

Problem Solving Session

- The remainder of today's class will comprise the **problem solving session (PSS)**.
- Your instructor will divide you into **teams of 3 or 4 students**.
- Each team will **work together** to solve the following problems over the course of **20-30 minutes**.
 - You may work on paper, a white board, or digitally as determined by your instructor.
 - You will submit your solution by pushing it to GitHub before the end of class.
- Your instructor will go over the solution before the end of class.
- If there is any time remaining, you will begin work on your homework assignment.



Class participation is a significant part of your grade (20%). This includes in class activities and the problem solving session.

Your Course Assistants will grade your participation by verifying that you pushed your solutions before the end of the class period each day.

Problem Solving Team Members




Record the name of each of your problem solving team members here.

Do not forget to **add every team member's name!**
Your instructor (or course assistant) may or may not use this to determine whether or not you participated in the problem solving session.

Denis Kešelj
Ivano Margaretić
Gabriel Muskaj

Coming Up

SUN	MON (4/3)	TUE	WED (4/5)	THU	FRI (4/7)	SAT
	Unit 10: Concurrent Programming				Project Time	
	Unit 9 Mini-Practicum Project Part 1 Due (start of class)				Project Part 2 Team Problem-Solving Assignment 10.1 Due (start of class)	
SUN	MON (4/10)	TUE	WED (4/12)	THU	FRI (4/14)	SAT
	Midterm Exam 3		Project Time		Unit 11: Thread Cooperation	
	Units 7-9 Written (30%) Practical (70%)		Project Part 3 Team Problem-Solving - Due Monday 4/17 (start of class) Protect Part 2 Due (start of class)		Unit 10 Mini-Practicum	

Rush Hour

- **Rush Hour** is a sliding block logic game.
- The goal is to get the red vehicle out the one and only exit.
- The game begins with the vehicles gridlocked in a puzzle pattern.
- A vehicle is aligned either horizontally or vertically and can move in both directions in that orientation. That is, a vehicle aligned vertically can move only up or down. A vehicle aligned horizontally can move only left or right.
 - Vehicles cannot move sideways or diagonally.
- The red vehicle is always aligned with the exit.
- The player moves one vehicle at a time in any valid direction one space at a time.
- Play continues until the red vehicle is at the exit.
- Take a few minutes to give the game a try:
<https://www.thinkfun.com/rush-hour-online-play/>



- **Pair programming** is a technique during which two developers collaborate to solve a software problem by writing code together.
- One developer takes on the role of **the driver**.
 - Shares their screen.
 - Is actively writing code.
- The other developer(s) takes on the role of **the navigator**.
 - Watches while the driver codes.
 - Takes notes.
 - Asks questions.
 - Points out potential errors.
 - Makes suggestions for improvements.
- The driver and navigator regularly **switch roles**, e.g. every **10-20 minutes**.
 - Set a timer!
 - **Push your code!**
- For the rest of today's problem solving session, you and your team will practice pair programming with **one** team member acting as the driver and the **remaining** team members acting as the navigators.

Pair Programming

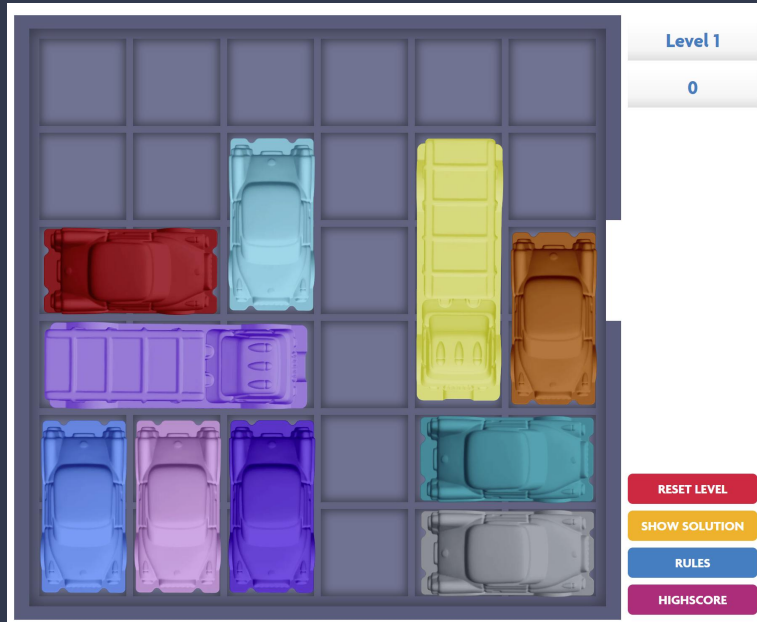
Everyone on the team should take a moment to watch [this short video on pair programming](#).

zoom



When you are the driver, you should be using Zoom or Discord to **share your screen**. Be sure to **push your code** and **switch roles** every 10-20 minutes!

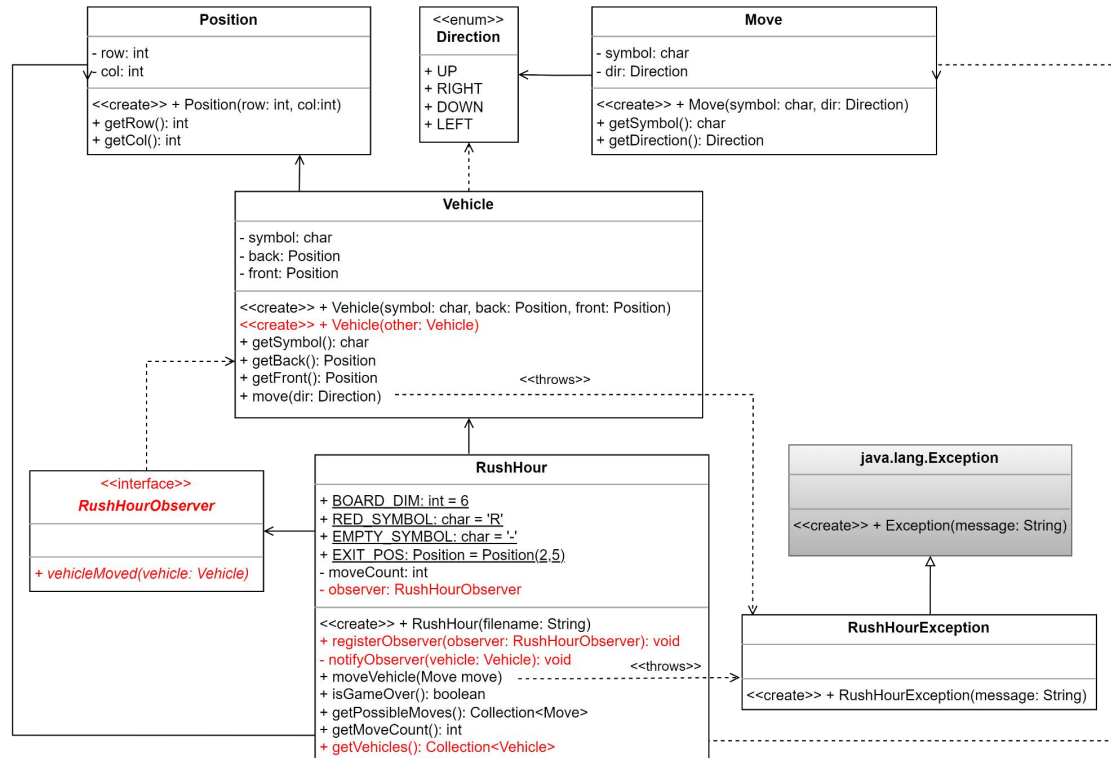
GUI Design



An image of an online version of Rush Hour from <https://www.thinkfun.com/rush-hour-online-play>. Your GUI does **not** need to visually match it and you should only implement the functionality outlined in the requirements.

- You have freedom in the look-and-feel of your GUI, but your application must
 - Display the a current **status** of the game, including
 - A New Game that has not been started.
 - Whether or not a move was valid, and if invalid, the reason why.
 - When the game has been won.
 - A hint?
 - Display and update the **number of moves** as the game progresses.
 - A 6x6 **board**. It is up to you if you want to show gridlines on the board.
 - Display all **vehicles** at their current positions on the board.
 - There must be a means to move each vehicle in its two valid directions.
 - Provide a means for the user to request a **hint**
 - How the hint is displayed to user is up to you as long as it is clearly visible.
 - The hint should be cleared after the next move.
 - Provide a means for the user to **reset** the game.
- Using paper, whiteboard, or a drawing application, sketch out your **GUI design**.
 - Indicate which **JavaFX controls** you will use for each aspect of the GUI.
 - **Upload** an image of your GUI design to your data directory of your repo.

An Enhanced Partial Design



Refer to this full UML class diagram as needed while you and your team work through the rest of this assignment.

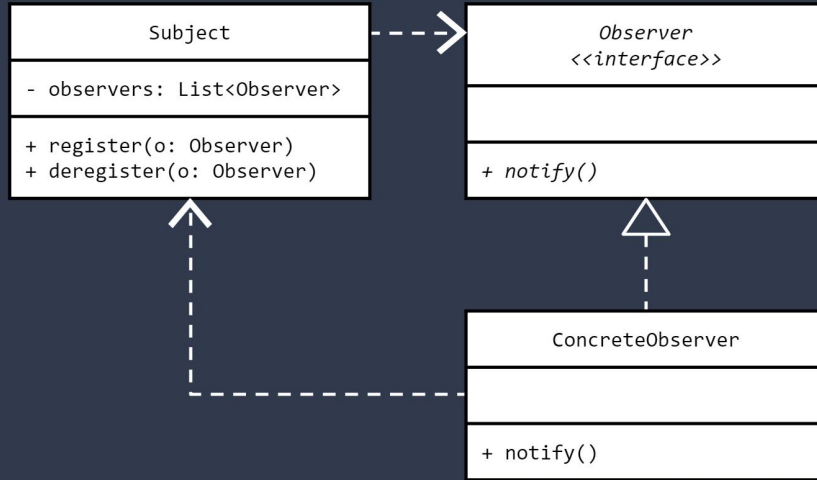
Copy Constructor

Vehicle
- symbol: char - back: Position - front: Position
<<create>> + Vehicle(symbol: char, back: Position, front: Position) <<create>> + Vehicle(other: Vehicle) + getSymbol(): char + getBack(): Position + getFront(): Position + move(dir: Direction)

A **copy constructor** creates a new object by making a **deep copy** of an object of the same class.

- Vehicle objects are reference types that experience state change in the model.
 - Therefore, it is recommended that when Vehicle objects are passed from the model to the GUI, a deep copy is made and the copy is returned to the GUI.
- To support the ability of a Vehicle object to make a copy of itself, we will need to add another constructor to the class.
- Next, you and your team should begin working on implementing the **copy constructor** method in your Vehicle implementation.
 - When called to create a new instance of the class, it should make a **deep copy** the Vehicle object that was passed in.
 - Don't forget that you will need to make deep copies of any **mutable types** or **data structures** like lists and arrays!

Review: The Observer Design Pattern

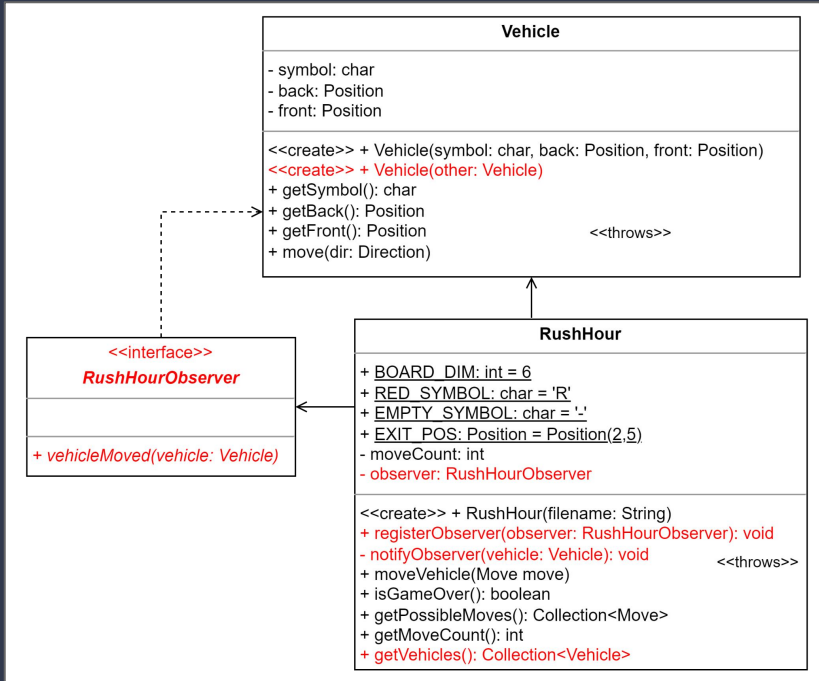


In the **Observer Pattern**, the **Subject** maintains a list of **Observers** that should be **notified** whenever something interesting happens.

Classes that **implement** the **Observer** interface can **register** to be notified.

- **The Observer Design Pattern** is a software design pattern in which there is some **subject** of interest.
 - We'd like to know when something interesting happens to the **subject**.
- The subject maintains a list of interested **observers**.
 - Each **concrete observer** implements the same interface.
 - The interface defines a method that can be used to **notify** the observer when something interesting happens.
- This semester, we have implemented the Observer Design Pattern when writing **graphical user interface (GUI)** applications.
 - To handle a user initiated event in the GUI, such as clicking on a button.
 - A small class or lambda expression that implements the **EventHandler<ActionEvent>** interface is registered with the control.
 - When the **event** occurs, the implementing code typically calls a method in a model class to effect a state change.
 - To update the GUI when a model state change has occurred.
 - A small class or the GUI application itself implements the observer interface defined by the model
 - The model notifies the observers of the state change by calling an observer interface method and the GUI is updated to reflect the change.

Observing Change



Remember to practice **pair programming** during your implementation.

- To support your GUI, you will need to make some changes to your **model** package.
 - Use the UML to the left to guide your implementation of the highlighted items
- **RushHourObserver** - an interface that receives notifications when a vehicle has moved.
- **RushHour** - the main class containing the game logic and maintains the game state.
 - **registerObserver** - registers an observer that wishes to be notified when a vehicle has moved.
 - **notifyObserver** - notifies an observer that a vehicle has moved.
 - a deep copy vehicle object, with updated positions to reflect the move, is passed to the **vehicleMoved** method.
 - **getVehicles** - returns deep copies of all vehicle objects in the game.
- Next, you and your team should begin implementing these changes.
 - You will need to update other **RushHour** methods to make calls to **notifyObserver**.

Meeting Times

This part of the project is due **Wednesday April 12th, 2023** at the start of class. Do not forget that **Midterm Exam III** is on **Monday April 10th**.

You should plan to **work together** as much as you can, even if that means setting up a remote meeting using Zoom or Discord.

Plan **at least 2 meetings** over the course of this part of the project. Each meeting should be at least **1 hour**. Following the example on the right, put the information for all scheduled meetings in a **meetings.txt** file and **push to your repository**.

Date/Time	Location	Area of Focus
4/8 @ 1:00PM	Discord	Observer
3/4 @ 6:00 PM	WhatsApp	Sharing classes for project
5/4 @ 1:00 PM	WhatsApp	RushHour and RushHourGUI

If You Made it This Far...

Your team is off to a good start, but you are **not quite** finished with **Part 2** of the Project yet.

You still need to create a **working implementation** of Rush Hour that can be played by a single human user via a **graphical user interface**.

If you have time remaining in class, you should begin reading the **full project description**, which you will find on MyCourses.

