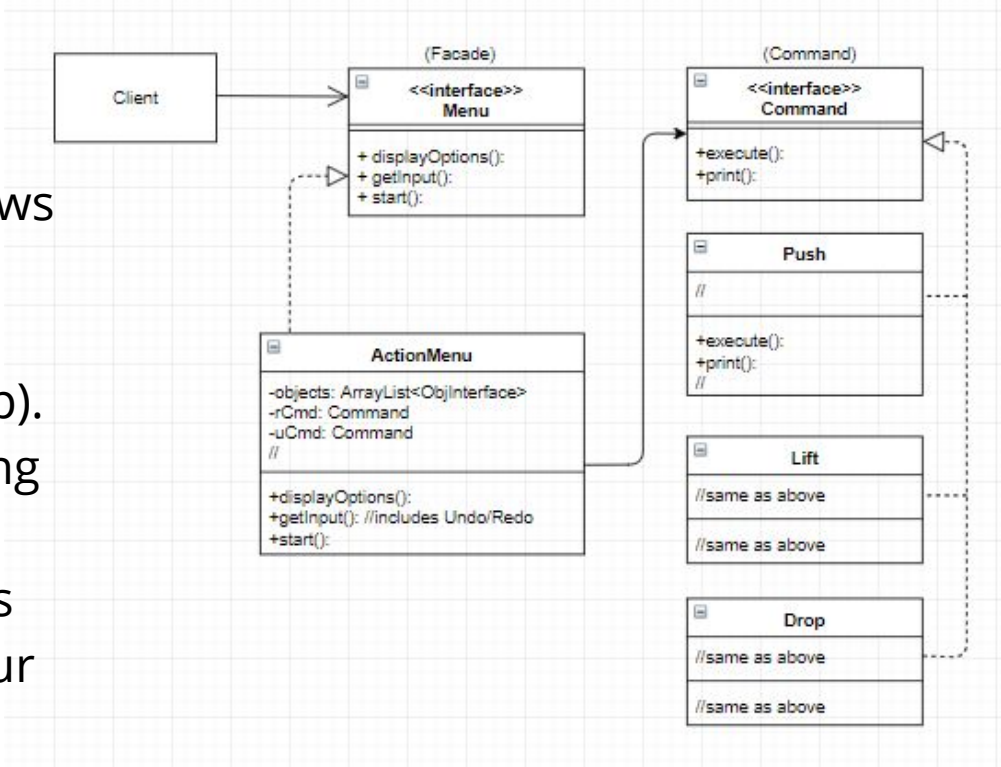
The Singleton Design pattern allows us to ensure that only one instance of our object factory and our snapshot caretaker exists. This allows the application to save memory for these expensive classes and prevents instantiation of these classes where it isn't needed.



System Design: Command

Why?

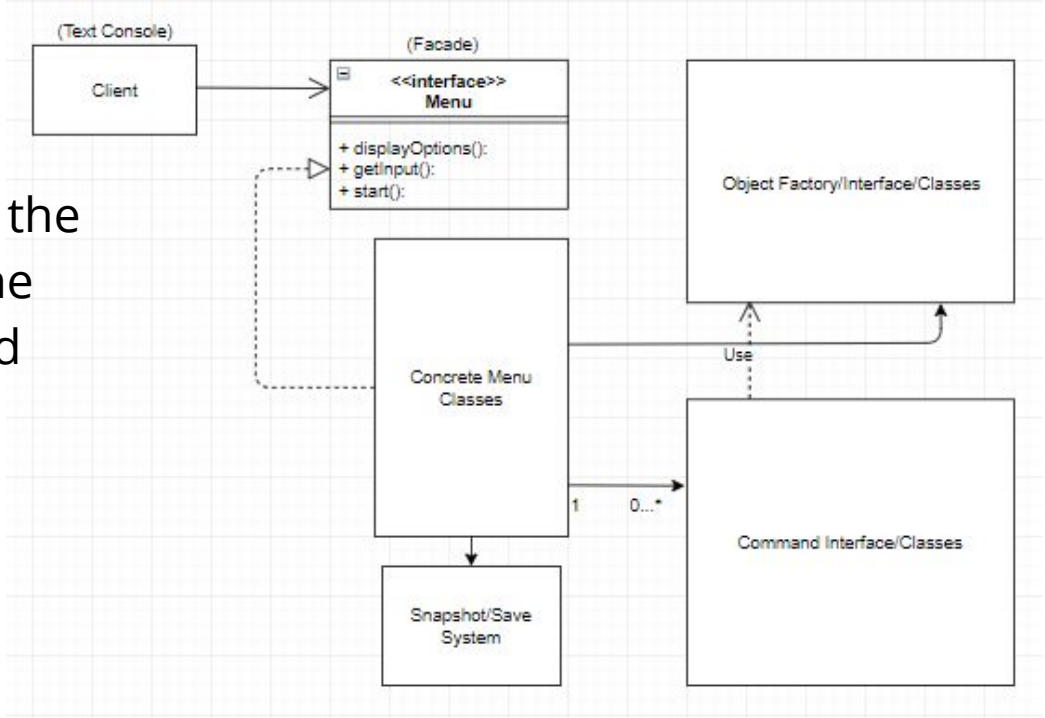
The Command Design pattern allows us to encapsulate the actions that users can select in order to apply force to objects (e.g. lift, push, drop). This encapsulation reduces coupling between the client (invoker) and shapes (receivers). It also facilitates Undo and Redo functionality for our application.



System Design: Facade

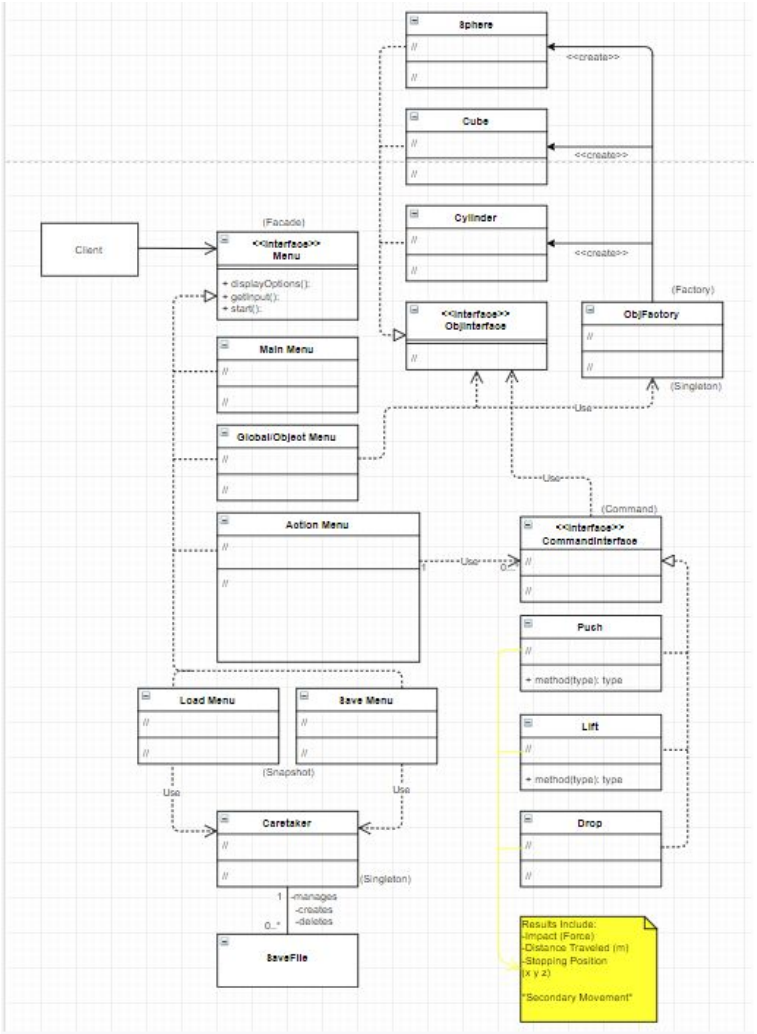
Why?

The Facade design pattern hides the complexity of our engine from the user while providing a simple and intuitive interface.



System Design: Full Design

When integrating the above design patterns, the result is:



Progress Update : System Goals

Original Goals:

- User shall be able to create an **environment** with specified parameters (e.g. friction, gravity)
- User shall be able to **create objects** with specified parameters (e.g. shape, height, radius, mass, position)
- User shall be able to apply a specified quantity of **force** to a selected object in a specified direction
- System shall **calculate and display the results** of this initial movement (e.g. distance traveled, stopping point)
- System shall calculate and display the results of any **collisions** predicated on the initial movement (e.g. what collisions take place, how much force, distance traveled)

Goals Completed:

- User shall be able to create an environment with specified parameters (e.g. friction, gravity)
- User shall be able to create objects with specified parameters (e.g. shape, height, radius, mass, position)
- User shall be able to apply a specified quantity of force to a selected object in a specified direction
- System shall calculate and display the results of this initial movement (e.g. distance traveled, stopping point)

Goals Remaining:

- System shall calculate and display the results of any collisions predicated on the initial movement (e.g. what collisions take place, how much force, distance traveled)

Progress Update : Design Pattern Changes

Original Design Patterns:

- Chain of Responsibility (user input, event signalling)
- Abstract Factory (shape creation)
- Object Pool (limit shape count)
- Command Pattern (perform operations, support undo/redo)
- Mediator Pattern (handle obj interaction from commands, simplify interface)

Final Design Patterns:

- Facade (handle user input)
- Factory (shape creation)
- Singleton (control instantiation of key+expensive classes)
- Command Pattern (perform operations, support undo/redo)
- Snapshot (Save/Load functionality)

Why we changed them:

-Chain of Responsibility added unwanted complexity to our design, which is not hierarchical in nature. In addition, Command Pattern already handles user input. Facade simplified the interface while reducing complexity. **Replaced.**

-Abstract Factory was unnecessary because we were building whole products that were similar in nature. Factory was a better choice. **Replaced.**

-Our objects are not expensive to create, so reusing them (Object Pool) added unnecessary complexity. **Removed.**

-Some key classes were costly to make and had potential file corruption issues. We added Singleton to handle this. **Added.**

-Interdependence was not an issue once facade and command patterns were implemented properly, so Mediator was unneeded. **Removed.**

-Save functionality was missing. We added Snapshot (memento) to handle this. **Added.**

System Demonstration -

Lessons Learned

What went well?

1) Version Control (Github)
2) Division of Labor (Who does what?)
3) Building Interfaces First (Working independently)

What should be improved?

1) Scheduling Meetings
2) Knowledge of Java (e.g. shallow/deep copies)
3) Limiting Scope of Project (e.g. Collisions, GUI)

Any Questions?

Design Patterns?

Extensibility?

How a certain feature works?

Does something look incorrect?