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.

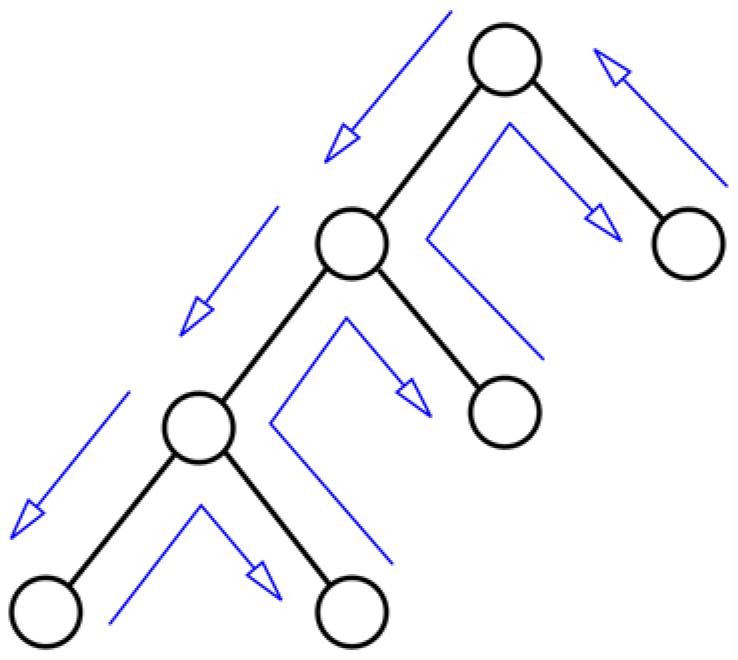
Coming Up

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 Protect Part 2 Due (start of class)		Unit 10 Mini-Practicum	
SUN	MON (4/17)	TUE	WED (4/19)	THU	FRI (4/21)	SAT
	Unit 11: Thread Cooperation				Unit 12: Networking	
	Project Part 3 Due (start of class)		Club Chèvre Assignment 11.1 Due (start of class)		Unit 11 Mini-Practicum	

You Are Here

Rush Hour Part 3

- This is the ***third*** part of a ***three*** part project.
  - This part is due on ***Monday April 17th, 2022*** at your ***class time***.
- In this part of the project, you will primarily be focused on creating a ***backtracking configuration*** that will attempt to find a series of vehicle moves that will win a Rush Hour game.
  - ***Zero or more*** moves may have already been made.
  - You will need to provide the ***list of winning moves***.
- Your configuration will be used to implement a solve feature for both your command line interface and graphical user interface.
- While you might be able to solve a game by using getPossibleMoves(), your project must implement a backtracking configuration.

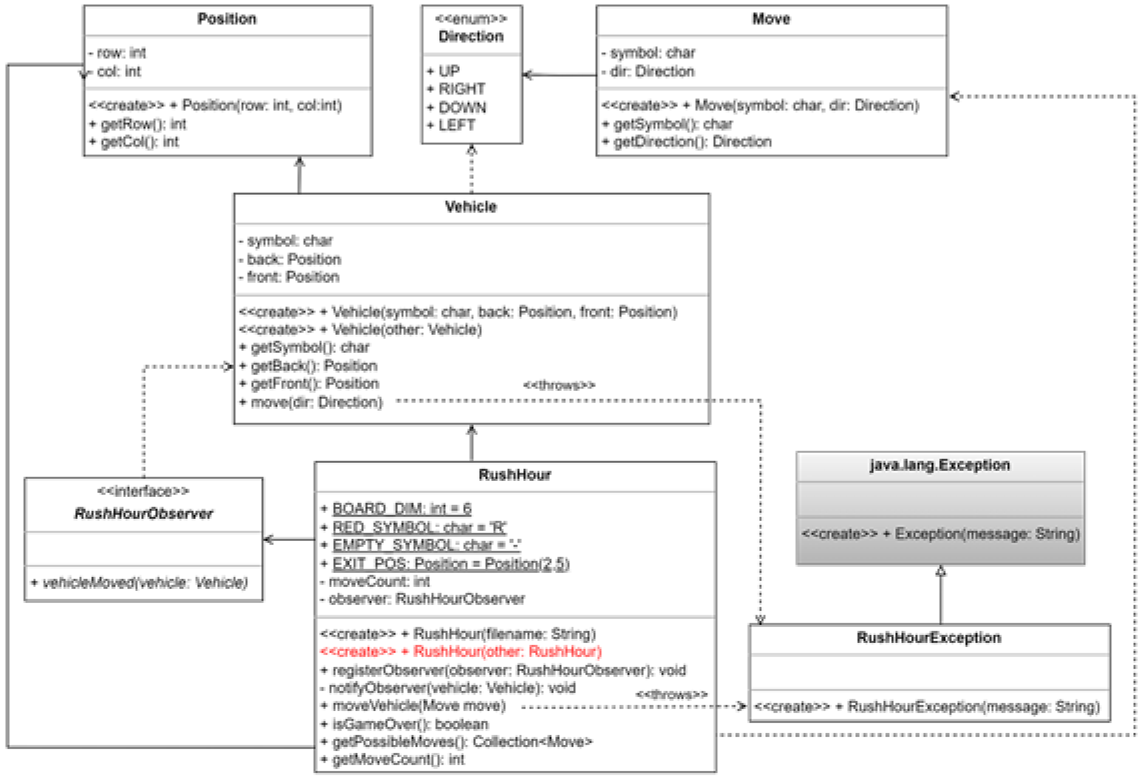


Pair Programming

- *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.

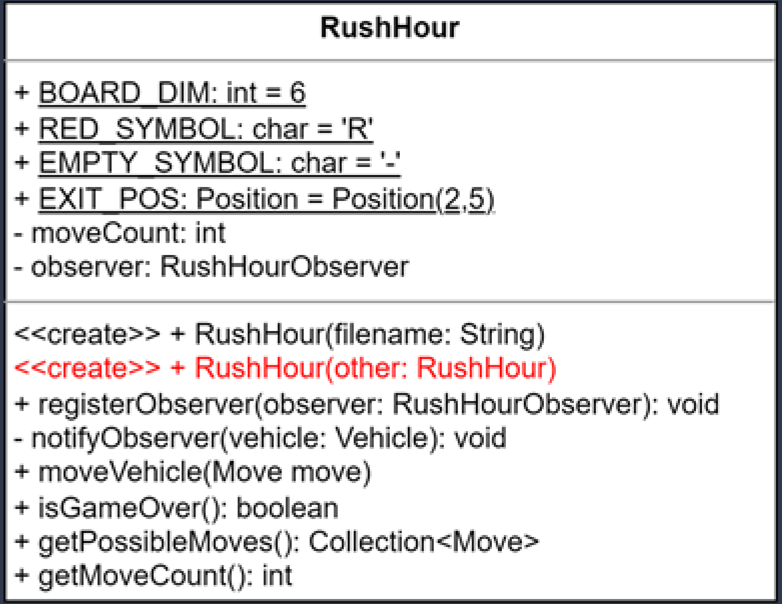
When you are the driver, you should be using Zoom or Discord to ***share your screen***. Be sure to ***push your code*** and ***switch roles*** every 10-20 minutes!

A Partial Design


Position
- row: int
- col: int
<<create>> + Position(row: int, col:int)
+ getRow(): int
+ getCol(): int
<<enum>>
Direction
+ UP
+ RIGHT
+ DOWN
+ LEFT
Move
- symbol: char
- dir: Direction
<<create>> + Move(symbol: char, dir: Direction)
+ getSymbol(): char
+ getDirection(): Direction
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)
<<throws>>
<<interface>>
RushHourObserver
+ vehicleMoved(vehicle: Vehicle)
RushHour
+ BOARD_DIM: int = 6
+ RED_SYMBOL: char = 'R'
+ EMPTY_SYMBOL: char = '-'
+ EXIT_POS: Position = Position(2,5)
- moveCount: int
- observer: RushHourObserver
<<create>> + RushHour(filename: String)
<<create>> + RushHour(other: RushHour)
+ registerObserver(observer: RushHourObserver): void
- notifyObserver(vehicle: Vehicle): void
+ moveVehicle(Move move)
+ isGameOver(): boolean
+ getPossibleMoves(): Collection<Move>
+ getMoveCount(): int
<<throws>>
java.lang.Exception
<<create>> + Exception(message: String)
RushHourException
<<create>> + RushHourException(message: String)

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

Copy Constructor



```
RushHour
-------------------------------------------------
+ BOARD_DIM: int = 6
+ RED_SYMBOL: char = 'R'
+ EMPTY_SYMBOL: char = '-'
+ EXIT_POS: Position = Position(2,5)
- moveCount: int
- observer: RushHourObserver
-------------------------------------------------
<<create>> + RushHour(filename: String)
<<create>> + RushHour(other: RushHour)
+ registerObserver(observer: RushHourObserver): void
- notifyObserver(vehicle: Vehicle): void
+ moveVehicle(Move move)
+ isGameOver(): boolean
+ getPossibleMoves(): Collection<Move>
+ getMoveCount(): int
```

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

- Your ***backtracking configuration*** will almost certainly need to create a ***deep copies*** of your `RushHour` implementation as it creates successors.
- To support the ability of a `RushHour` 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 `RushHour` implementation.
  - When called to create a new instance of the class, it should make a ***deep copy*** the RushHour 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!
  - Do not make a copy of the `observer`, but rather set it to `null`.
    - The `observer` was registered to be notified of the changes to ***original*** object.
    - If the `observer` is replicated in the ***copy constructor***, your GUI will be notified for every successor!

Backtracking

In this part of the project you will be creating a ***backtracking configuration*** that will attempt to solve a Rush Hour Game. You will also need to provide the list of selections needed to win.

Examine your code and think about how you will implement your configuration and answer the questions to the left in a ***backtracking.txt*** file. Be as detailed as possible. Push your file to the ***data directory*** of your repository when complete.

Remember:

- A ***configuration*** is at least a partial attempt at a solution.
- A ***successor*** is a new configuration that includes one additional choice.
- A configuration is ***invalid*** if it is impossible to find a solution from this point.
- A configuration is the ***goal*** if it is a valid solution to the problem.

What state will your ***configuration*** need to keep track of as it attempts to find a solution?

```
 public class Configuration {
    private Map<Character, Vehicle> vehicles; private
List<Move> moveHistory; useful for backtracking
}
```

It must track the positions of all vehicles on the board, along with the move history for solution reconstruction.

How will you make ***successor*** configurations?

```
for (Vehicle v : config.getVehicles()) {
   for (Direction dir : getValidDirections(v)) {
      try {
         Configuration newConfig = config.clone();  // deep copy
         newConfig.moveVehicle(v.getSymbol(), dir);
         successors.add(newConfig);
      } catch (RushHourException e) {
         // Ignore invalid moves
      }
   }
}
```

By attempting all valid moves for each vehicle and generating a deep copy of the board state for each resulting configuration.

How will you determine if a configuration is ***invalid***?

A configuration is invalid if any vehicle is out of bounds , any vehicle overlaps with another, or a move breaks vehicle orientation rules

How will you determine if the configuration is ***the goal***?

```
Vehicle red = vehicles.get('R');
return red.getFront().getCol() == boardWidth - 1;
```

The configuration is a goal if the goal vehicle (usually 'R') reaches the board's exit column.

Do you need to be concerned with ***cycles***? If so, how will you avoid them?

```
Set<Configuration> visited = new HashSet<>();
```

Yes, cycles must be avoided by storing visited configurations in a set using unique vehicle positions for comparison.

A Configuration

You may want to consider adding a ***main*** method to your configuration to manually test it. Use the following example to guide you.

```java
MySolver initial = new MySolver(game);
Backtracker<MySolver> backtracker
                       = new Backtracker<>(true);
MySolver solution = backtracker.solve(initial);
if (solution == null) {
   System.out.println("No solution.");
} else {
   System.out.println(solution);
}
```

You will of course need to create an instance of your RushHour Game implementation class to use as well.

- The `Backtracker` and `Configuration` classes have been provided in your repository.
    - You have also been provided with ***command-line interface*** output examples on MyCourses. There are screenshots of the updated ***graphical user interface*** at the end of the Part 3 Assignment. You can use these to guide the implementations of the new feature.
- Using the information that your team brainstormed in the previous activity, begin implementing a ***backtracking configuration*** to solve a `RushHour` game.
    - ***Do not*** change the provided classes.
    - It is recommended that your `Configuration` implementation be a separate class from `RushHour`.
    - Other than the copy constructor, do you need to refactor your `RushHour` class at all?
    - Consider overriding the `toString()` method in your configuration for use when debug mode is enabled in the backtracker.

Date/Time	Location	Area of Focus
4/15 @ 1:00PM	Discord	Get Successors
4/19 @ 7:00PM	WhatsApp	Ending tasks, rushhour, rushhourGUI
4/24 @ 12:00PM	WhatsApp	RushHourGUI implementation

Meeting Times

This part of the project is due ***Monday April 17th, 2023*** at ***class time***.

You should plan to ***work together*** with your team 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 left, put the information for all scheduled meetings in a *meetings.txt* file and *push to your repository.*

If You Made it This Far...

Your team is off to a good start, but you are ***not quite*** finished with ***Part 3*** of the Project yet. Remember that this part of the project is due on ***Monday April 17th, 2023*** at ***class time***.

You still need to implement the automatic solve feature in your ***command-line*** and ***graphical user interfaces***.

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