CSE 1325: Object-Oriented Programming
Lecture 09

Writing an OO Java Program
"Soup to Nuts"

"Henry, serve the nuts.
I mean, serve *our guests* the nuts!"

Mr. George F. Rice
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

The human brain is amazing, functioning 24 hours a day
from birth until death, stopping only when we take an Exam.

"The Thin Man" ©1934 MGM – Educational use is asserted

Today's Topics

- Understanding Requirements
  - Use Case and Activity Diagrams
  - Written Spec
  - Ambiguity
- Designing a Program in UML
  - Model View Controller (MVC)
  - Class Diagram
- Implementing a UML Design
  - Implementation
  - Regression Testing
  - Debugging
  - Packaging

Writing Programs 101

- So how do you write a program from scratch?
  - You need to gather **use cases**: What will the users need your program to do for them?
  - You need to derive and validate **requirements**: What specific features will enable the user's use cases?
  - You need to create a **design**: Which classes, data structures, and methods will collaborate to fulfill the requirements? In which order should they be developed?
  - You need to implement the **design** in code, resources, documentation, and tooling.
  - You need to integrate and **test** your solution frequently (with user engagement) to validate your design and verify your implementation.
  - You need to actually **deliver** a series of releases that delight (ahem) your customers.

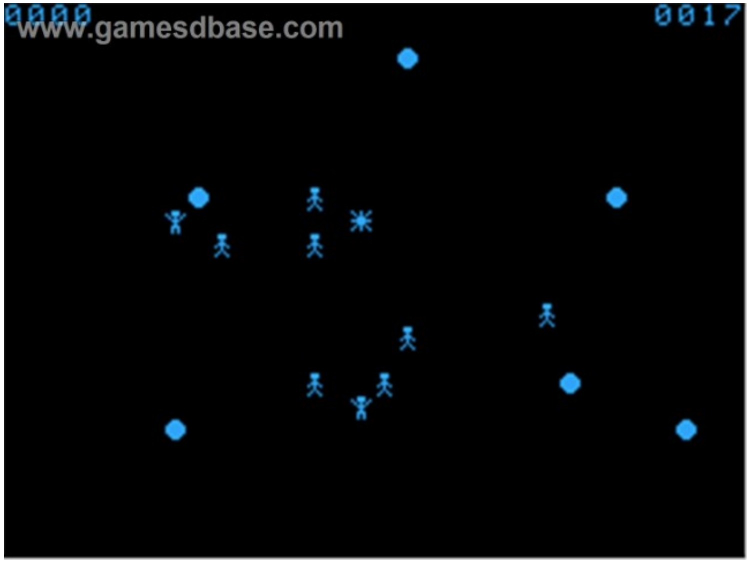
CSE1325 Emphasis

Writing Programs 101

- We’ll *briefly* walk through an example development
  - “Soup to nuts”, that is, start to finish
  - We’ll shortcut some details to keep it moving
- This will double as the exam review
  - We’ll discuss much of the technology we’ve learned thus far
- Management of a team is *hard*, often harder than code
- Our project will be a game
  - Because I like writing games
  - More than I like playing them, actually
  - But writing games is very helpful for learning a language

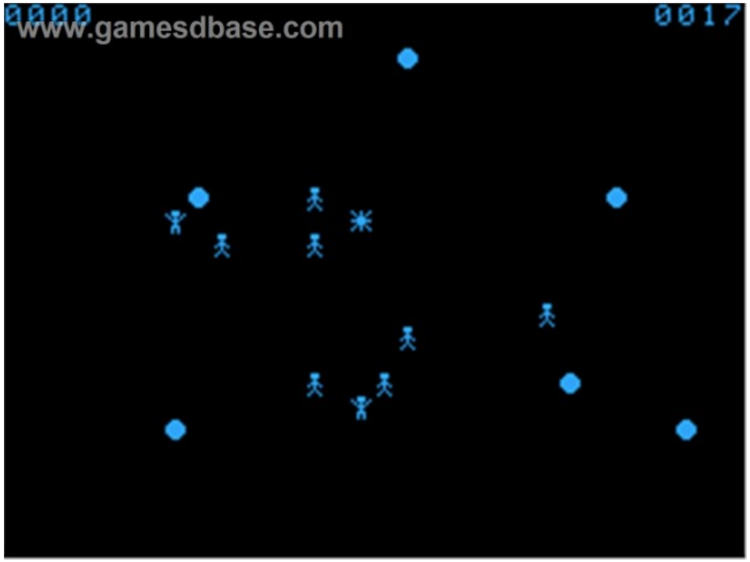
Really, Really Simple Requirements

- Recreate the 1970s classic game “Robots” using a CLI
- The game alternates turns between human and computer
- The human uses a keypad to move their own robot “Ralph”
  - Ralph can move 1 step, stay in place, or teleport randomly
- The computer's robots always take one step toward Ralph
- Robot collisions result in destruction of all robots involved
  - Collisions leave a lethal debris field
  - Colliding with a debris field is also a collision
- The player wins if all robots except Ralph are destroyed



Identify the Ambiguities

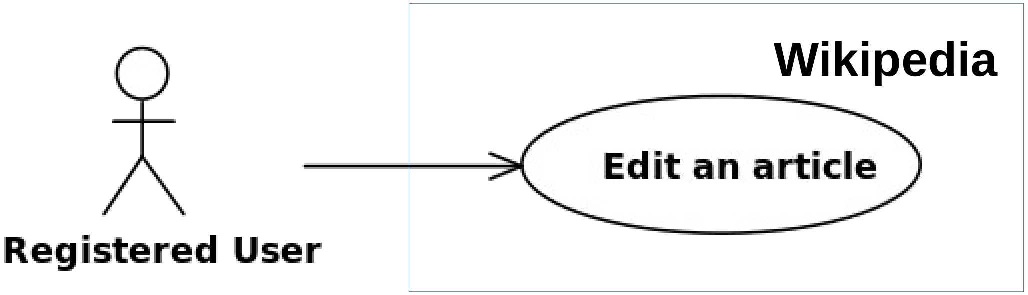
- Recreate the 1970s classic game “Robots” using a CLI
- The game alternates turns between human and computer
- The human uses a keypad to move their own robot “Ralph”
  - Ralph can move 1 step, stay in place, or teleport randomly
- The computer's robots always take one step toward Ralph
- Robot collisions result in destruction of all robots involved
  - Collisions leave a lethal debris field
  - Colliding with a debris field is also a collision
- The player wins if all robots except Ralph are destroyed



Modeling the Requirements: The Use Case Diagram

- A use case is a series of related interactions that enable an actor (e.g., a user) to achieve a goal.
  - Thus, a use case is always from the *actor's* perspective
  - A use case may be specified graphically using other UML diagrams and / or textually using a form
- A use case diagram graphically depicts the required use cases and their relationship to actors (users) and each other within a system
  - The diagram is read “<Actor> uses <System> to <Use Case>”

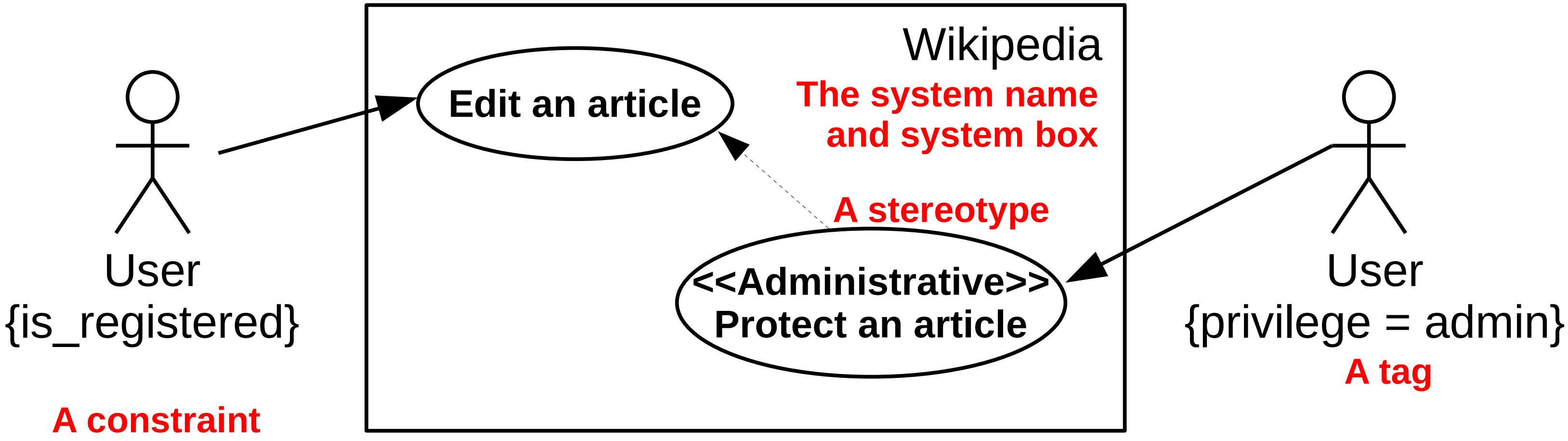
Read this diagram as
“Registered User uses Wikipedia to Edit an article.”



Use case diagram by Softzen and released as Public Domain

Use Cases are *Classes*

- Though drawn as ovals and stick figures, Use Cases and Actors are simply classes
  - You can use the same relationships as with classes
    - Include, extend, derive, ...
  - You can <<stereotype>>, {tag=value}, and {is_constrained} them
- Use cases help model *any* requirements



The system name and system box

A stereotype

A tag

A constraint

What Actors and Use Cases Do You See?

- Recreate the 1970s classic game “Robots” using a CLI
- The game alternates turns between human and computer
- The human uses a keypad to move their own robot “Ralph”
  - Ralph can move 1 step, stay in place, or teleport randomly
- The computer's robots always take one step toward Ralph
- Robot collisions result in destruction of all robots involved
  - Collisions leave a lethal debris field
  - Colliding with a debris field is also a collision
- The player wins if all robots except Ralph are destroyed

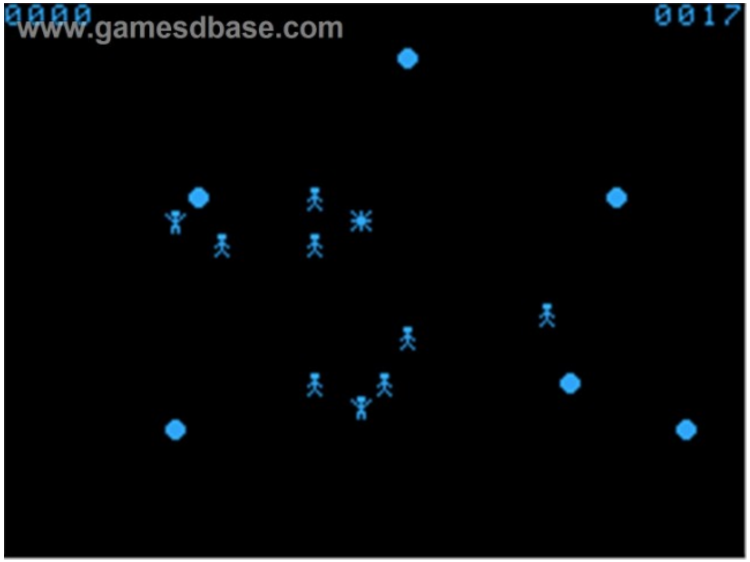
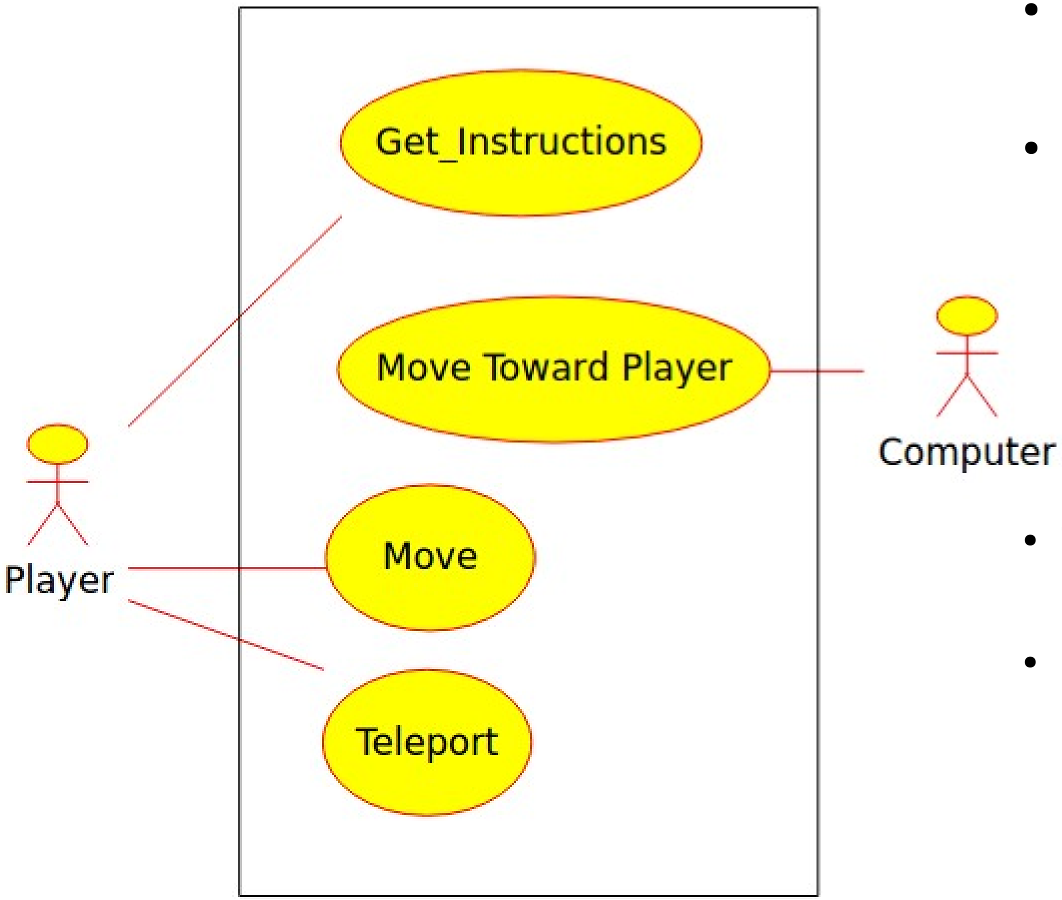


Image copyright gamesdbase.com. Used under non-commercial license.

Simple Use Case Diagram For Roving Robots



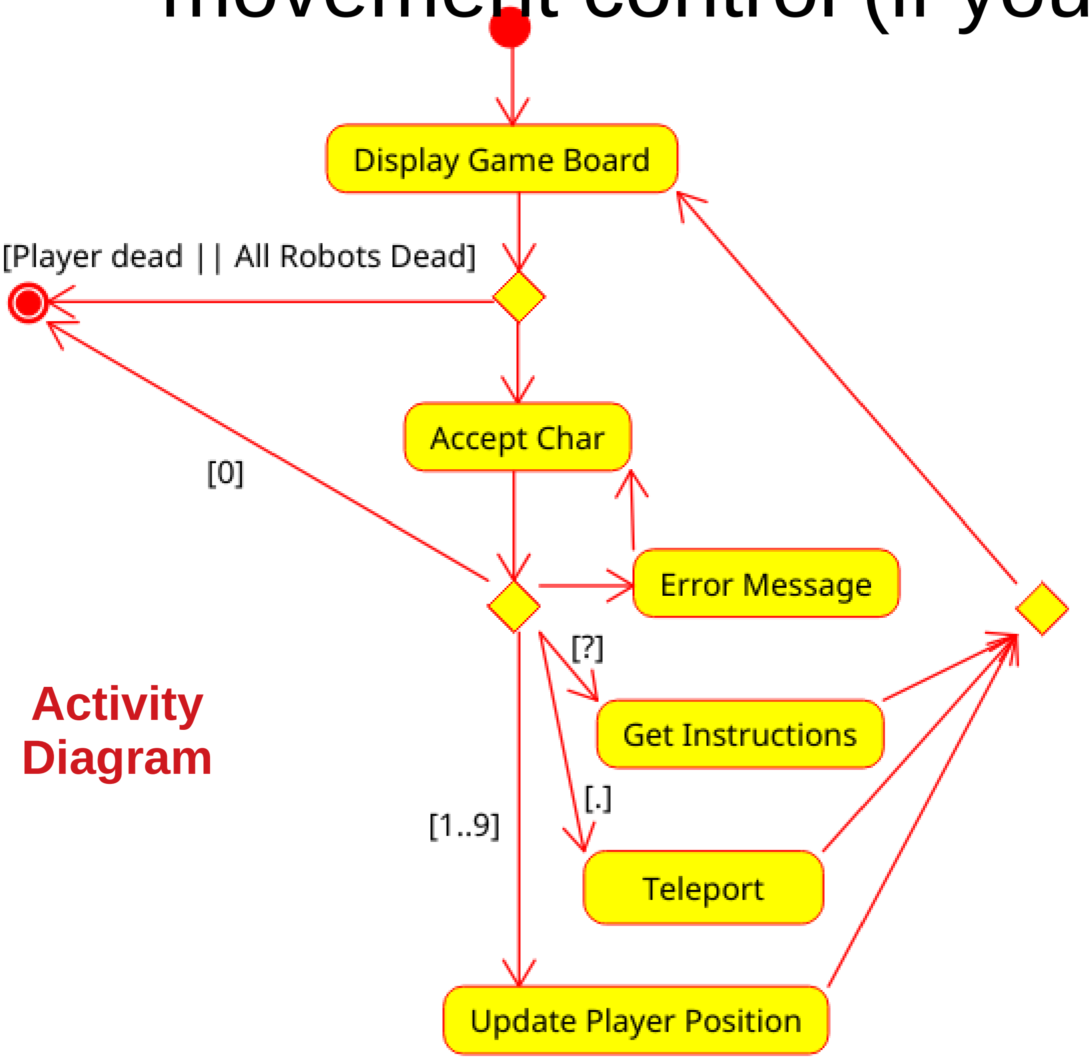
- Get Instructions
  - Print text explaining the game
- Move Toward Player
  - Alternate moves with player
  - Move diagonally (if possible) toward player
    - Robot collisions result in destruction of all robots involved
      - Collisions leave a lethal debris field
      - Colliding with a debris field is deadly
- Teleport
  - Move the player to a random map position
- Move
  - Accept a single char input from Player and move as indicated – up, down, left, right, diagonally, or no move
  - The player wins if all robots are destroyed

Use Case Diagram

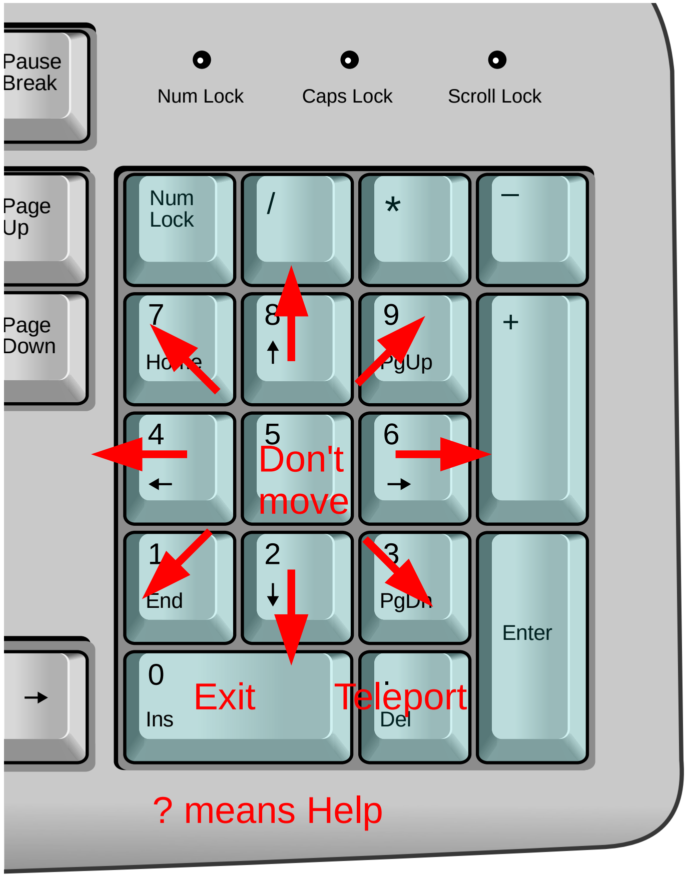
Text Documentation

Design: Activity Diagram and User Interface

- The numeric keypad works great for 8-way movement control (if you have one!)

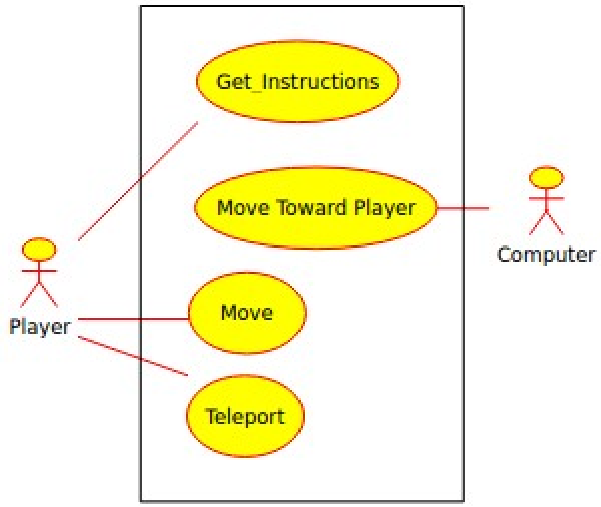


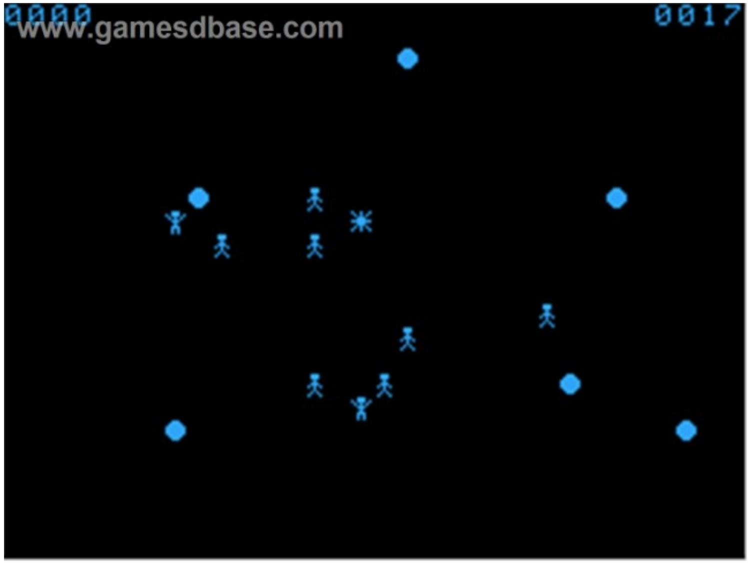
Activity Diagram



By Mysid (Self-made in Inkscape) [Public domain], via Wikimedia Commons
https://upload.wikimedia.org/wikipedia/commons/9/99/Numpad.svg

Simple Use Case Diagram For Roving Robots

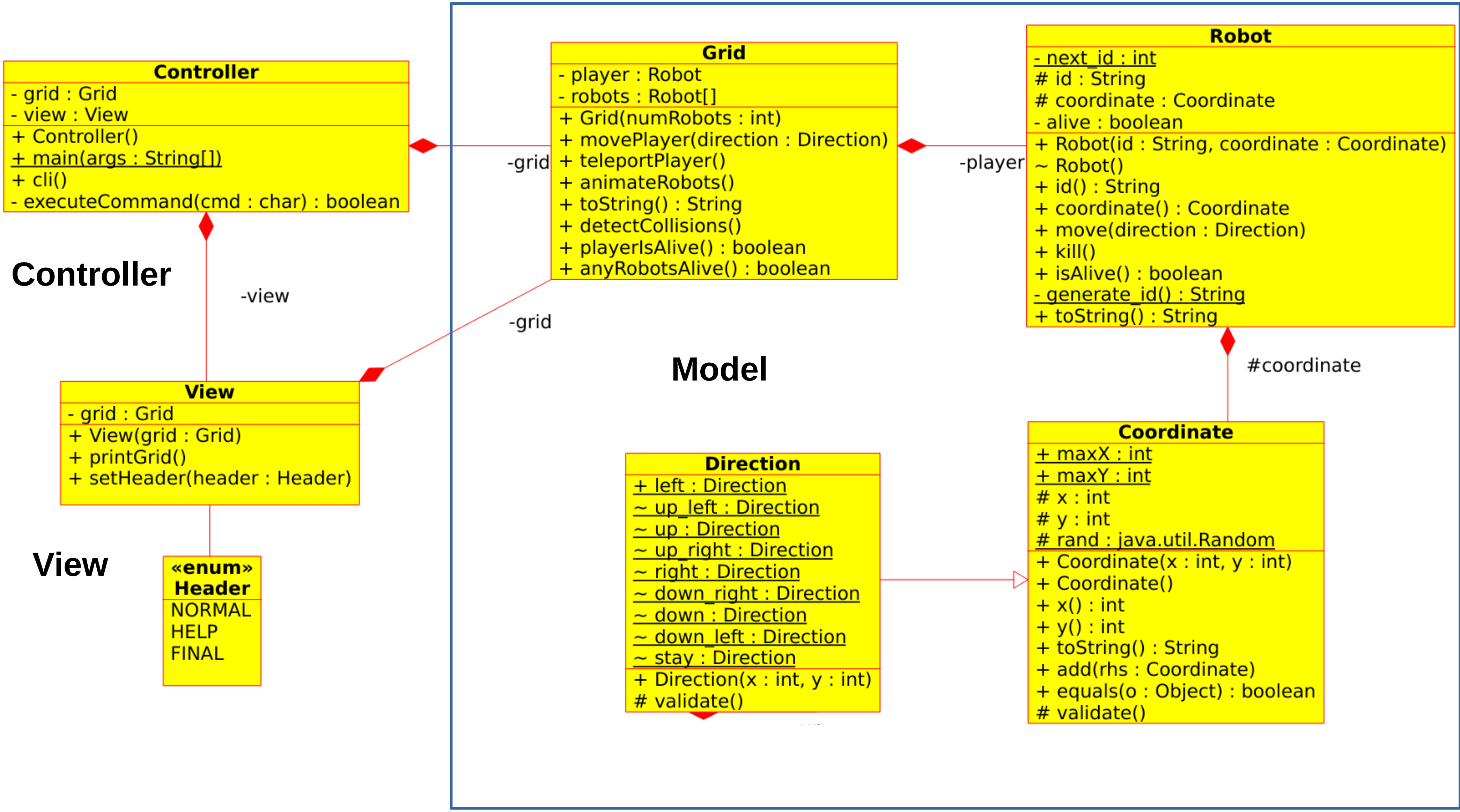




- Get Instructions
  - Print text explaining the game
- Move Toward Player
  - Alternate moves with player
  - Move diagonally (if possible) toward player
    - Robot collisions result in destruction of all robots involved
      - Collisions leave a lethal debris field
      - Colliding with a debris field is deadly
- Teleport
  - Move the player to a random map position
- Move
  - Accept a single char input from Player and move as indicated – up, down, left, right, diagonally, or no move
  - The player wins if all robots are destroyed

Given these requirements, what classes / objects do you see?

Class Diagram
For Roving Robots



Controller
- grid : Grid
- view : View
+ Controller()
+ main(args : String[])
+ cli()
- executeCommand(cmd : char) : boolean

Grid
- player : Robot
- robots : Robot[]
+ Grid(numRobots : int)
+ movePlayer(direction : Direction)
+ teleportPlayer()
+ animateRobots()
+ toString() : String
+ detectCollisions()
+ playerIsAlive() : boolean
+ anyRobotsAlive() : boolean

Robot
- next_id : int
id : String
coordinate : Coordinate
- alive : boolean
+ Robot(id : String, coordinate : Coordinate)
~ Robot()
+ id() : String
+ coordinate() : Coordinate
+ move(direction : Direction)
+ kill()
+ isAlive() : boolean
- generate_id() : String
+ toString() : String

View
- grid : Grid
+ View(grid : Grid)
+ printGrid()
+ setHeader(header : Header)

«enum»
Header
NORMAL
HELP
FINAL

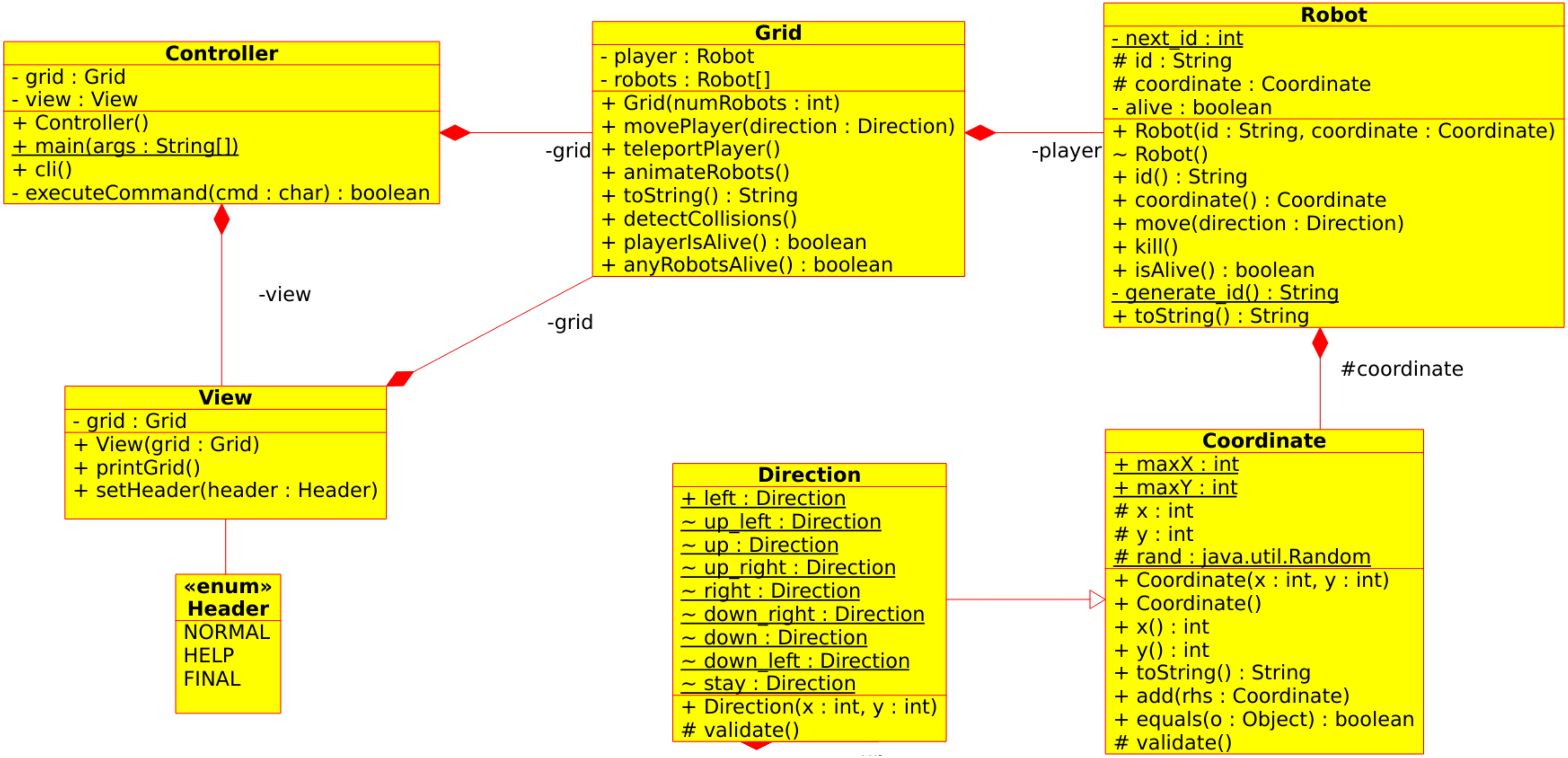
Direction
+ left : Direction
~ up_left : Direction
~ up : Direction
~ up_right : Direction
~ right : Direction
~ down_right : Direction
~ down : Direction
~ down_left : Direction
~ stay : Direction
+ Direction(x : int, y : int)
# validate()

Coordinate
+ maxX : int
+ maxY : int
# x : int
# y : int
# rand : java.util.Random
+ Coordinate(x : int, y : int)
+ Coordinate()
+ x() : int
+ y() : int
+ toString() : String
+ add(rhs : Coordinate)
+ equals(o : Object) : boolean
# validate()

Model-View-Controller (MVC) Pattern

- The MVC pattern separates the business logic ("model") from the data visualization ("view") and the human or machine user ("controller")
  - The **Model** contains the encapsulated data and any logic necessary to update that data – often on a server
  - The **View** presents the data to the consumer (which may be a user, a machine, or both). Multiple independent views are not uncommon.
  - The **Controller** acts on both the model and view to manage data flow.

Which Class Should We Write First?



Controller

```
- grid : Grid
- view : View
+ Controller()
+ main(args : String[])
+ cli()
- executeCommand(cmd : char) : boolean
```

Grid

```
- player : Robot
- robots : Robot[]
+ Grid(numRobots : int)
+ movePlayer(direction : Direction)
+ teleportPlayer()
+ animateRobots()
+ toString() : String
+ detectCollisions()
+ playerIsAlive() : boolean
+ anyRobotsAlive() : boolean
```

Robot

```
- next_id : int
# id : String
# coordinate : Coordinate
- alive : boolean
+ Robot(id : String, coordinate : Coordinate)
~ Robot()
+ id() : String
+ coordinate() : Coordinate
+ move(direction : Direction)
+ kill()
+ isAlive() : boolean
- generate_id() : String
+ toString() : String
```

View

```
- grid : Grid
+ View(grid : Grid)
+ printGrid()
+ setHeader(header : Header)
```

«enum»
Header

```
NORMAL
HELP
FINAL
```

Direction

```
+ left : Direction
~ up_left : Direction
~ up : Direction
~ up_right : Direction
~ right : Direction
~ down_right : Direction
~ down : Direction
~ down_left : Direction
~ stay : Direction
+ Direction(x : int, y : int)
# validate()
```

Coordinate

```
+ maxX : int
+ maxY : int
# x : int
# y : int
# rand : java.util.Random
+ Coordinate(x : int, y : int)
+ Coordinate()
+ x() : int
+ y() : int
+ toString() : String
+ add(rhs : Coordinate)
+ equals(o : Object) : boolean
# validate()
```

So Now What?

- Now... We Write!
  - The Class diagram is our “battle map”
  - Follow the dependencies on your Class diagram
    - Which class has few or no dependencies? Start there!
- Iterate through the diagram
  - For each class:
    - For each method
      - Write a simple version **with tests**!
      - Verify the tests pass (obviously)
      - Commit the code to git!
    - Repeat until the class is “complete”
  - Move on the the next class

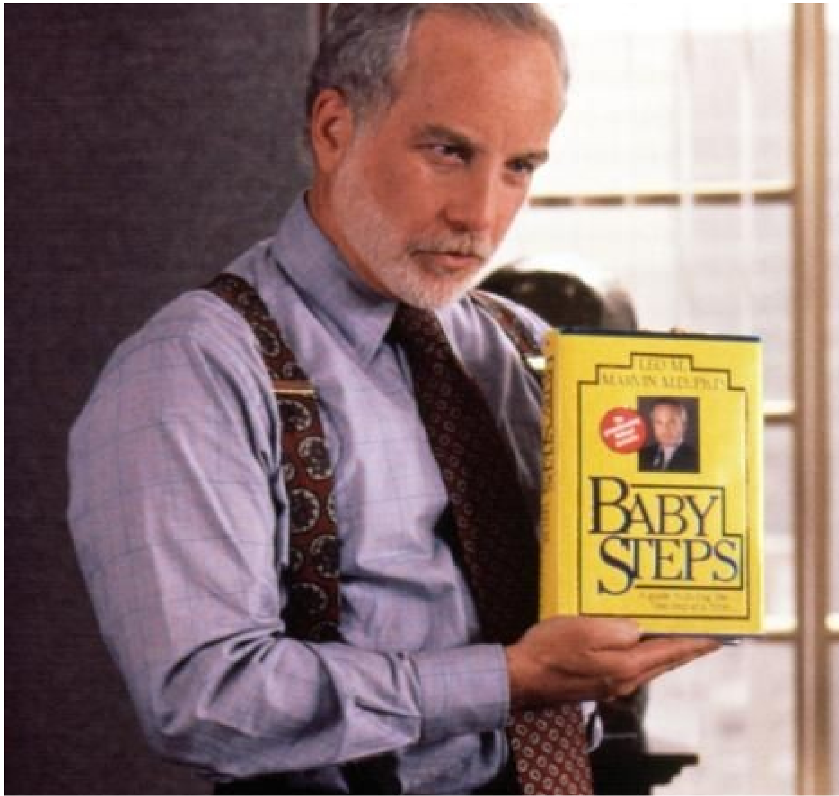
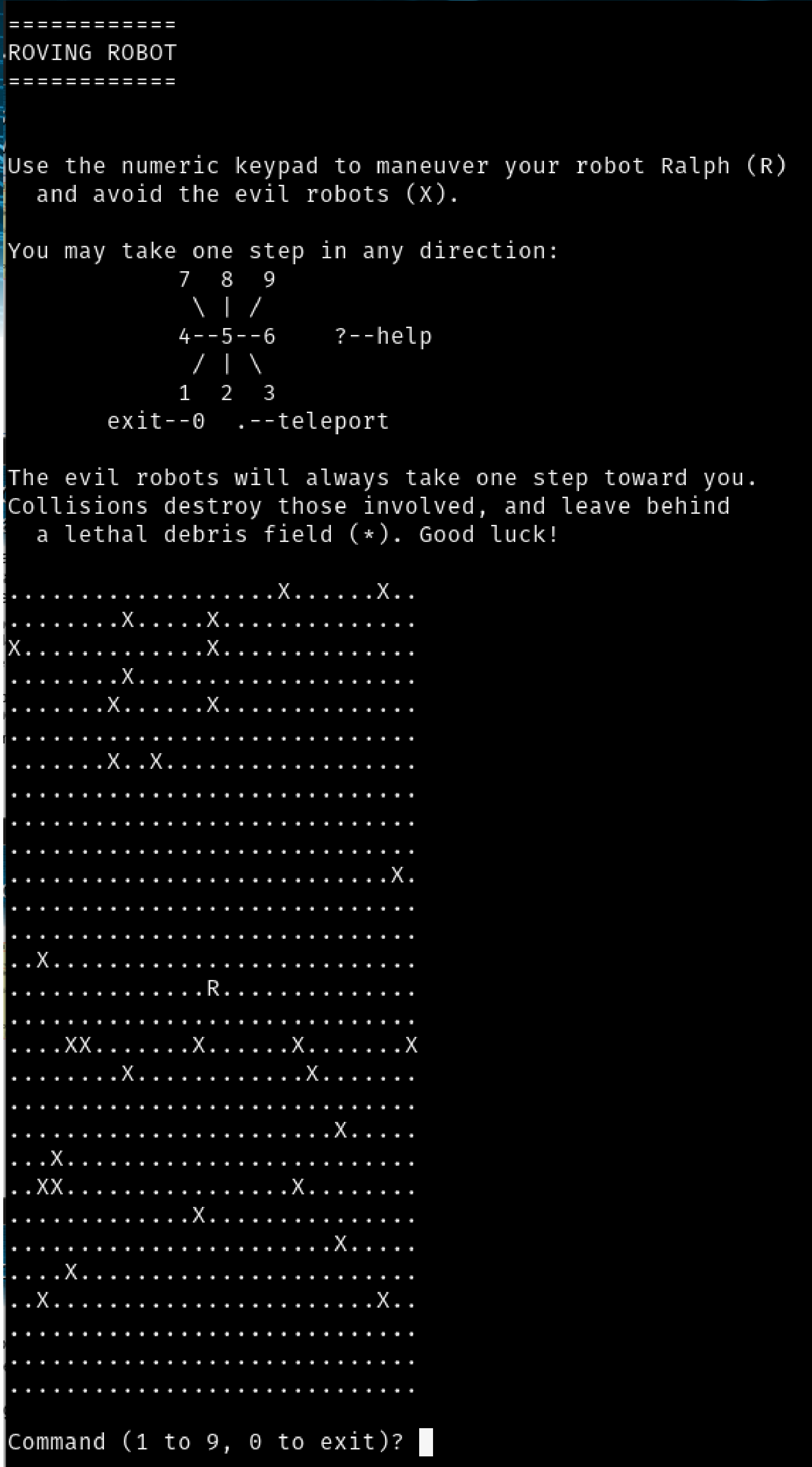


Image ©1991 by Buena Vista Pictures. Educational fair use is asserted.

We review the code here

Final Code

- The game is now playable and ready for beta test
  - "Alpha test" is field testing without the final feature set
  - "Beta test" is field testing with the final feature set ("feature freeze")
  - "Acceptance test" is field testing with intent to not change the code ("code freeze")
- Serious design or code issues may require a "thaw"!



```
============
ROVING ROBOT
============


Use the numeric keypad to maneuver your robot Ralph (R)
  and avoid the evil robots (X).

You may take one step in any direction:
            7  8  9
             \ | /
            4--5--6    ?--help
             / | \
            1  2  3
       exit--0  .--teleport

The evil robots will always take one step toward you.
Collisions destroy those involved, and leave behind
  a lethal debris field (*). Good luck!

...................X......X..
........X.....X..............
X.............X..............
........X....................
.......X......X..............
.............................
.......X..X..................
.............................
.............................
.............................
...........................X.
.............................
.............................
..X..........................
..............R..............
.............................
....XX.......X......X.......X
........X............X.......
.............................
.......................X.....
...X.........................
..XX................X........
.............X...............
.......................X.....
....X........................
..X.......................X..
.............................
.............................
.............................

Command (1 to 9, 0 to exit)?
```

Step 15: Packaging

- For Windows, .msi or a setup.exe is common
  - Commercial tools are best – Visual Studio includes one
  - The Windows Store is still rather... sparse
- Mac OS X is somewhat similar to Windows
  - Mac supports flat, meta-, distribution, and hybrid packages that work on multiple OS versions, but the Homebrew package manager works well when available
- For Linux, flatpack (or snap) packages are the latest thing
  - AppImage is legacy, but simple and highly portable
    - AppImage Hub is the universal “app store”, but not yet well-populated (~1000 apps)
    - Flatpak and (Ubuntu-specific) Snap are also (in theory) universal but require infrastructure
  - Legacy systems use Red Hat Package Manager (.rpm) or Debian (.deb)
    - Repositories of packages (“repos”) are massive with >10,000 programs available to install
  - **`.tgz and ./configure ; make ; make install`** are still occasionally used
- Enterprise package management solutions are commonly deployed in the corporate world for patch and app deployment

For Next Class

- NO assignment or post-lecture quiz this week
- **Exam #1** (16⅔% to 21⅔% of your final grade) on *Thursday*
- Study sheet and practice exam are on Canvas

Questions?

Owl by PeterKraayvanger per the Pixabay License
https://pixabay.com/photos/owl-bird-animal-uhu-european-uhu-62703/