CSE 1325: Object-Oriented Programming

Lectures 06 and 07 – Chapters 06 and 07

(These two lectures do not follow the book.)

Writing an OO C++ Program

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment

Lecture #5
Quick Review

- A statement that introduces a name with an associated type into a scope is a **declaration**.

  A statement that also fully specifies that entity is a **definition**.
- Why should programs #include header files instead of cpp files? **The .cpp files include all of the definitions. This causes circular references with non-trivial programs, resulting in no valid compile order. The headers contain (mostly) declarations.**
- True or False: Memory is allocated for a variable when it is declared. **False – memory is allocated when it is defined.**
- True or False: It is an error to include the same file in both the header and the body of a class file. **False**
- What is the purpose of "#ifndef __FILE_H" and "#define __FILE_H 2016" at the top of a header file? **To avoid including a header file more than once during a compile**

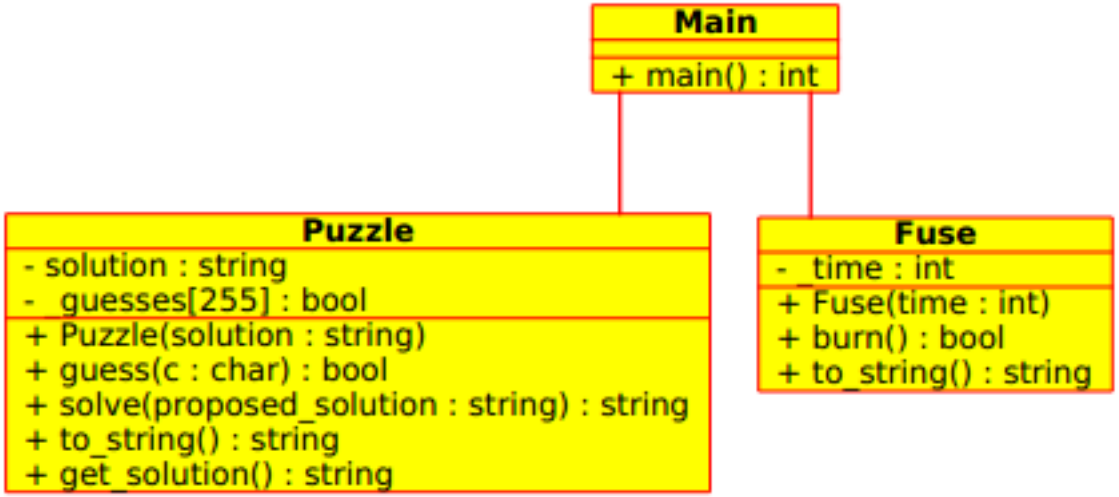
Lecture #5
Quick Review

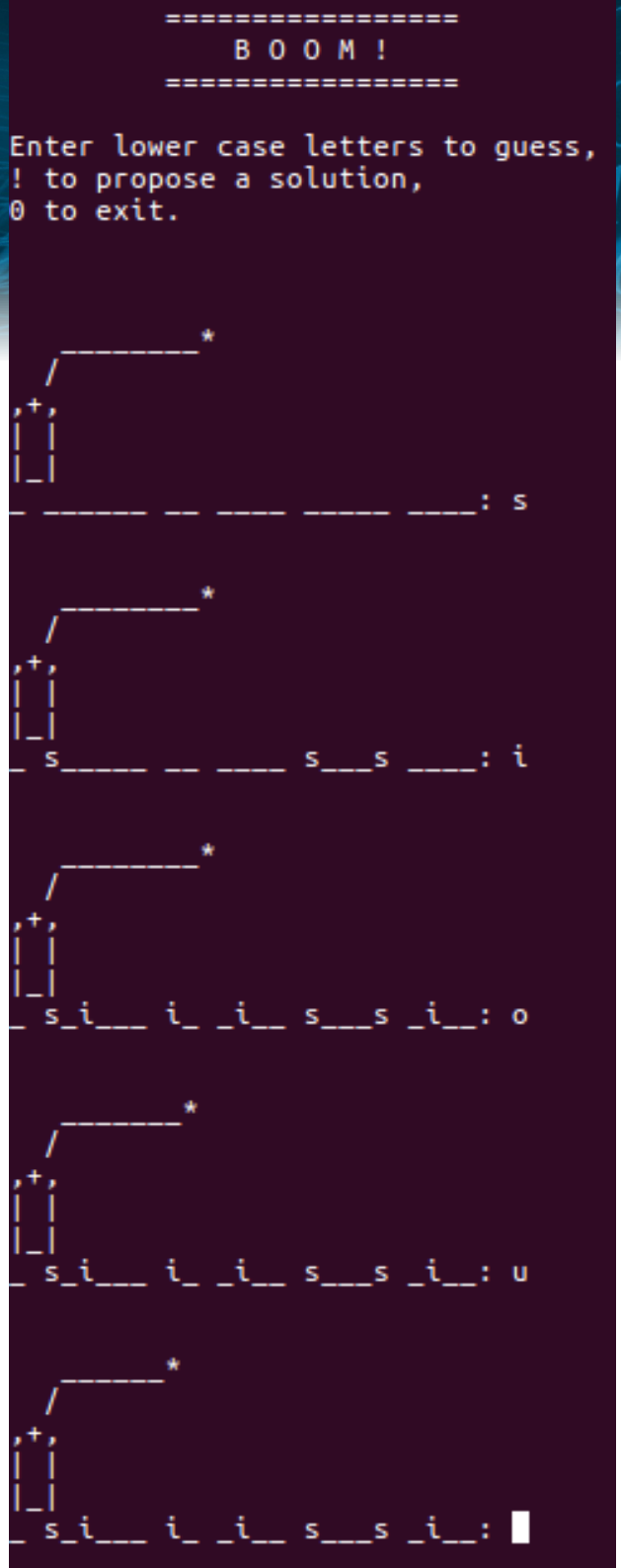
- A special kind of member function that initializes an instance of its class is a **constructor**.
  Direct assignment is a shortcut to assign a value to a class's private variables in a constructor.
  Delegated constructors (or **constructor chaining**) implements one constructor by calling another.
- True or False: A default constructor with no parameters is always provided.
  False – It is only provided if you define no constructors yourself.
- To modify a parameter, the parameter must be passed by **reference (&)** - otherwise, the method only has a copy of the parameter.
  A **const** reference cannot modify the parameter even though it references to the original data.
- Constexpr specifies that a variable can be evaluated at **compile time**.
- A **namespace** is a named scope, which can be accessed via the **using** keyword or the **::** operator.
- A **Makefile** builds a C++ program optimally with a simple “make” command.

Questions on Homework #3?

This assignment involves writing a word / phrase guessing game (always a good way to learn a new language and new technology). We'll utilize a Makefile, use either the ddd or gdb debugger, employ git (of course!), and use Umbrello to design our class and use case diagrams.

Due Thursday, September 14 at 8 am.





Today's Topics

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

A 2-Lecture Departure

- Stroustrop's PPP teaches C++ and general programming
  - Chapters 6 & 7 demonstrate “structured programming” by an exercise in “functional decomposition”
  - **Structured Programming - A style of programming focused on subroutines and functions, block structures, and loops.**
  - **Functional Decomposition - Resolving a problem into a hierarchy of constituent parts such that the original problem can be reconstructed from those parts.**
- CSE 1325 teaches Object-Oriented Programming (OOP)
  - **Object-Oriented Programming – A style of programming focused on the design and use of classes and class hierarchies.**
    → OOP = Polymorphism + Inheritance + Encapsulation.
  - PPP doesn't get around to that (lightly) until Chapter 9
  - Hence, we will use a different example than PPP

Chapters 6 and 7 are very useful for learning C++ programming.
We highly recommend that you study these chapters in addition to this lecture.

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

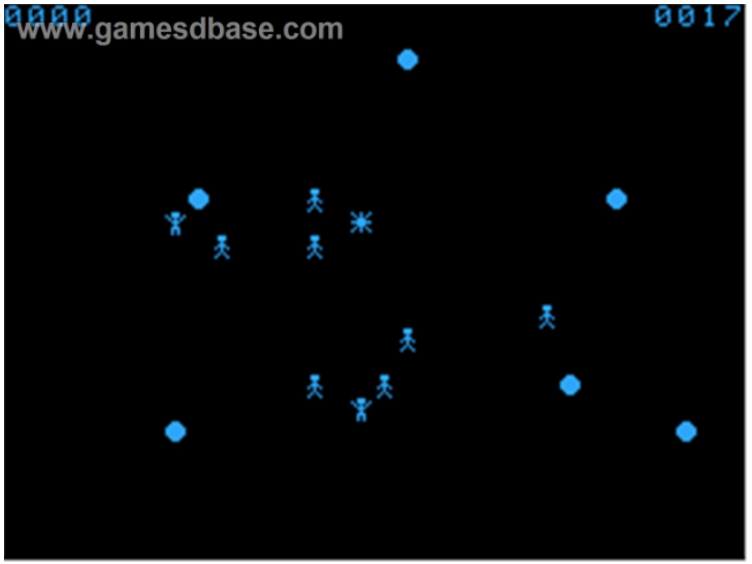


Image copyright gamesdbase.com. Used under non-commercial license.

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

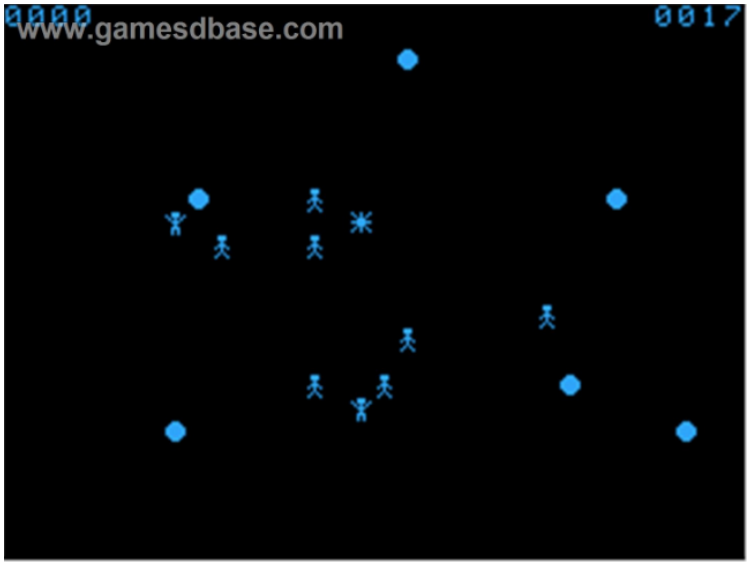
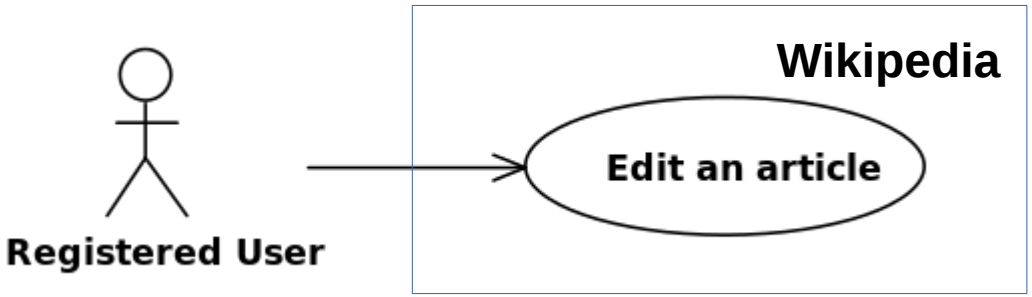


Image copyright gamesdbase.com. Used under non-commercial license.

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>”

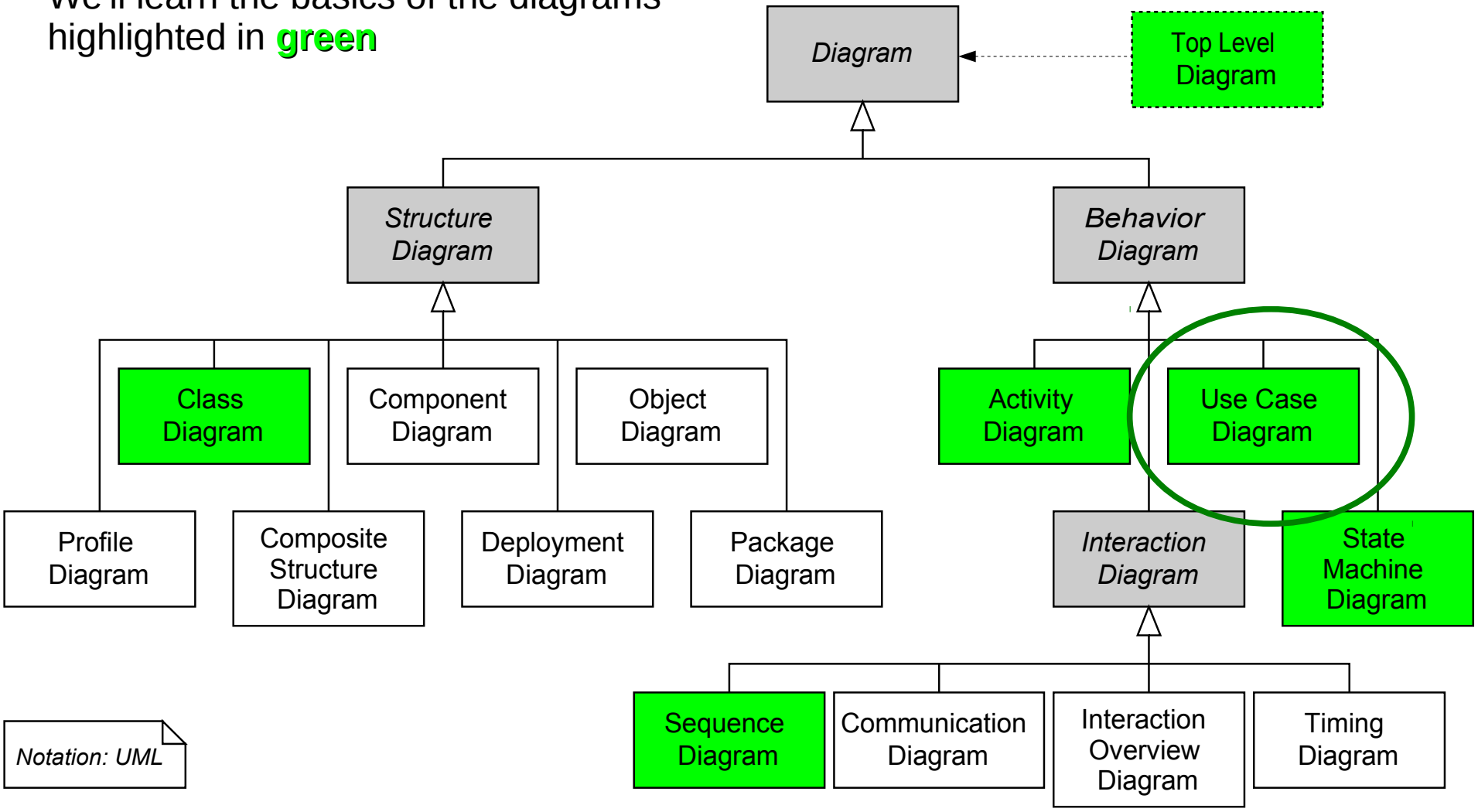
Read this diagram as
“Registered User uses Wikipedia to Edit an article.”



Use case diagram by Softzen and released as Public Domain

The Use Case Diagram in Context

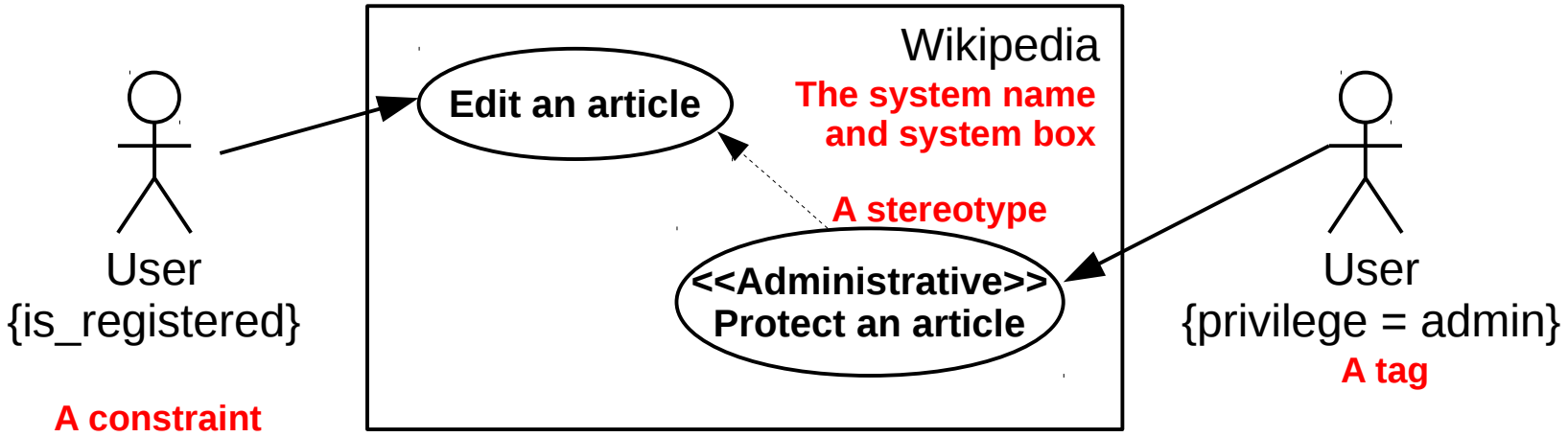
We'll learn the basics of the diagrams highlighted in **green**



Original source: Wikipedia, Public Domain SVG

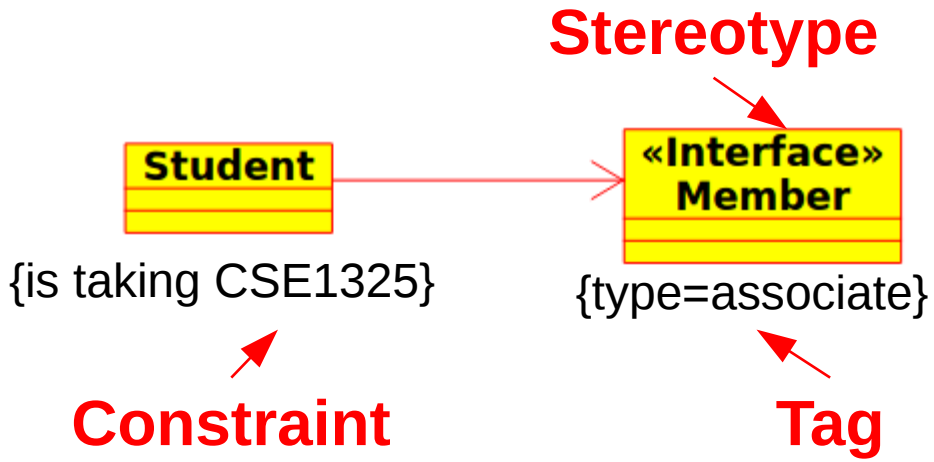
Use Cases are *Classes*

- 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



Extending the UML

- UML offers 3 distinct ways to extend its notation
  - **Stereotype** – guillemets-enclosed specialization, e.g., «interface» may represent a C++ class declaration (a .h file) rather than a definition (a .cpp file)
  - **Tag** – curly-brace-enclosed assignment of value to tag name, which e.g., may prove useful to identify attributes of a class or control code generation
  - **Constraint** – curly-brace-enclosed Boolean expression that adds a condition, restriction, or assertion on the class



A given tool may specify certain semantic interpretations it will apply to specific extensions, but the UML spec does NOT limit what stereotypes, tags, or constraints you may create for your diagrams.

Documenting a Use Case

- Each use case may be documented using one or more of the following
  - Text and pictures, often from a template
  - UML Sequence diagram
  - UML Statechart diagram
  - UML Activity diagram
- We'll cover the Sequence diagram next class, and the Statechart and Activity diagrams later this year

Use Case: Edit an article

Primary Actor: Member *(Registered User)*

Scope: a Wiki system

Level: ! *(User goal or sea level)*

Brief: *(equivalent to a user story or an epic)*

The member edits any part (the entire article or just a section) of an article he/she is reading. Preview and changes comparison are allowed during the editing.

Stakeholders

...

Postconditions

Minimal Guarantees:

Success Guarantees:

- The article is saved and an updated view is shown.
- An edit record for the article is created by the system, so watchers of the article can be informed of the update later.

Preconditions:

The article with editing enabled is presented to the member.

Triggers:

The member invokes an edit request (for the full article or just one section) on the article.

Basic flow:

1. The system provides a new editor area/box filled with all the article's relevant content with an informative edit summary for the member to edit. If the member just wants to edit a section of the article, only the original content of the section is shown, with the section title automatically filled out in the edit summary.
2. The member modifies the article's content till satisfied.
3. The member fills out the edit summary, tells the system if he/she wants to watch this article, and submits the edit.
4. The system saves the article, logs the edit event and finishes any necessary post processing.
5. The system presents the updated view of the article to the member.

Extensions:

2-3.

a. Show preview:

1. The member selects *Show preview* which submits the modified content.
2. The system reruns step 1 with addition of the rendered updated content for preview, and informs the member that his/her edits have not been saved yet, then continues.

b. Show changes:

1. The member selects *Show changes* which submits the modified content.
2. The system reruns step 1 with addition of showing the results of comparing the differences between the current edits by the member and the most recent saved version of the article, then continues.

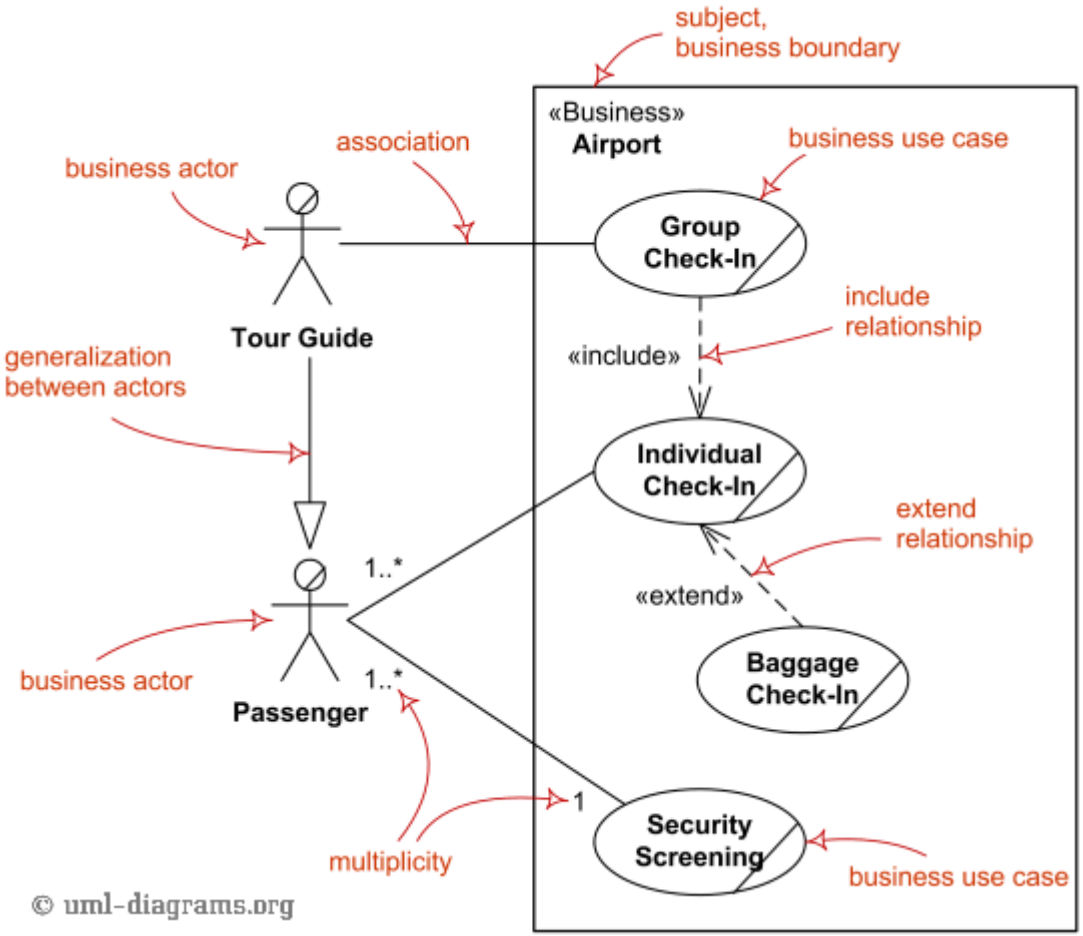
c. Cancel the edit:

1. The member selects *Cancel*.
2. The system discards any change the member has made, then goes to step 5.

4a. Timeout:

Use case example template is (c) Wikimedia Foundation.
Used under CC-BY-SA and GFDL with no invariant sections, front-cover texts, or back-cover texts

Basic Graphical Elements of a UML Use Case Diagram



http://www.uml-diagrams.org/use-case-diagrams.html

Used with written permission.

What 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

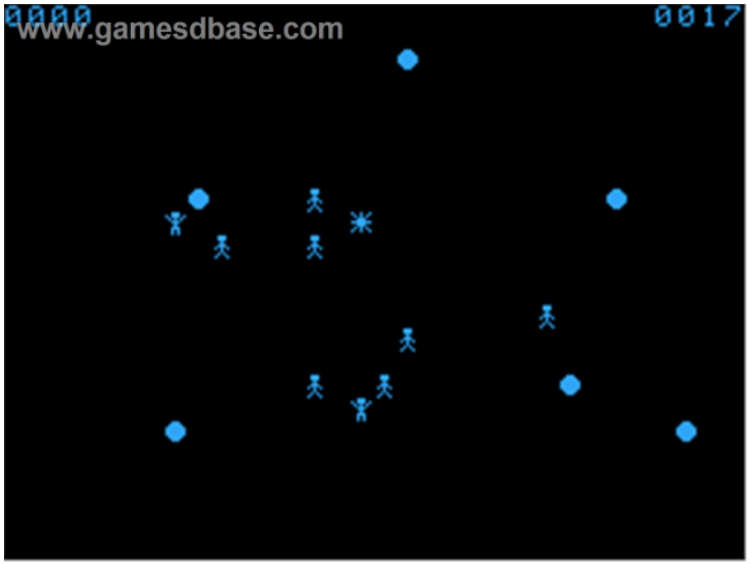
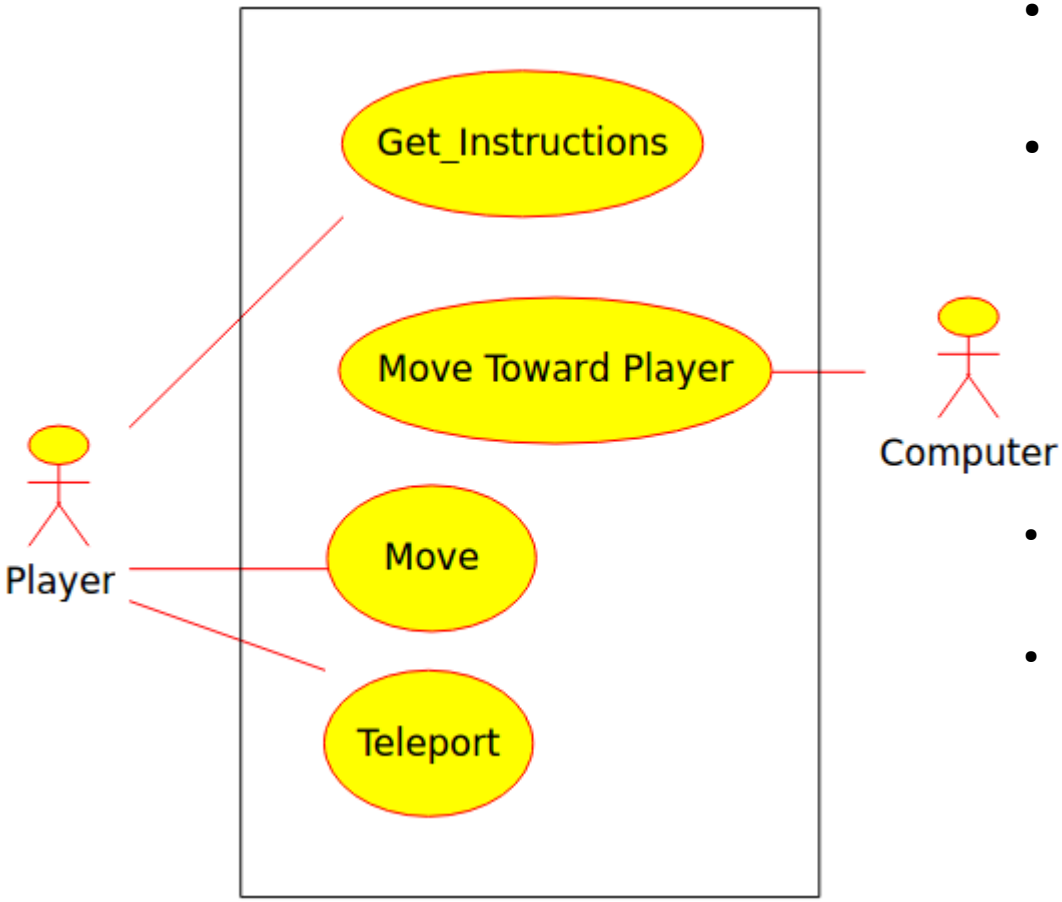


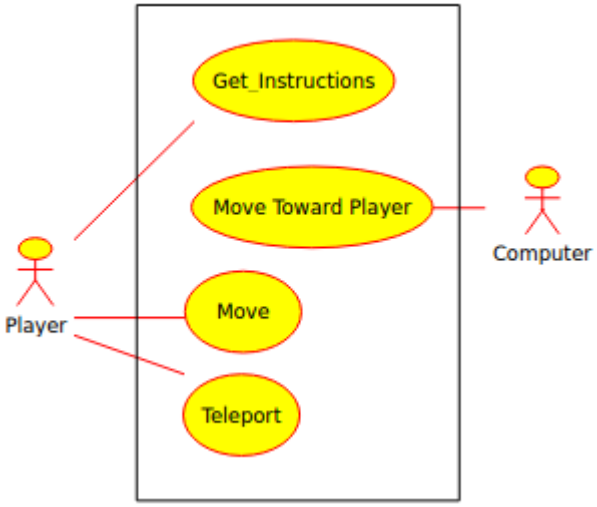
Image copyright gamesdbase.com. Used under non-commercial license.

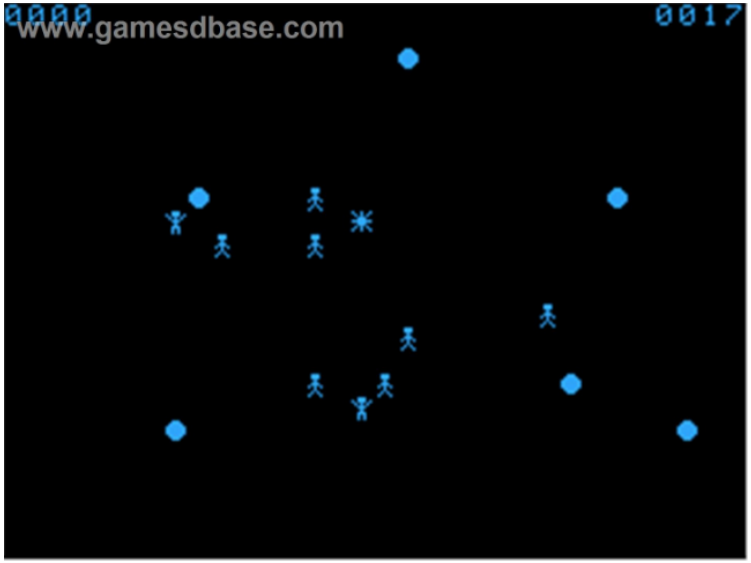
Simple Use Case Diagram For Roving Robots



- 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

Simple Use Case Diagram For Roving Robots

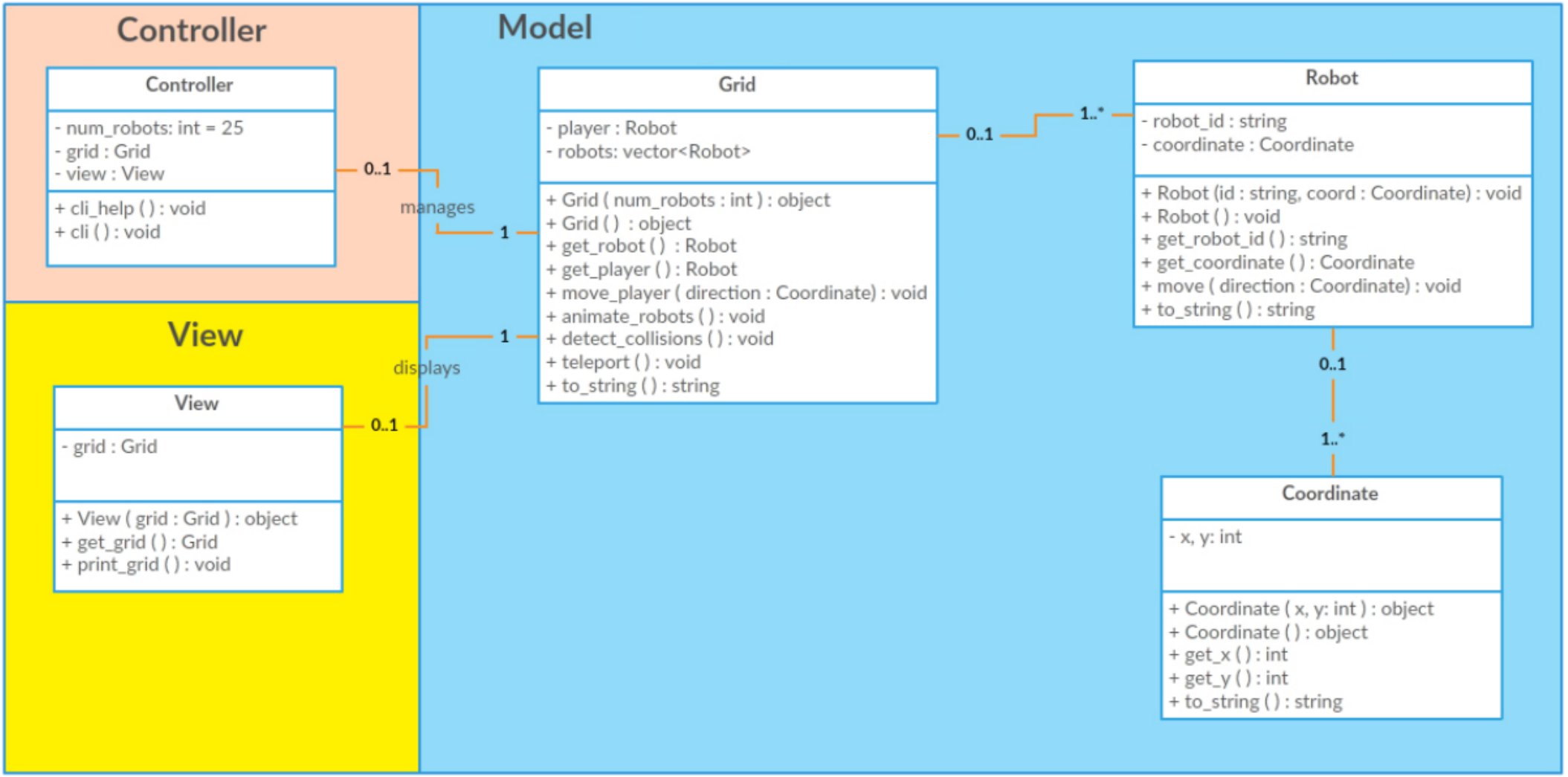




- Get Instructions
  - Print text explaining the game
- Move Toward Player
  - Alternate moves with player
  - Move diagonally (if possible) toward player
    - Robot collisions result in destruction of all robots involved
      - Collisions leave a lethal debris field
      - Colliding with a debris 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?

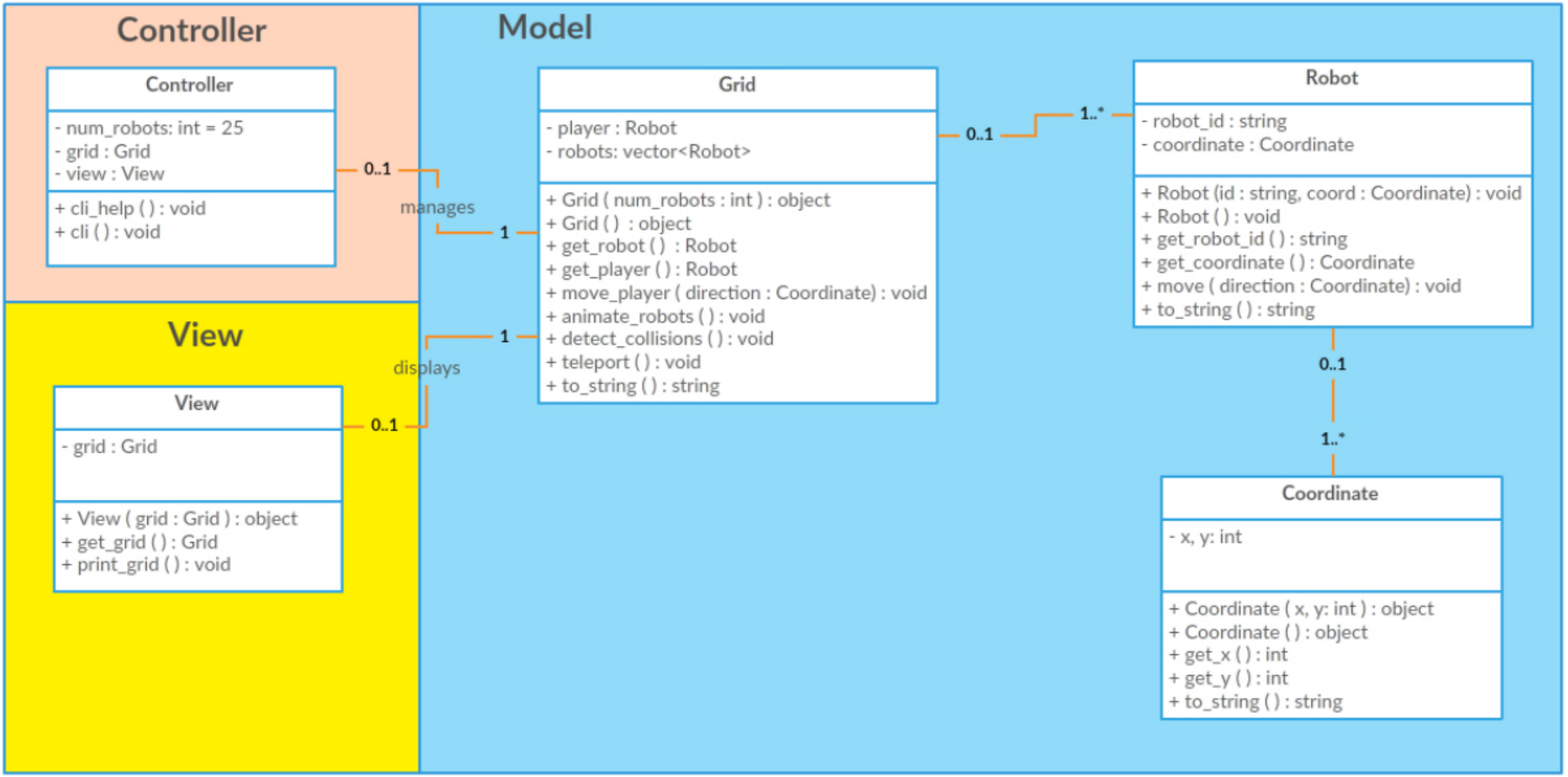
Class Diagram For Roving Robots



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?



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

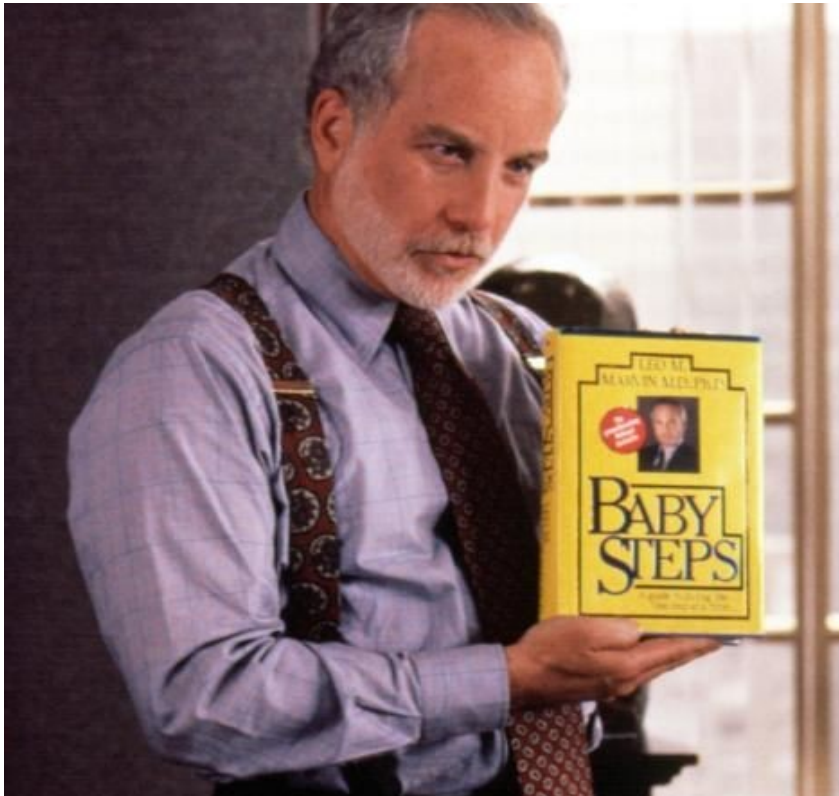
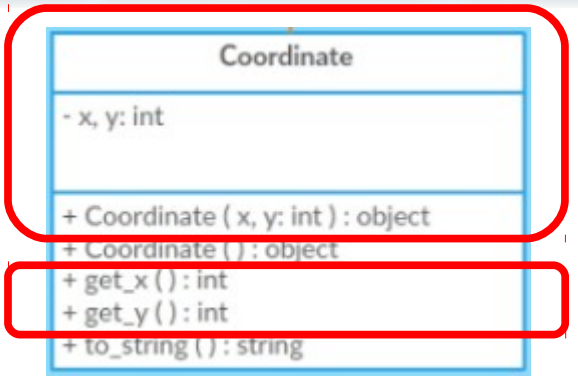


Image ©1991 by Buena Vista Pictures. Educational fair use is asserted.

Step 1: coordinate.h

- We write **part** of the Coordinate class
  - Just enough to test
  - Don't write too much before you test
- coordinate.h is the *interface* to a coordinate
  - The implementation will be in coordinate.cpp




```
#ifndef COORDINATE_H
#define COORDINATE_H 2017
class Coordinate {
  public:
    Coordinate(int x, int y)
      : _x{x}, _y{y} { }
    int get_x();
    int get_y();
  private:
    int _x;
    int _y;
};
#endif
```

These avoid redefinition errors – for .h only. Some compilers support **#pragma once**, but it is not part of the standard,

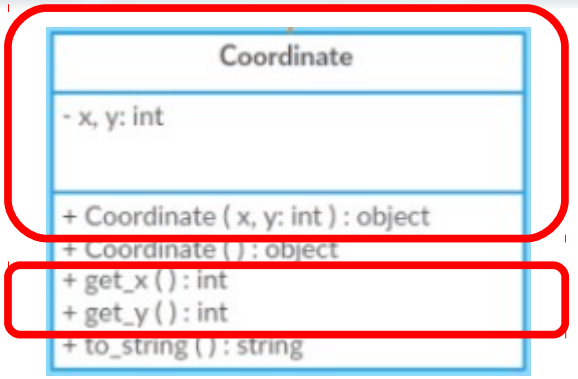
The "constructor" that initializes every object created by this class. This code runs *before* your code can use the object.

"Getters" - because they "get" private variables via a public interface. Note: A public variable can be changed by any code with visibility - **"getters" control access to private variables!**

The private variables that identify a position on a grid as (x, y)

Step 1: coordinate.cpp

- coordinate.cpp is the *definition* of the declarations in coordinate.h
- ALWAYS separate declarations into a .h and define them in a .cpp of the same name
  - This is good computer science
  - This is critical to building large c++ programs!




```
#include "coordinate.h"

int Coordinate::get_x( ) {return _x;}
int Coordinate::get_y( ) {return _y;}
```

This is the class This is the class member

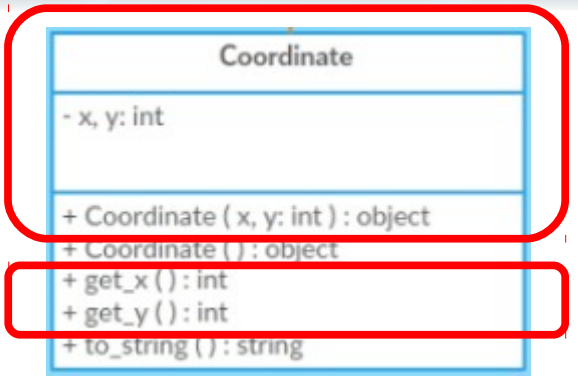
ALWAYS include the .h in the associated .cpp. This ensures that your interface is consistent with the implementation.

Remember, " " means to check in the local directory for the .h file first, while < > means check only in the system libraries.

You CANNOT redefine "class Coordinate" in the .cpp file – it has already been defined in the .h file. Instead, define the members that were only declared as part of the class definition – in this case, get_x() and get_y().

Step 1 First Test – Does it compile?

- First we need a Makefile
  - ALWAYS build with a Makefile, even for trivial builds
  - Good habits make good code




```
# Makefile for Roving Robots
CXXFLAGS = -std=c++11

all: coordinate.o

debug: CXXFLAGS += -g
debug: coordinate.o

coordinate.o: coordinate.cpp coordinate.h
	$(CXX) $(CXXFLAGS) -c coordinate.cpp

clean:
	rm -f *.o a.out
```

Use C++ version 11 OR 14

By default (for now), just compile coordinate.cpp

To debug, also use the -g flag

Build coordinate.o if .h or .cpp changed

Bash command to build coordinate.o

For a “make clean” command: Remove the .o files plus a.out

```
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ make
g++ -std=c++11 -c coordinate.cpp
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$
```

Step 1 Next Test – Regression Test

```
#include "coordinate.h"
#include <iostream>
#include <string>

using namespace std;

int main() {
  string expected = "(0,0)  (0,3)  (0,6)  (2,0)  (2,3)  (2,6)  (4,0)  (4,3)  (4,6)  ";
  string actual = "";
  for(int x = 0; x < 3; ++x) {
    for(int y = 0; y < 3; ++y) {
      Coordinate c(x*2, y*3);
      actual += '(' + to_string(c.get_x()) + ',' + to_string(c.get_y()) + ")  ";
    }
  }

  if (expected != actual) {
    cerr << "fail: coordinate.h: static initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  return 0;
}
```

Define the expected output so it can be compared. (The operations and expected outputs together are commonly called the *test vectors*.)

If it fails, report “fail” and a concise description of why, and return -1.

If it works, return 0 silently. This supports a large body of tests without “chattering”.

Step 1 Next Test – Result of Test

Edit coordinate.cpp to make it fail (“testing the test”)

```
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ vi coordinate.cpp
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ make
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_coordinate.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ ./a.out
fail
expected: "(0,0)  (0,3)  (0,6)  (2,0)  (2,3)  (2,6)  (4,0)  (4,3)  (4,6)  "
actual:   "(1,0)  (1,3)  (1,6)  (3,0)  (3,3)  (3,6)  (5,0)  (5,3)  (5,6)  "
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ vi coordinate.cpp
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ make
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_coordinate.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/01-basic_coordinate$
```

Test fails as expected

Remove the error

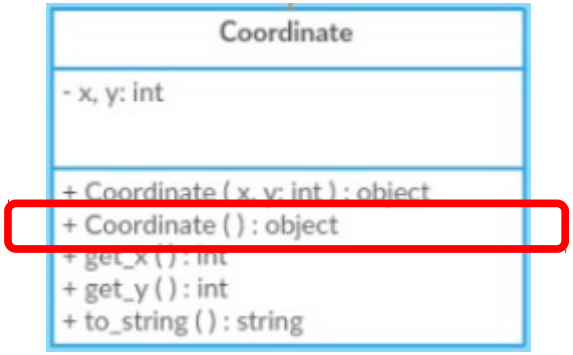
Silence is golden
Pass!

```
ricegf@pluto:~/dev/cpp/201701/06$ git add Makefile coordinate.cpp coordinate.h test_coordinate.cpp
ricegf@pluto:~/dev/cpp/201701/06$ git commit -m "One coordinate constructor"
[master (root-commit) 112c081] One coordinate constructor
 4 files changed, 83 insertions(+)
 create mode 100644 Makefile
 create mode 100644 coordinate.cpp
 create mode 100644 coordinate.h
 create mode 100644 test_coordinate.cpp
ricegf@pluto:~/dev/cpp/201701/06$
```

Commit the first step to git

Step 2: 2nd Constructor

- A 2nd constructor is added that presets the name and assigns a random coordinate
  - We define the maximum (x, y) in a header file
- Additional test is added for the new constructor



globals.h

```
const int MAX_X = 9;
const int MAX_Y = 9;
```

globals.h (included in coordinate.h)

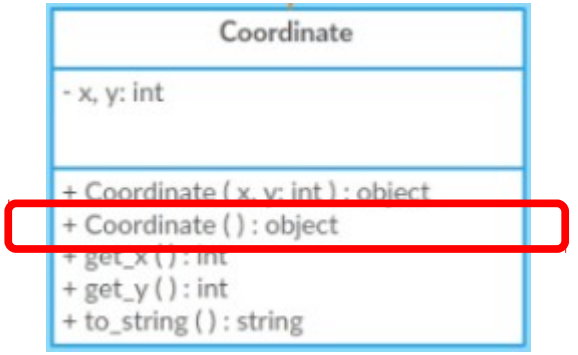
coordinate.h

```
#ifndef COORDINATE_H
#define COORDINATE_H 2017
#include "globals.h"

class Coordinate {
  public:
    Coordinate() : Coordinate(rand() % MAX_X, rand() % MAX_Y) { }
    Coordinate(int x, int y) : _x{x}, _y{y} { }
    int get_x();
    int get_y();
  private:
    int _x;
    int _y;
};
#endif
```

The 2nd constructor uses “constructor chaining” to assign random coordinates

Constructor Chaining

- Constructor Chaining is the calling of one constructor by another, to avoid duplicating code and maintenance




```
class Coordinate {
  public:
    Coordinate() : Coordinate(rand() % MAX_X, rand() % MAX_Y) { }
    Coordinate(int x, int y) : _x{x}, _y{y} { }
    int get_x();
    int get_y();
  private:
    int _x;
    int _y;
};
```

Step 2 – Regression Test

```
  // ^^^ static test from Step 1 goes here

  srand(0xbadcafe);
  expected = "(20,26)  (10,3)  (22,21)  (7,19)  (5,17)  (2,13)  (12,24)  (18,18)  (22,12)  ";
  actual = "";
  for(int i = 0; i < 9; ++i) {
    Coordinate c;
    actual += '(' + to_string(c.get_x()) + ',' + to_string(c.get_y()) + ")  ";
  }

  if (expected != actual) {
    cerr << "fail: coordinate.h: random initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  return 0;
}
```

Predictably initialize the psuedo-random number generator

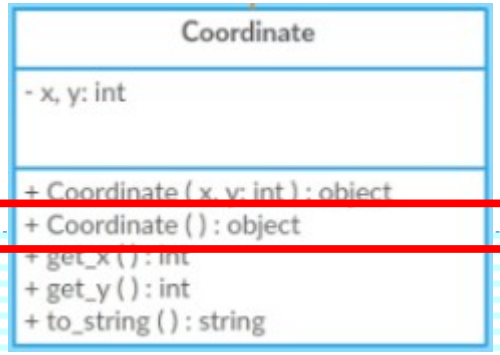
"expected" is determined experimentally

Generate a *random* coordinate

test_coordinate.cpp

Step 2: Updated Makefile

- Add globals.h dependency




```
# Makefile for Roving Robots
CXXFLAGS = -std=c++11
PREV = 01-basic_coordinate

all: executable

debug: CXXFLAGS += -g
debug: executable

rebuild: clean executable
executable: test_coordinate.o coordinate.o
    $(CXX) $(CXXFLAGS) test_coordinate.o coordinate.o
test_coordinate.o: test_coordinate.cpp coordinate.h
    $(CXX) $(CXXFLAGS) -c test_coordinate.cpp
coordinate.o: coordinate.cpp coordinate.h globals.h
    $(CXX) $(CXXFLAGS) -c coordinate.cpp
clean:
    rm -f *.o a.out
```

```
ricegf@pluto:~/dev/cpp/201701/06/02-add_random_coordinate$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/02-add_random_coordinate$ make
g++ -std=c++11 -c test_coordinate.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_coordinate.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/02-add_random_coordinate$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/02-add_random_coordinate$
```

Makefile

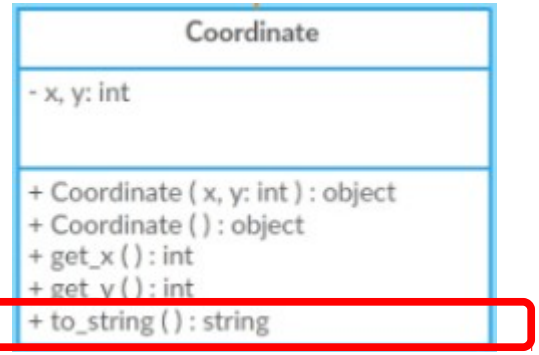
...And Commit to Git

git add -u adds any updated files already in the repository, but does not add any new files. Quick and easy!

```
ricegf@pluto:~/dev/cpp/201701/06$ git add -u
ricegf@pluto:~/dev/cpp/201701/06$ git commit -m "Add 2nd coordinate constructor"
[master cb384f3] Add 2nd coordinate constructor
 4 files changed, 26 insertions(+), 3 deletions(-)
ricegf@pluto:~/dev/cpp/201701/06$
```

Step 3: Add to_string method

- Fairly simple, though this illustrates a scoping problem with to_string
  - C++ 11 added a function called to_string
  - We must distinguish ours from theirs
- Update test to use to_string instead of brute force




```
#ifndef COORDINATE_H
#define COORDINATE_H 2017
#include "globals.h"
#include <stdlib.h>
#include <string>
using namespace std;

class Coordinate {
  public:
    Coordinate() : Coordinate(rand() % MAX_X, rand() % MAX_Y) { }
    Coordinate(int x, int y) : _x{x}, _y{y} { }
    int get_x();
    int get_y();
    string to_string();
  private:
    int _x;
    int _y;
};
#endif
```

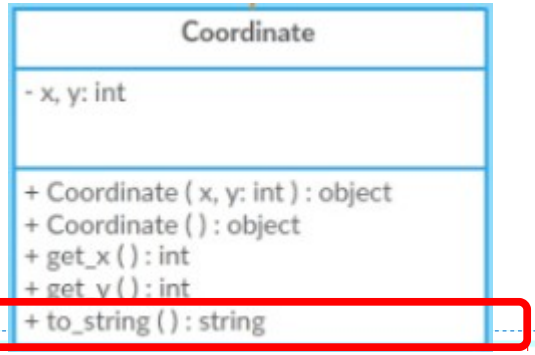
Need the string library, of course

Declare the to_string method

coordinate.h

Step 3: Add to_string method

- In defining our to_string method, C++ is confused
  - We're defining to_string() in local scope
  - So it looks for to_string(int) in local scope, too!



coordinate.cpp

```
#include "coordinate.h"

int Coordinate::get_x( ) {return _x;}
int Coordinate::get_y( ) {return _y;}
string Coordinate::to_string( ) {return '(' + to_string(_x) + ',' + to_string(_y) + ")";}
```

The method or the function?

```
ricegf@pluto:~/dev/cpp/201701/06/03-add_to_string$ g++ -std=c++11 -c coordinate_bad.cpp
coordinate_bad.cpp: In member function ‘std::string Coordinate::to_string()’:
coordinate_bad.cpp:13:59: error: no matching function for call to ‘Coordinate::to_string(int&)’
 string Coordinate::to_string( ) {return '(' + to_string(_x) + ',' + to_string(_y) + ")";}
                                                           ^
coordinate_bad.cpp:13:59: note: candidate is:
coordinate_bad.cpp:13:8: note: std::string Coordinate::to_string()
 string Coordinate::to_string( ) {return '(' + to_string(_x) + ',' + to_string(_y) + ")";}
        ^
coordinate_bad.cpp:13:8: note:   candidate expects 0 arguments, 1 provided
```

C++ chose... poorly

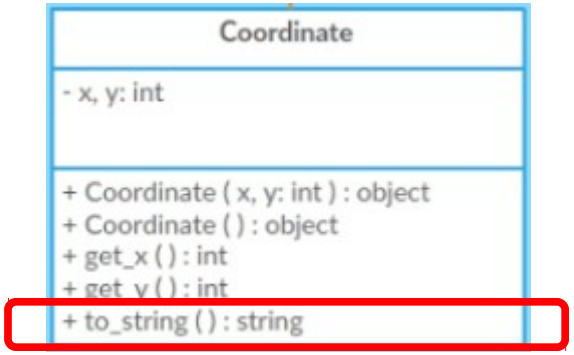
Image ©1989 by Paramount Pictures. Educational fair use is asserted.

Step 3: Add to_string method

- So we explicitly specify std::to_string instead

coordinate.cpp

```
#include "coordinate.h"

int Coordinate::get_x( ) {return _x;}
int Coordinate::get_y( ) {return _y;}
string Coordinate::to_string( ) {
  return '(' + std::to_string(_x) + ','
             + std::to_string(_y) + ")";
}
```



- test_coordinate.cpp certainly benefits

test_coordinate.cpp

Before

```
for(int i = 0; i < 9; ++i) {
  Coordinate c;
  actual += '(' + to_string(c.get_x()) + ',' + to_string(c.get_y()) + ")  ";
}
```

After

```
for(int i = 0; i < 9; ++i) {
  Coordinate c;
  actual += c.to_string() + "  ";
}
```

```
ricegf@pluto:~/dev/cpp/201701/06/03-add_to_string$ make
g++ -std=c++11 -c test_coordinate.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_coordinate.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/03-add_to_string$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/03-add_to_string$
```

Invariants

- Wait – how do we change x or y in a Coordinate object?
  - We don't! If a Robot moves, we'll simply create a new Coordinate object to represent its position
- An object with fields and internal state that cannot be changed is an example of an “invariant”
  - Invariants are usually simpler to write
  - Invariants, when used appropriately, tend to reduce the number and severity of bugs

Invariant – Code for which specified assertions are guaranteed to be true

Asserting Invariance

- If you #include <cassert> (or the C version, <assert.h>), you get an “assert” function that accepts a Boolean
  - If true, no action is taken
  - If false, an environment-consistent error is printed
- For example, we could redefine our constructor as

coordinate.h

```
Coordinate(int x, int y) : _x{x}, _y{y} {
  assert(x >= 0 && x < MAX_X);
  assert(y >= 0 && y < MAX_Y);
}
```

Attempting to create an invalid Coordinate object then causes a standard error message to be dumped to STDERR:

```
a.out: coordinate_with_assert.cpp:18: void Coordinate::init(int, int):
Assertion `x >= 0 && x < MAX_X' failed.
Aborted (core dumped)
```

In bash, use ' ulimit -c unlimited' to *actually* dump a core
ddd can then help you to debug from a core

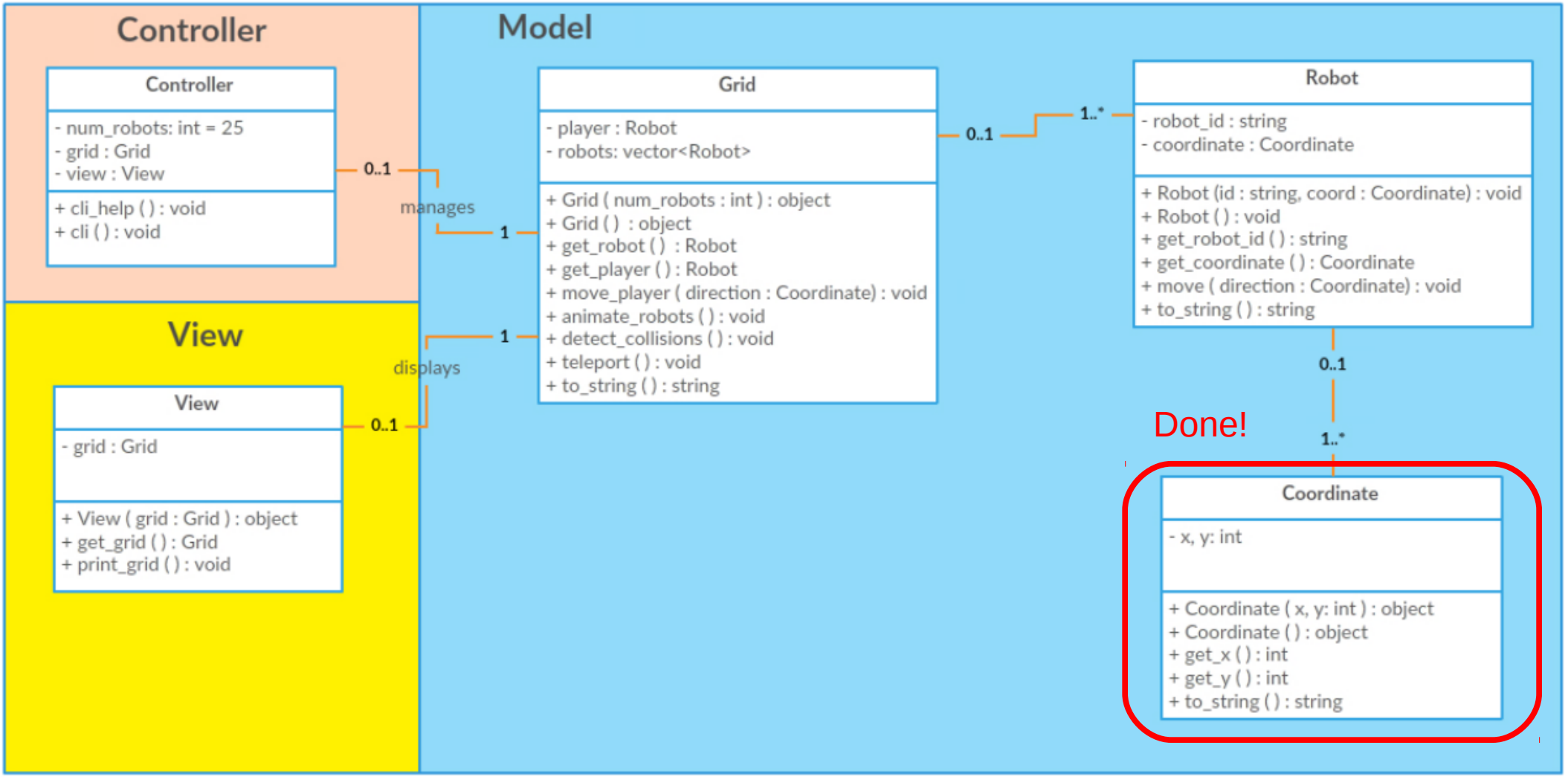
But do we WANT to do this?

- An easy way to specify a direction is by a *delta* coordinate
  - (-1,0) is left, (0,-1) is up – but these aren't *valid* coordinates
  - Thus, we can conveniently create a new coordinate by adding the x and the y values (respectively) of an existing coordinate and a delta coordinate
- Eventually we'll get into *inheritance*, where we can declare this:

```
class valid_coordinate : public coordinate {
  // add assertions into the constructor here
}
```

  - This creates a class called “valid_coordinate” that will be *exactly like* “coordinate” *except* guaranteed to be on the grid
- Previews of coming attractions... ;-)
- Until then, we'll avoid the assertion and use a common class

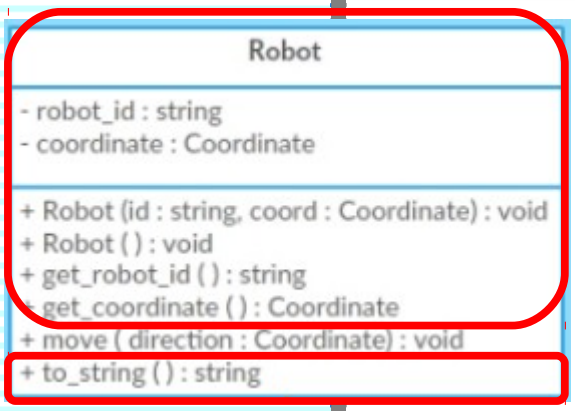
Where We Stand



Step 4: On to Robots.h!

robot.h

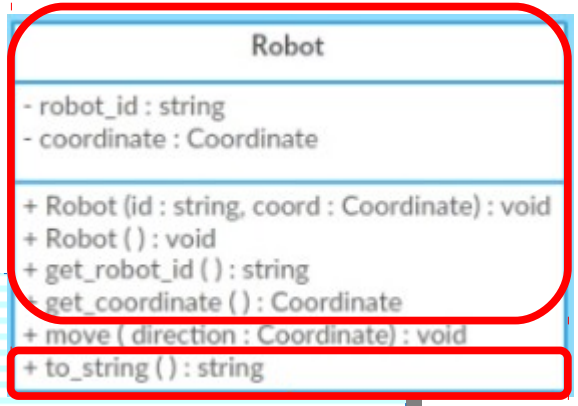
```
#ifndef ROBOT
#define ROBOT 2016

#include "globals.h"
#include "coordinate.h"
#include <string>

using namespace std;

class Robot {
  public:
    Robot(string robot_id, Coordinate coordinate);
    Robot();

    string get_robot_id( );
    Coordinate get_coordinate( );
    string to_string( );

  private:
    string _robot_id; // "Name" of the robot to display on-screen
    Coordinate _coordinate;  // location of robot on an X-Y grid

    int generateID() {       // generate unique ID for each robot
      static int s_robot_id = 0;
      return ++s_robot_id;
    }
};
#endif
```



A “static” variable's value is retained between calls to the enclosing method. So every time generateID() is called, it returns the next higher integer – 1, 2, 3, 4, ...

Step 4: Implementation (.cpp)

- Our to_string method for Robot will be
  - The name of the robot
  - Its coordinate on the grid



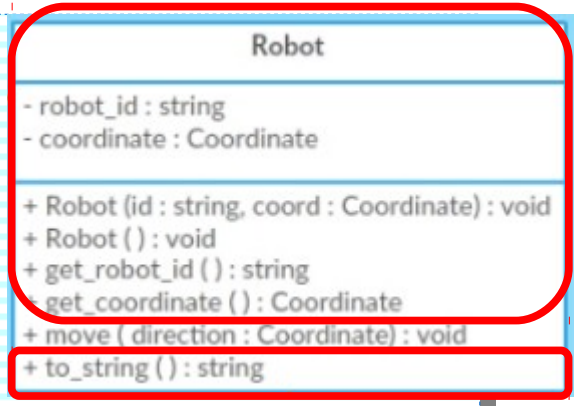
robot.cpp

```
#include "robot.h"
#include "globals.h"
#include "coordinate.h"

using namespace std;

Robot::Robot(string robot_id, Coordinate coordinate) :
      _robot_id{robot_id}, _coordinate{coordinate} { }
Robot:: Robot() : Robot("Robot"+std::to_string(generateID()), Coordinate()) { }

string Robot::get_robot_id( ) {return _robot_id;}
Coordinate Robot::get_coordinate( ) {return _coordinate;}
string Robot::to_string() {return _robot_id + ' ' +  _coordinate.to_string();}
```

Step 4: Don't Forget to Test!

test_robot.cpp

```
#include "globals.h"
#include "robot.h"
#include "coordinate.h"
#include <iostream>

int main() {
  string expected = "Robot00 (0,0)  Robot02 (0,2)
                     Robot20 (2,0)  Robot22 (2,2)  ";
  string actual = "";
  for(int x = 0; x < 3; x += 2) {
    for(int y = 0; y < 3; y += 2) {
      Robot r("Robot" + std::to_string(x) + std::to_string(y), Coordinate(x, y));
      actual += r.to_string( ) + "  ";
    }
  }

  if (expected != actual) {
    cerr << "fail: robot.h: static initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  // continued on next page
```

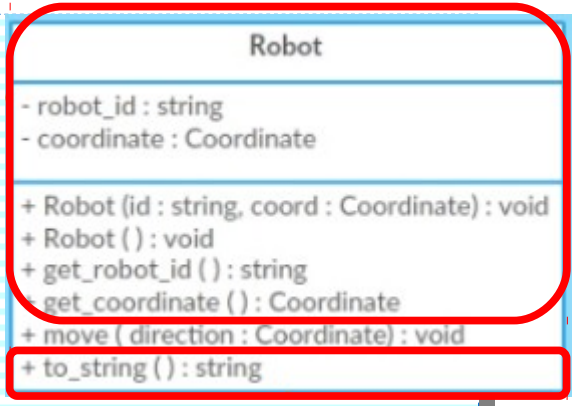


This should look very similar to testing coordinates.
Code similarity is very common in programming.
Look long enough and you'll detect *patterns*...

Step 4: Don't Forget to Test...

test_robot.cpp

```
  // continued from previous page

  srand(0xbadcafe);
  expected = "Robot1 (5,6)  Robot2 (6,5)  Robot3 (8,7)
              Robot4 (4,6)  Robot5 (4,5)  ";
  actual = "";
  for(int i = 0; i < 5; ++i) {
    Robot r;
    actual += r.to_string( ) + "  ";
  }

  if (expected != actual) {
    cerr << "fail: robot.h: random initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  return 0;
}
```




```
ricegf@pluto:~/dev/cpp/201701/06/04-add_robot$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/04-add_robot$ make
g++ -std=c++11   -c -o test_robot.o test_robot.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_robot.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/04-add_robot$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/04-add_robot$
```

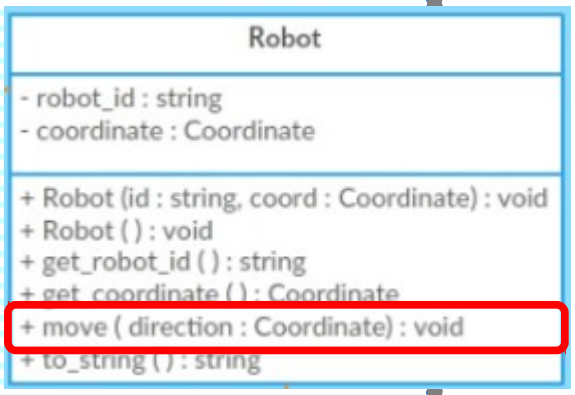
...and Commit to git

```
ricegf@pluto:~/dev/cpp/201701/06$ git add coordinate.* globals.h Makefile robot.* test_*
ricegf@pluto:~/dev/cpp/201701/06$ git commit -m 'First version of Robot'
[master f5e0334] First version of Robot
 8 files changed, 162 insertions(+), 56 deletions(-)
 create mode 100644 globals.h
 create mode 100644 robot.cpp
 create mode 100644 robot.h
 rewrite test_coordinate.cpp (80%)
 create mode 100644 test_robot.cpp
ricegf@pluto:~/dev/cpp/201701/06$
```

Step 5: Finishing Robots.cpp

robot.h

```
#ifndef ROBOT
#define ROBOT 2016
#include "globals.h"
#include "coordinate.h"
#include <string>
using namespace std;

class Robot {
  public:
    Robot(string robot_id, Coordinate coordinate) :
       _robot_id{robot_id}, _coordinate{coordinate} { }
    Robot() : Robot("Robot"+std::to_string(generateID()), Coordinate()) { }

    string get_robot_id( );
    Coordinate get_coordinate( );
    string to_string( );
    void move(Coordinate direction);

  private:
    string _robot_id; // "Name" of the robot to display on-screen
    Coordinate _coordinate;  // location of robot on an X-Y grid

    int generateID() {       // generate unique ID for each robot
      static int s_robot_id = 0;
      return ++s_robot_id;
    }
};
#endif
```

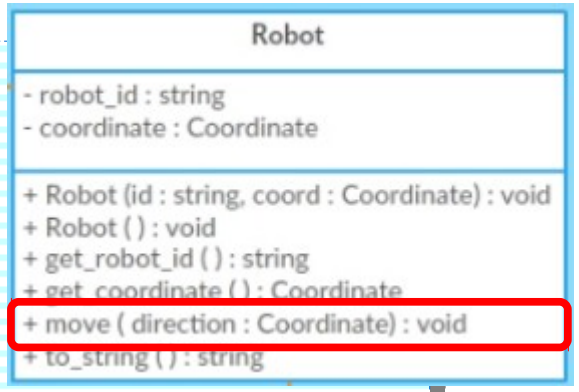


Not a major change...

Step 5: Finishing Robots.cpp

robot.cpp

```
#include "robot.h"
#include "globals.h"
#include "coordinate.h"

using namespace std;

string Robot::get_robot_id( ) {return _robot_id;}
Coordinate Robot::get_coordinate( ) {return _coordinate;}
string Robot::to_string() {return _robot_id + ' ' +  _coordinate.to_string();}
void Robot::move(Coordinate direction) {
    _coordinate = Coordinate{_coordinate.get_x() + direction.get_x(),
                             _coordinate.get_y() + direction.get_y()};
}
```



Not a major change, either...

Again, note the invariance of the Coordinate class. To move an instance of the Robot class, we instance a NEW coordinate rather than change the existing one. This makes Coordinate more supportable and testable.

Step 5: Testing Robots.cpp

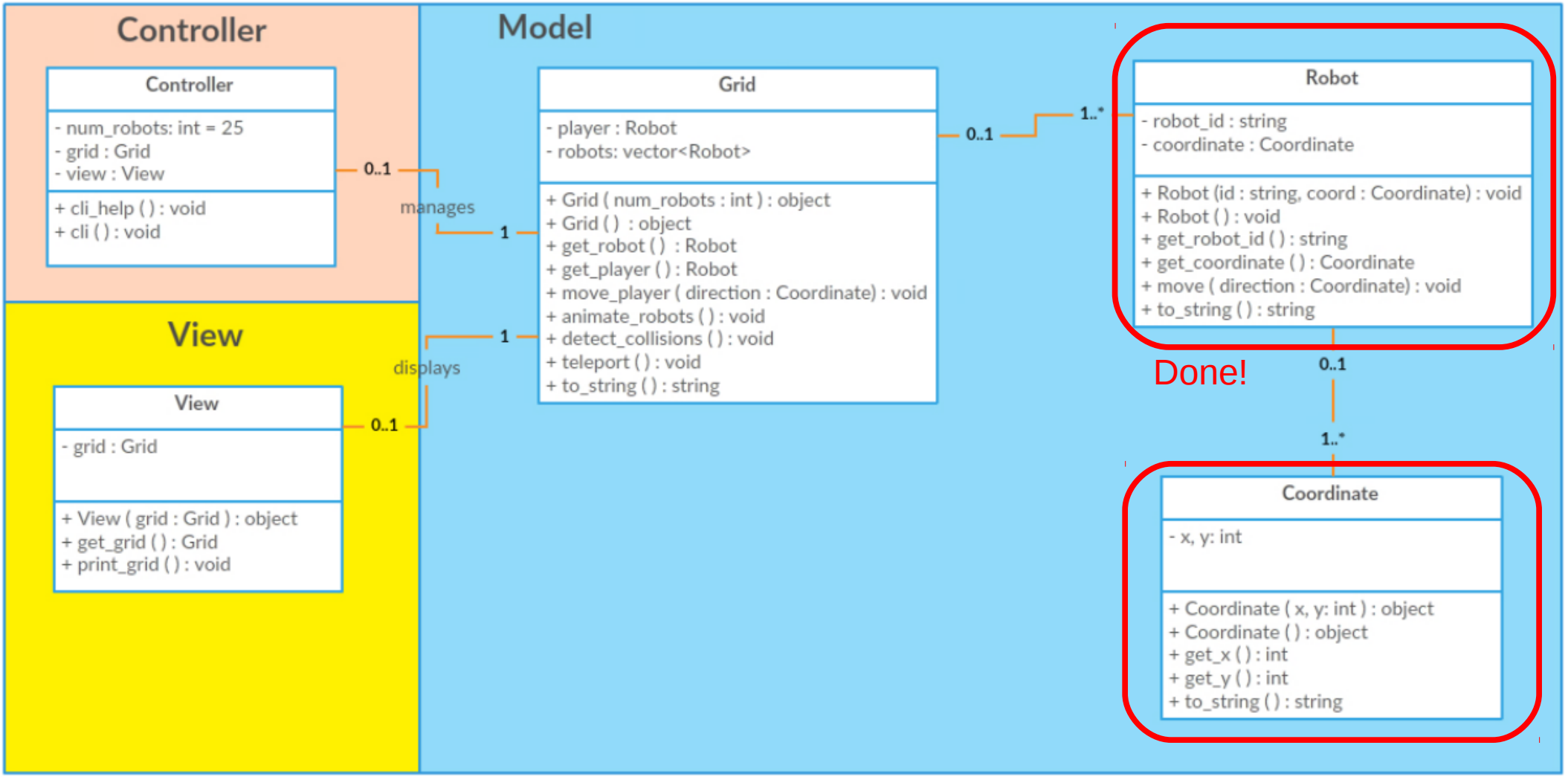
test_robot.cpp

```
  // ... added to previous step

  expected = "Marvin (5,5) Marvin (2,5) Marvin (2,3) Marvin (32,3) Marvin (29,8) ";
  actual = "";
  Robot r("Marvin", Coordinate(5, 5));  // "Marvin" starts at (5, 5)
  actual += r.to_string( ) + "  ";
  r.move(Coordinate(-3,0)); // Move "Marvin" left 3 spaces, to (2, 5)
  actual += r.to_string( ) + "  ";
  r.move(Coordinate(0,-2)); // Move "Marvin" up 2 spaces, to (2, 3)
  actual += r.to_string( ) + "  ";
  r.move(Coordinate(30,0)); // Move "Marvin" right 30 spaces, to (32, 3)
  actual += r.to_string( ) + "  ";
  r.move(Coordinate(-3,5)); // Move "Marvin" down and left, to (29, 8)
  actual += r.to_string( ) + "  ";

  if (actual != actual) {
    cerr << "fail: robot.h: movement" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  return 0;
}
```

```
ricegf@pluto:~/dev/cpp/201701/06/05-move_robot$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/05-move_robot$ make
g++ -std=c++11   -c -o test_robot.o test_robot.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_robot.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/05-move_robot$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/05-move_robot$
```

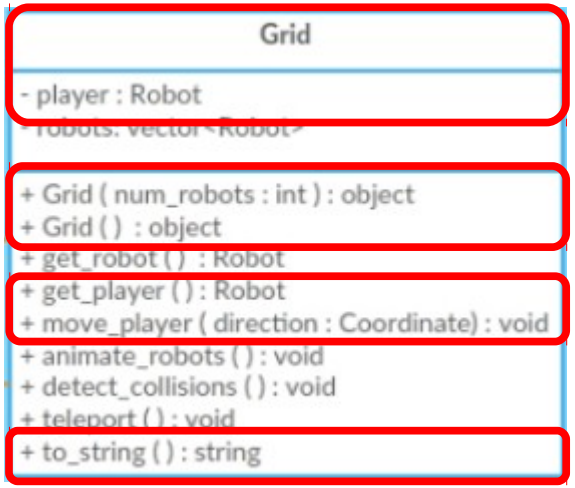
No change needed to Makefile
Commit to git in the usual way

Where We Stand



Step 6: On to Grid!

- Handling a lot of robots looks hard
  - Let's start by moving a *single robot* ("player") around on a visible grid
  - This is an example of a sprint
    - Take small steps toward a larger goal



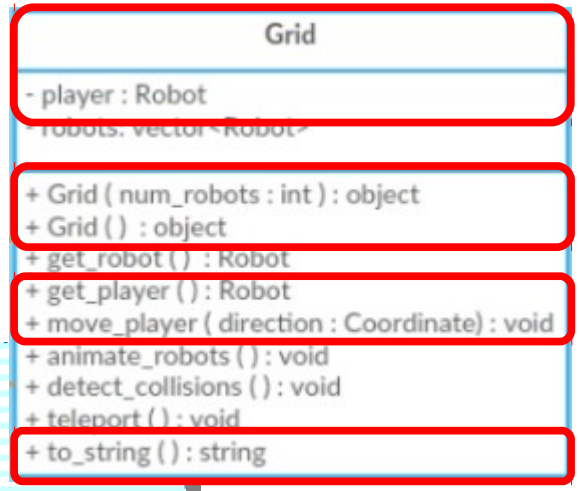
grid.h

```
#ifndef GRID
#define  GRID 2017
#include "robot.h"

class Grid {
  public:
    Grid(Robot player) : _player{player} { }
    Grid() : Grid(Robot()) { }
    Robot get_player( );
    void move_player(Coordinate direction);
    string to_string();

  private:
    Robot _player; // The robot wandering the grid
};
#endif
```

Technically, we could use grid.get_player().move(...), but it's better form to provide interfaces at the class itself.

Step 6: The Important to_string

- Since we don't have many string handling tools yet, we'll just brute force the grid
  - Add a "." to every grid position *except* the one where the Robot is

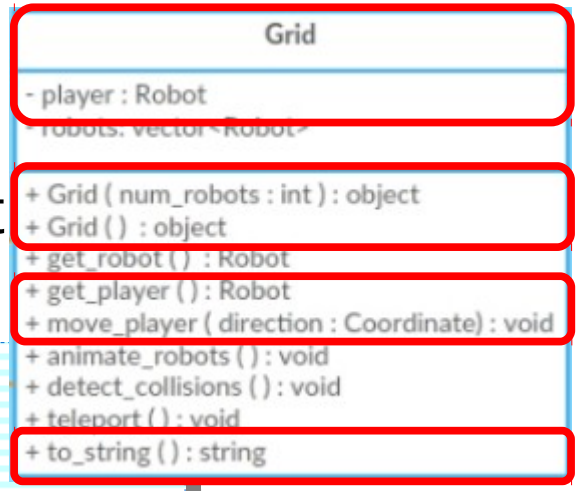


grid.cpp

```
#include "grid.h"
#include "globals.h"
#include "robot.h"

Robot Grid::get_player( ) {return _player;}
void Grid::move_player(Coordinate direction) {
    _player.move(direction);
}
string Grid::to_string() {
    string grid = "";
    for (int x = 0; x < MAX_X; ++x) {
      for (int y = 0; y < MAX_Y; ++y) {
        grid += (x == _player.get_coordinate().get_x() &&
                 y == _player.get_coordinate().get_y()) ? "R" : ".";
      }
      grid += '\n';
    }
    return grid;
}
```

Step 6: Viewing the Grid

- Now that we have a grid...
  What does it look like?
- Need a “visual test” or “User Interface (UI)” test

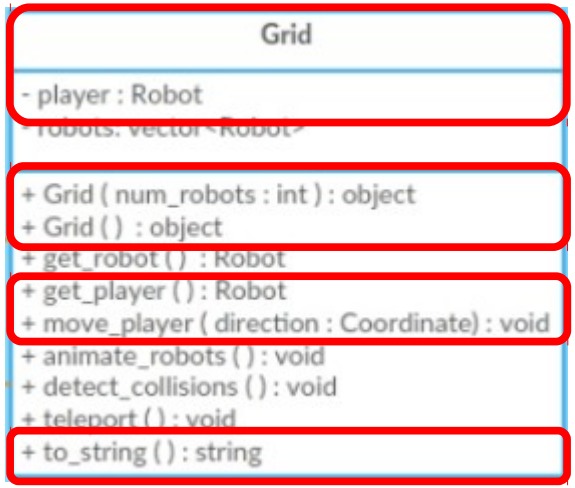


show_grid.cpp

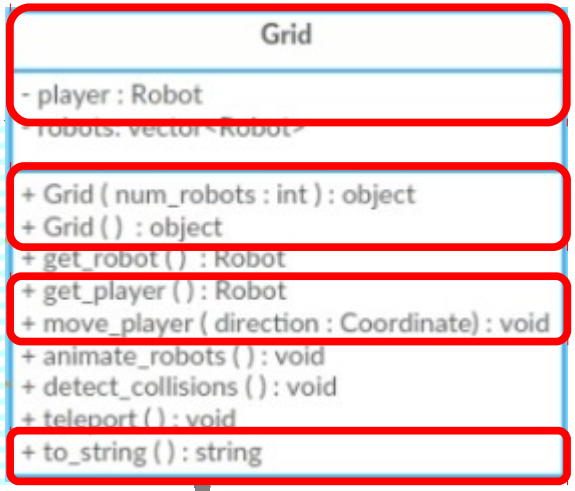
```
#include "globals.h"
#include "robot.h"
#include "grid.h"
#include <string>
#include <iostream>

using namespace std;

int main() {
  cout << "Grid at (5, 5)" << endl << endl;
  Grid grid(Robot("Marvin", Coordinate(5, 5)));
  cout << grid.to_string( ) << endl;

  cout << "Grid at random location" << endl << endl;
  grid = Grid{};
  cout << grid.to_string( ) << endl;

}
```

Step 6: Viewing the Grid

```
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$ make showgrid
g++ -std=c++11   -c -o show_grid.o show_grid.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 show_grid.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$ ./a.out
Grid at (5, 5)

.........
.........
.........
.........
.........
.....R...
.........
.........
.........

Grid at random location

.........
.........
.........
.........
.........
.........
.........
.R.......
.........

ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$
```



Step 6: Testing the Grid



test_grid.cpp

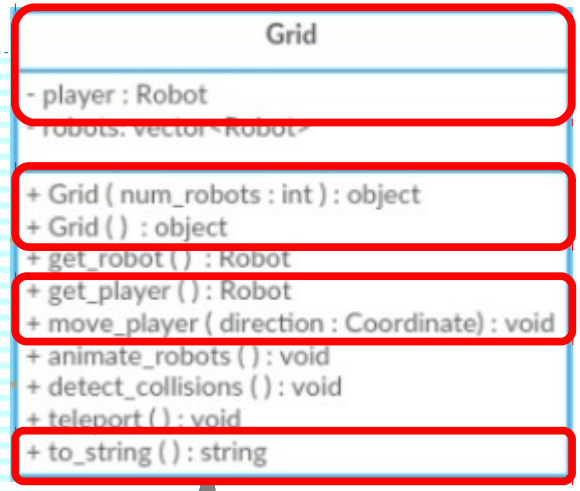
```
#include "globals.h"
#include "robot.h"
#include "grid.h"
#include <string>
#include <iostream>
using namespace std;

int main() {
  string expected =
".........\n.........\n.........\n.........\n.........\n.....R...\n...
.....\n.........\n.........\n";
  string actual = "";
  Grid grid(Robot("Marvin", Coordinate(5, 5)));
  actual += grid.to_string( );

  if (expected != actual) {
    cerr << "fail: grid.h: static initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }

  // Continued on next page
```

Step 6: Testing the Grid

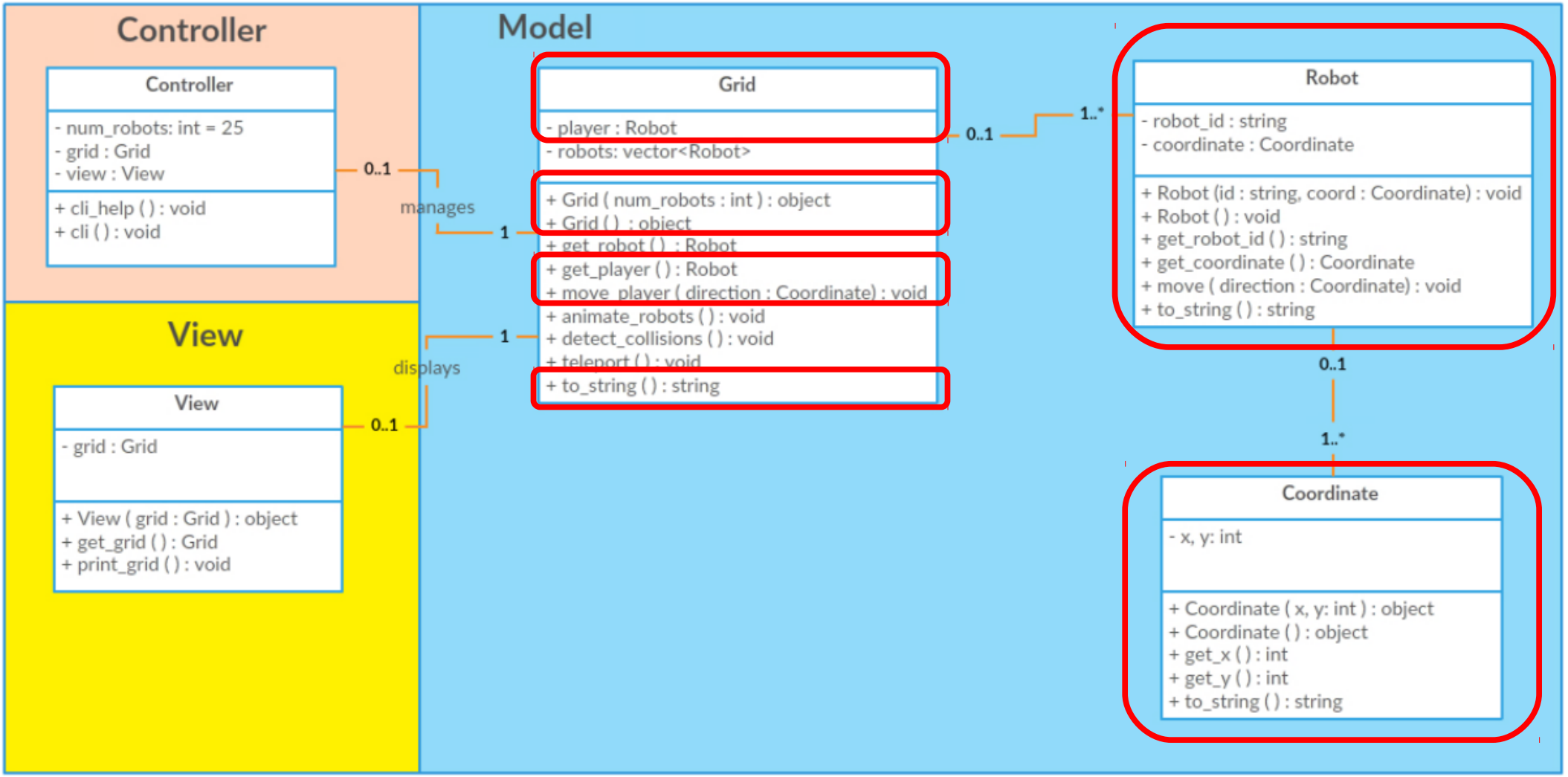
test_grid.cpp

```
  // Continued from previous page

  srand(0xbadcafe);
  expected =
".........\n.........\n.........\n.........\n.........\n
......R..\n.........\n.........\n.........\n";
  actual = "";
  grid = Grid{};
  actual += grid.to_string( );

  if (expected != actual) {
    cerr << "fail: grid.h: random initialization" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }
}
```

```
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$ make rebuild
rm -f *.o a.out
g++ -std=c++11   -c -o test_grid.o test_grid.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_grid.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/06-add_grid$
```

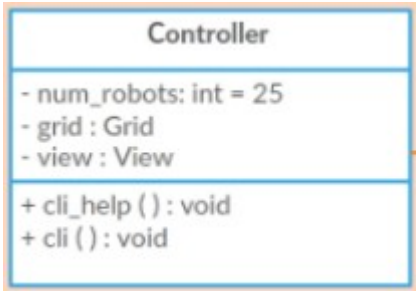


Where We Stand



Step 7: Skipping to Controller

- At this point we elect to implement Controller (and then View), and come back afterward to add support for multiple robots
  - This will enable interactive testing
  - Sprints are not planned around implementing full *classes* from the class diagram, but around implementing customer-defined *features* of value
    - And moving the player robot around would be more useful than seeing a lot of non-player robots sitting on the grid

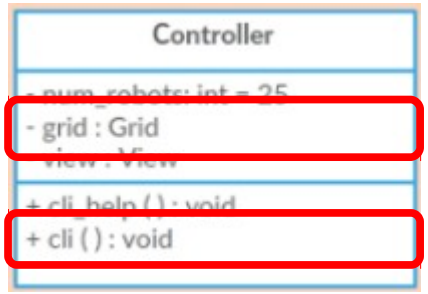


Step 7: Skipping to Controller...

- The interface for Controller is straightforward

controller.h

```
#include "grid.h"
#include "robot.h"

class Controller {
  public:
    void cli();
    void execute_cmd(string cmd);

  private:
    Robot _player;
    Grid grid{_player};

};
```



cli() gets a command from the user and passes it to execute_cmd(), which implements it. This division makes it easy to regression test commands, and (in some cases) to add gui(), touch(), and / or web() front ends to the program efficiently.

Main is then almost trivial – instance the controller and invoke the CLI.

Coding the CLI as part of the Controller class makes it easy to manage multiple CLIs without changing the code, and is good OO practice.

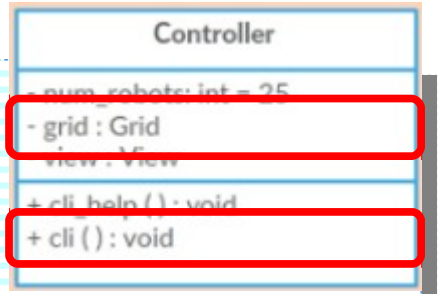
main.cpp

```
#include "controller.h"

using namespace std;

int main() {
  Controller controller;
  controller.cli();
}
```

Note that a *trivial* main isn't usually shown on the class diagram, though it could be.

Step 7: Controller::CLI Implementation

- The CLI method for Controller is also straightforward



controller.cpp

```
#include "controller.h"
#include "globals.h"
#include "grid.h"
#include <iostream>

void Controller::cli() {
    cout << "============" << endl
         << "ROVING ROBOT" << endl
         << "============" << endl << endl;

    string cmd = "x";
    while (cmd[0] != '0') {
        cout << grid.get_player().get_coordinate().to_string() << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
        cout << grid.to_string() << endl;
    }
};
```

Display a title banner

'0' is the exit command

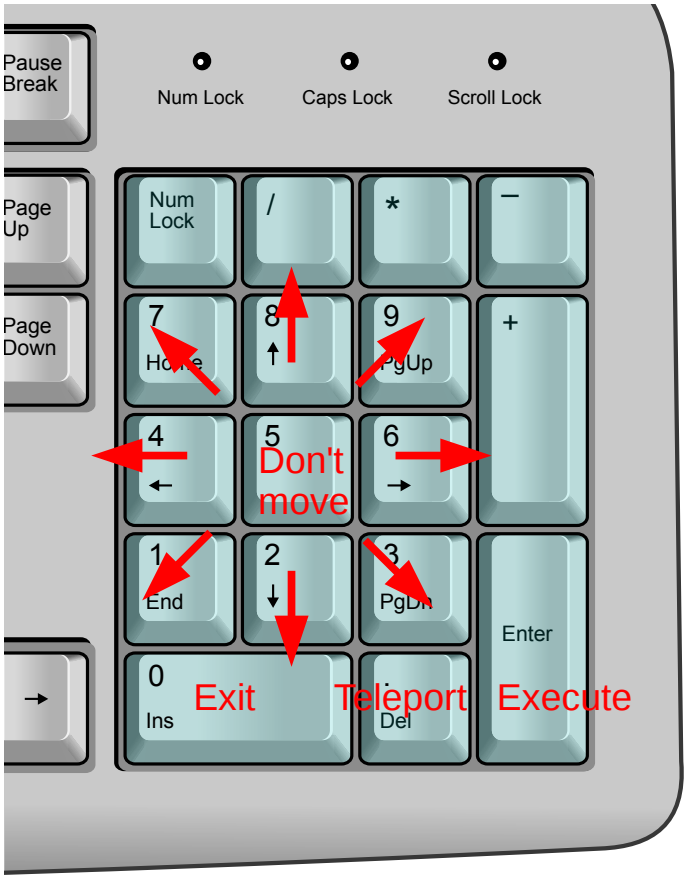
Display the player coordinates and a prompt, then get a command for execute_cmd

Show the grid

So... what commands do we need to execute?

Step 7: Robot Controller

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



Rather than press Execute (Enter) after each command, it's more intuitive for the command key itself to execute the command, e.g., pressing 3 would *immediately* move the player's robot down and to the right.

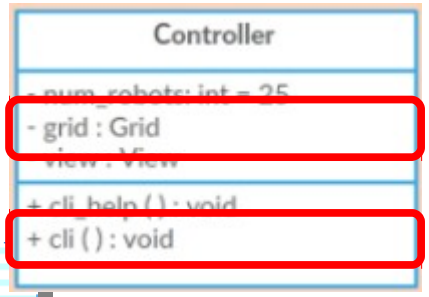
This is difficult to do portably, however, because you must work around each operating system's layers of keyboard management software.

So for purposes of this demo, we punt and make do with the Execute key.

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

Step 7: Controller::execute_cmd Implementation

- Execute_cmd initially just implements moving by changing the coordinate attribute of the player via the grid's move_player method

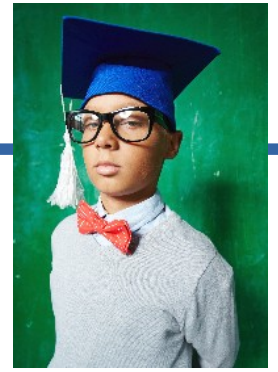


controller.cpp

```
void Controller::execute_cmd(string cmd) {
    if (cmd[0] == '1') { grid.move_player(Coordinate(-1, 1)); }
    if (cmd[0] == '2') { grid.move_player(Coordinate( 0, 1)); }
    if (cmd[0] == '3') { grid.move_player(Coordinate( 1, 1)); }
    if (cmd[0] == '4') { grid.move_player(Coordinate(-1, 0)); }
    if (cmd[0] == '5') { grid.move_player(Coordinate( 0, 0)); }
    if (cmd[0] == '6') { grid.move_player(Coordinate( 1, 0)); }
    if (cmd[0] == '7') { grid.move_player(Coordinate(-1,-1)); }
    if (cmd[0] == '8') { grid.move_player(Coordinate( 0,-1)); }
    if (cmd[0] == '9') { grid.move_player(Coordinate( 1,-1)); }
};
```

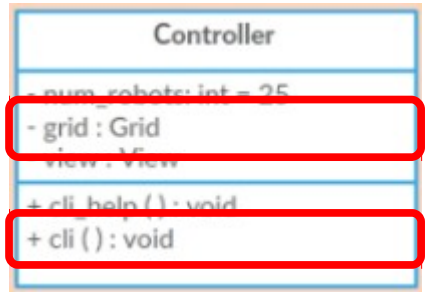
Challenge

What other C++ construct(s) could be used to execute commands?



Step 7: And it works!
(Sort of...)

```
ricegf@pluto:~/dev/cpp/201701/06/07-add_controller$ make rebuild
rm -f *.o a.out
g++ -std=c++11 -c main.cpp
g++ -std=c++11 -c controller.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 main.o controller.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/07-add_controller$ ./a.out
============
ROVING ROBOT
============

(7,1) Command? 4
.........
.........
.........
.........
.........
.........
.R.......
.........
.........

(6,1) Command? 4
.........
.........
.........
.........
.........
.R.......
.........
.........
.........

(5,1) Command?
```



Wait a minute...

'4' should move our robot left, but it moves up instead!

Where's the bug?

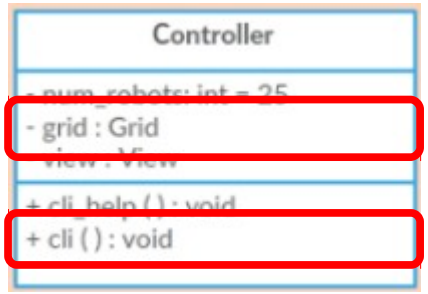
Why did we not detect his bug earlier in our testing?

(For those of you following on-line, this is left as an exercise for the student...)

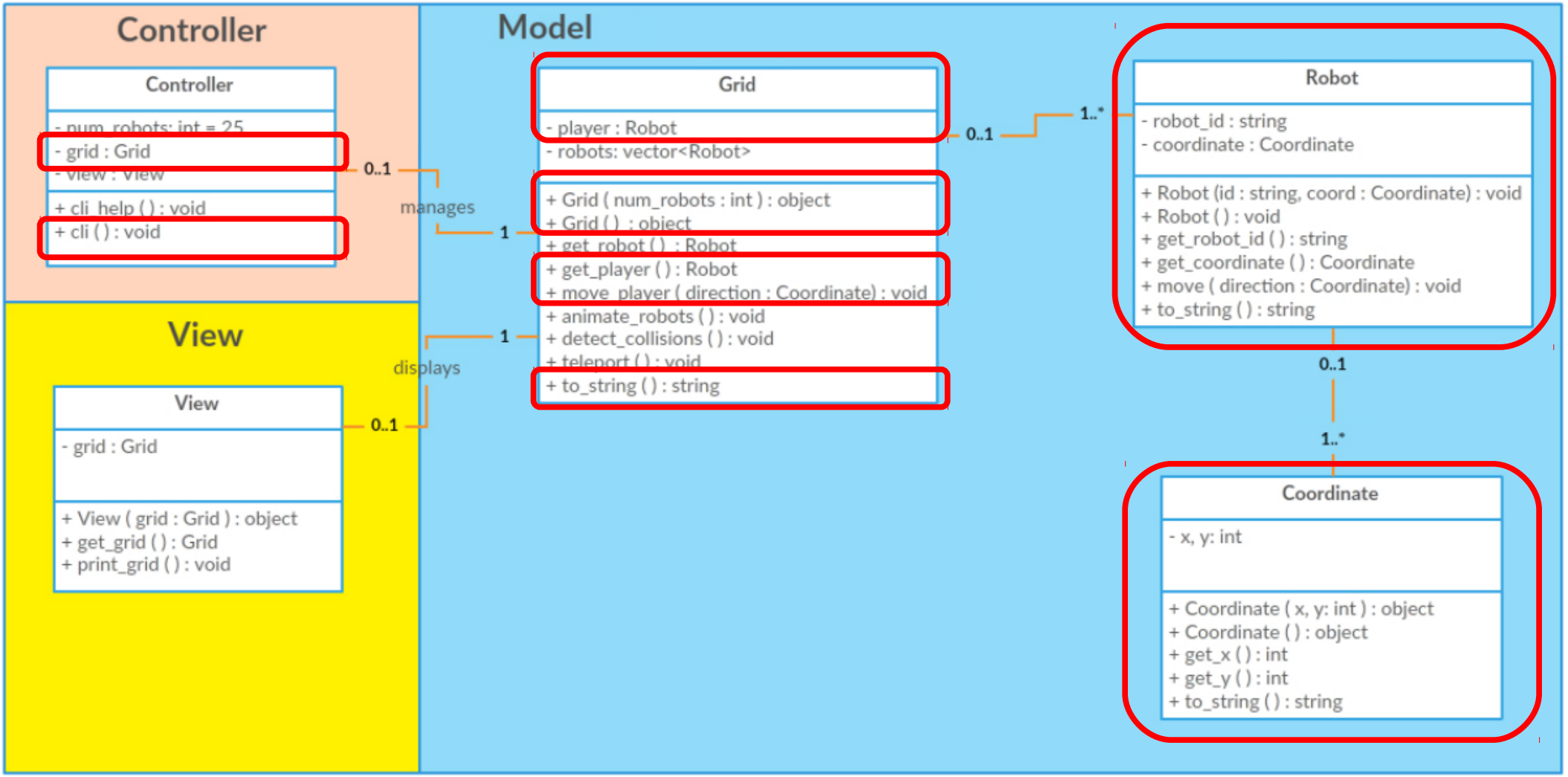
Does our CLI handle other non-command inputs well?

Step 7: And (NOW!) it works

```
ricegf@pluto:~/dev/cpp/201701/06/07-add_controller$ make rebuild
rm -f *.o a.out
g++ -std=c++11 -c main.cpp
g++ -std=c++11 -c controller.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 main.o controller.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/07-add_controller$ ./a.out
============
ROVING ROBOT
============

(7,1) Command? 4
.........
......R..
.........
.........
.........
.........
.........
.........
.........
.........

(6,1) Command? 4
.........
.....R...
.........
.........
.........
.........
.........
.........
.........
.........

(5,1) Command?
```

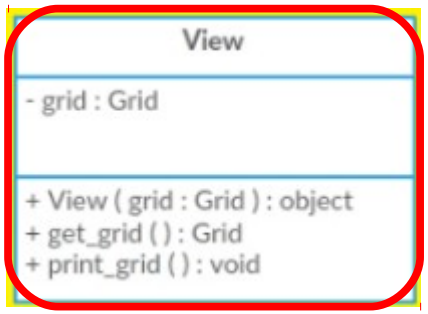


Where We Stand



Step 8: Implement View

- View centralizes our major output into a single class (or in bigger programs, a package of classes)
- The View class is quite simple, since we have Grid's to_string() method on which to rely

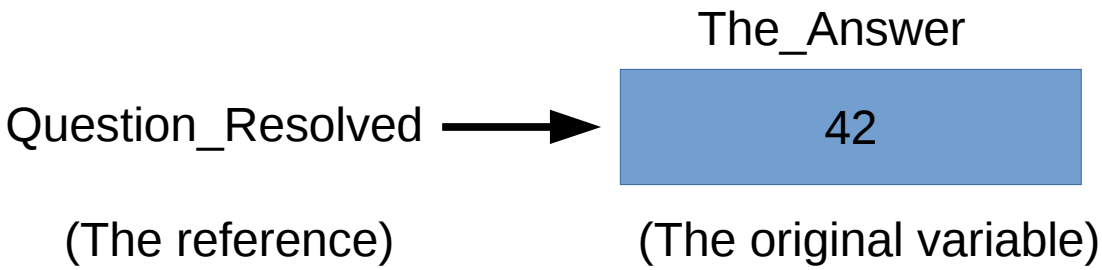



```
#ifndef VIEW
#define  VIEW 2017

#include "grid.h"

class View {
  public:
    View(Grid& g) : grid{g} {};

    void print_banner();
    Grid get_grid();
    void print_grid();

  private:
    Grid& grid; // The grid containing the wandering robot

};
#endif
```

In this case, we don't want a *copy* of the grid for our view, but the *exact same object* as the controller is changing. Otherwise, the view would never change. We accomplish this by using a *reference*.

References – Less Prickly Pointers

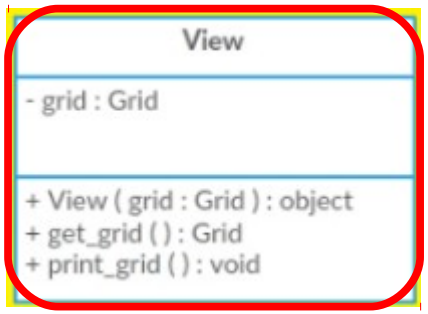
- In C++, a *reference* is another name for an existing variable. The same variable may be accessed using either the variable name or the reference name.



- Changing Question_Resolved also changes The_Answer – it's exactly the same memory!
- A reference must *always* point to an active variable of the correct type. Unlike pointers, references never “dangle”.
  - Well, *almost* never...

Using a Reference

- First we declare a reference to a Grid object and name the reference “grid”
  - This must be given the address of a Grid object in either the declaration or the constructor
- Second, the View constructor receives *a reference to* a Grid object as “g”, and assigns it immediately to “grid” via “: grid{g}”
  - By the time the object is constructed, the reference is defined (and can never change)



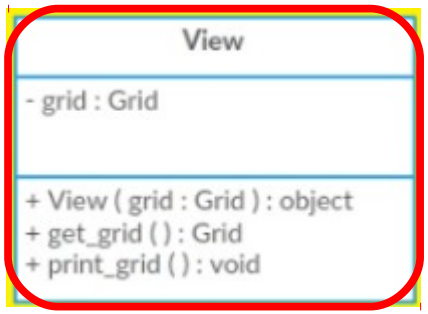

```
class View {
  public:
    View(Grid& g);

    void print_banner();
    Grid get_grid();
    void print_grid();

  private:
    Grid& grid; // The grid containing the wandering robot

};
```

A Brief Note on Making the Sausage

- The original C++ 11 standard erroneously declared that a reference must be direct-assigned to a variable using *parentheses* (per older C++ standards)
  - `View(Grid& g) : grid{g} {}; // incorrectly incorrect`
  - `View(Grid& g) : grid(g) {}; // incorrectly correct`
- Gcc 4.8 implemented the standard literally, and thus flagged the first version as an error even when using the -std=c++11 flag
- The error was corrected in a standards addendum, and gcc 5.x and later accepts both syntax

Step 8: View Implementation

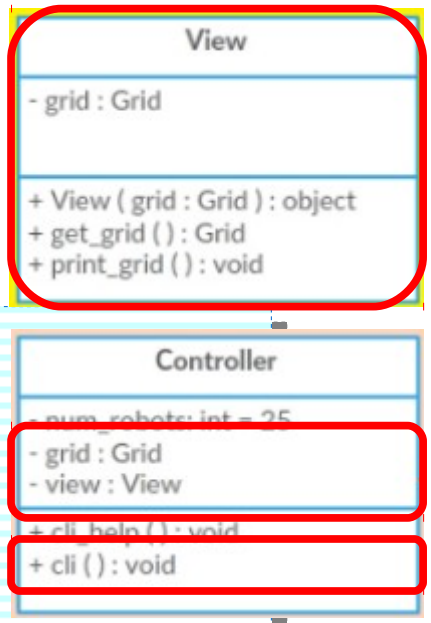
- References are used exactly like copies
  - No special notation, e.g., ->
  - We'll get to pointers (and ->) later




```
#include "view.h"
#include <iostream>

View::View(Grid& g) : grid{g} {};

void View::print_banner() {
    cout << "============" << endl
         << "ROVING ROBOT" << endl
         << "============" << endl << endl;
}

Grid View::get_grid() {return grid;}
void View::print_grid() {
  cout << grid.to_string() << endl;
};
```

Step 8: Updating Controller

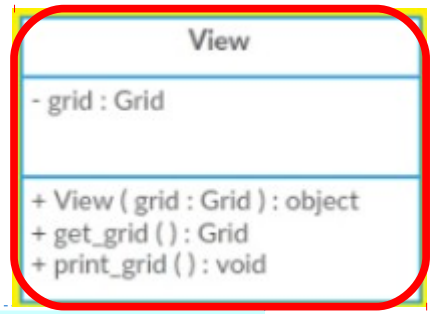
- We automatically create a view when controller is instanced.

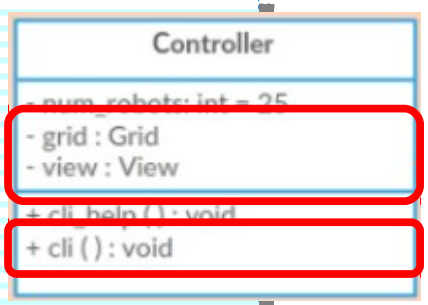



```
#include "robot.h"
#include "grid.h"
#include "view.h"

class Controller {
  public:
    void cli();
    void execute_cmd(string cmd);
    View get_view();

  private:
    Robot _player;
    Grid _grid{_player};
    View _view{_grid};
};
```

Step 8: Updating Controller

- We now can call view to handle major output.
- Note that prompts (as with all minor output) remain with controller, not delegated to view.

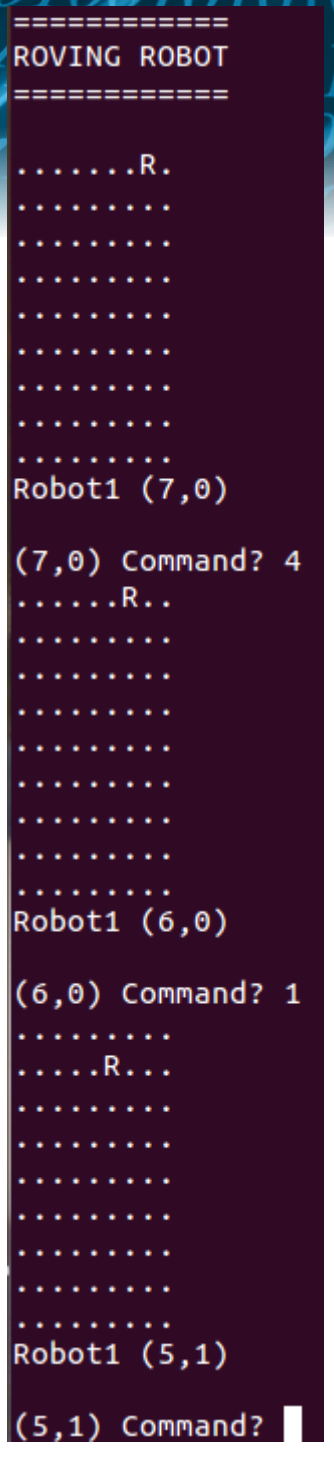





```
void Controller::cli() {
    _view.print_banner();
    string cmd = "x";
    while (cmd[0] != '0') {
        _view.print_grid();
        cout << _grid.get_player().get_coordinate().to_string()
             << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
    }
};
```

Step 8: Interactive Tests

- Our robot roves!
- This is useful, so we elect to keep grid.cpp as a separate interactive test unit.
  - We'll create a separate grid class in multigrid.cpp as our next step
  - Obviously, only one grid class can be used within a given program, selected via the “include”
- But what about automated tests?



Automating UI Tests: Option 1

- In controller, we separated cli() and execute_cmd() to faciliate automated tests.
  - We could use a vector called tests of type string, and load commands into it. We could then call execute_cmd() followd by print_grid().
  - This requires additional coding, but can be done completely in C++.

Automating UI Tests: Option 1

```
#include "globals.h"
#include "controller.h"
#include "grid.h"
#include "robot.h"
#include "coordinate.h"
#include <iostream>
#include <vector>

int main() {
  vector<string> tests;
  tests.push_back("4");
  tests.push_back("1");
  tests.push_back("2");
  tests.push_back("3");
  tests.push_back("6");
  tests.push_back("9");
  tests.push_back("8");
  tests.push_back("7");
```

```
  cout << "=================" << endl
       << "ROVING ROBOT TEST" << endl
       << "=================" << endl << endl;

  Controller controller;
  controller.get_view().print_grid();
  for(int i=0; i<tests.size(); ++i) {
     cout << " Command? " << tests[i] << endl;
     controller.execute_cmd(tests[i]);
     controller.get_view().print_grid();
   }
}
```

```
ricegf@pluto:~/dev/cpp/201701/06/08-add_view$ make controller
g++ -std=c++11 -c test_controller.cpp
g++ -std=c++11 -c controller.cpp
g++ -std=c++11 -c view.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 test_controller.o controller.o view.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/08-add_view$ ./a.out
=================
ROVING ROBOT TEST
=================

.........
.......R.
.........
.........
.........
.........
.........
.........
.........

 Command? 4
```

Automating UI Tests: Option 2

- Since we are relying on STDIN to receive our commands, we can use input redirection in the bash shell. (Most shells support input redirection.)
  - This requires little coding, but requires familiarity with the bash (or another) shell.

Automating UI Tests: Option 2

```
ricegf@pluto:~/dev/cpp/201701/06/08-add_view$ make
g++ -std=c++11 -c main.cpp
g++ -std=c++11 -c controller.cpp
g++ -std=c++11 -c view.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 main.o controller.o view.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/08-add_view$ cat test_vector.txt
4
1
2
3
6
9
8
7
0
ricegf@pluto:~/dev/cpp/201701/06/08-add_view$ ./a.out < test_vector.txt
============
ROVING ROBOT
============


.........
.......R.
.........
.........
.........
.........
.........
.........
.........

(7,1) Command?
.........
......R..
.........
.........
```

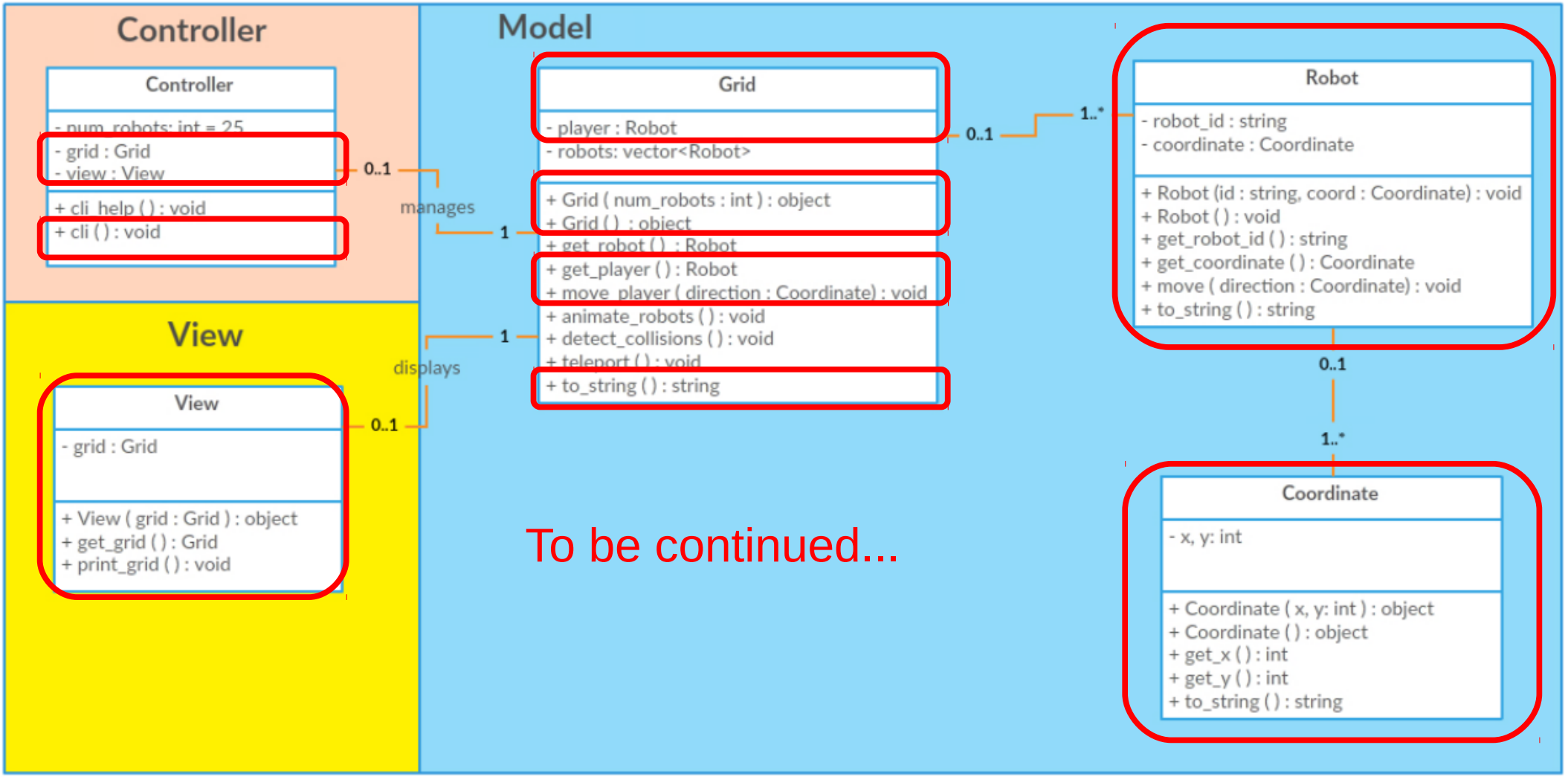
Building our CLI app.

The “test vector” of commands to drive our app.

Output looks similar to the C++ version. Note that the commands aren't shown. However, we're testing cli() in addition to execute_cmd().

Where We Stand

- We have a single, roving robot.



Quick Review

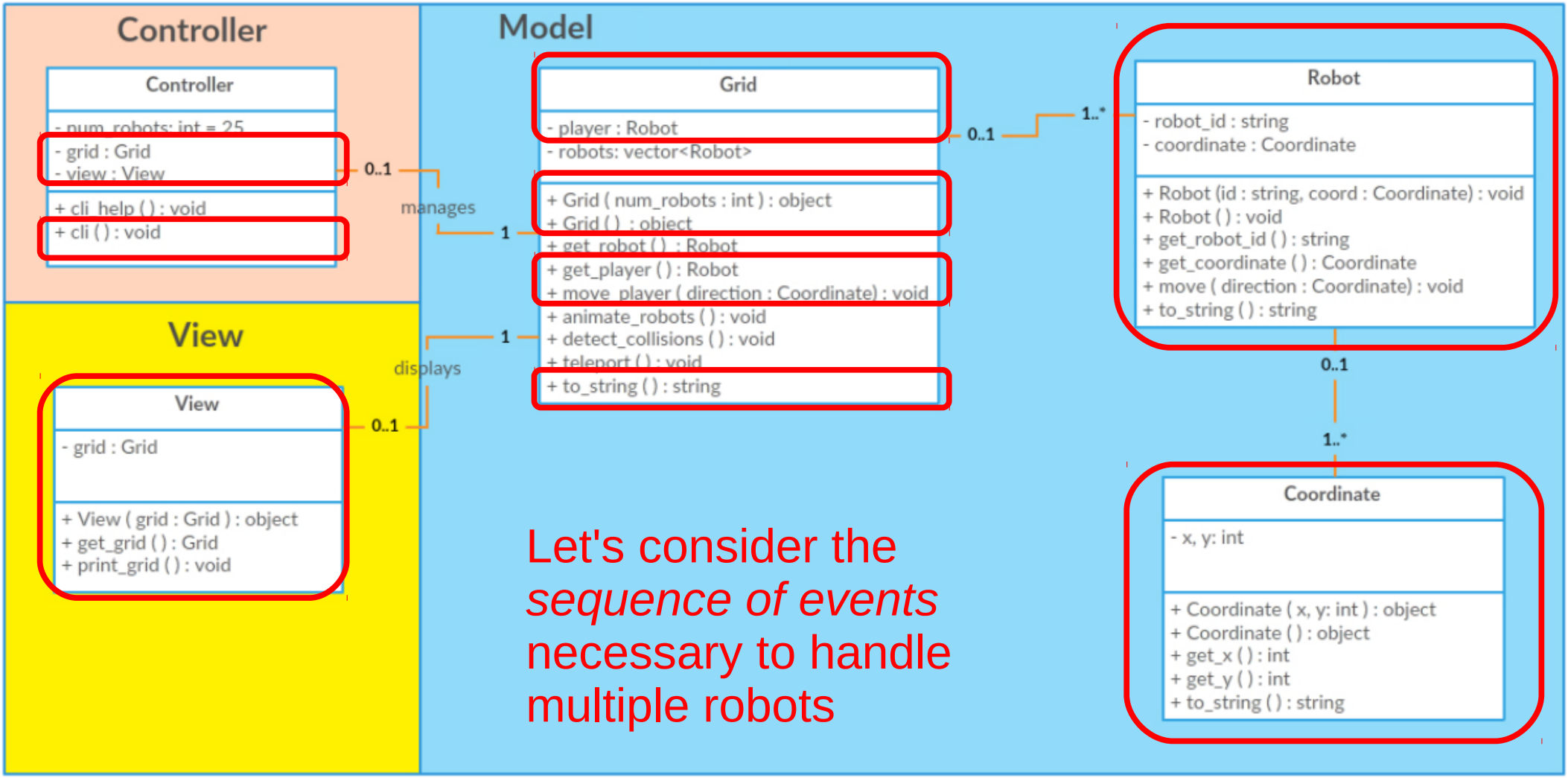
- A series of interactions that enable an actor to achieve a goal is a ___ ____.
- True or false: In UML, a use case is treated as a class.
- The pattern that separates the business logic, data visualization, and human or machine user is the _____ – ____ – _____ (___) pattern.
- An object-oriented design is usually implemented starting with the class with the fewest _____. We implement a small set of functionality along with a _____ _____. A ____ _____ can make this task easier.
- Code for which specified assertions are guaranteed to be true is _____.
- To retain the value of a method-level variable between calls to that method, we declare the variable to be _____.
- Rather than a to_string method, a more C++-like approach to providing a string representation of an object is to overload the _____.

For Next Class

- **Homework #3 is due Thursday, Sep 14 at 8 am**
- We'll complete the Roving Robots program
- Prepare for the **Exam on Tuesday, Sep 19** using the Blackboard materials
  - Review and learn from the study sheet
  - Take the practice exam in as realistic a setting as practical – quiet, just you and your pencils
    - Correct answers are already posted – don't peek!
  - Bring questions for the 2nd half of the next class

Where We Stand

- We have a single, roving robot. What next?

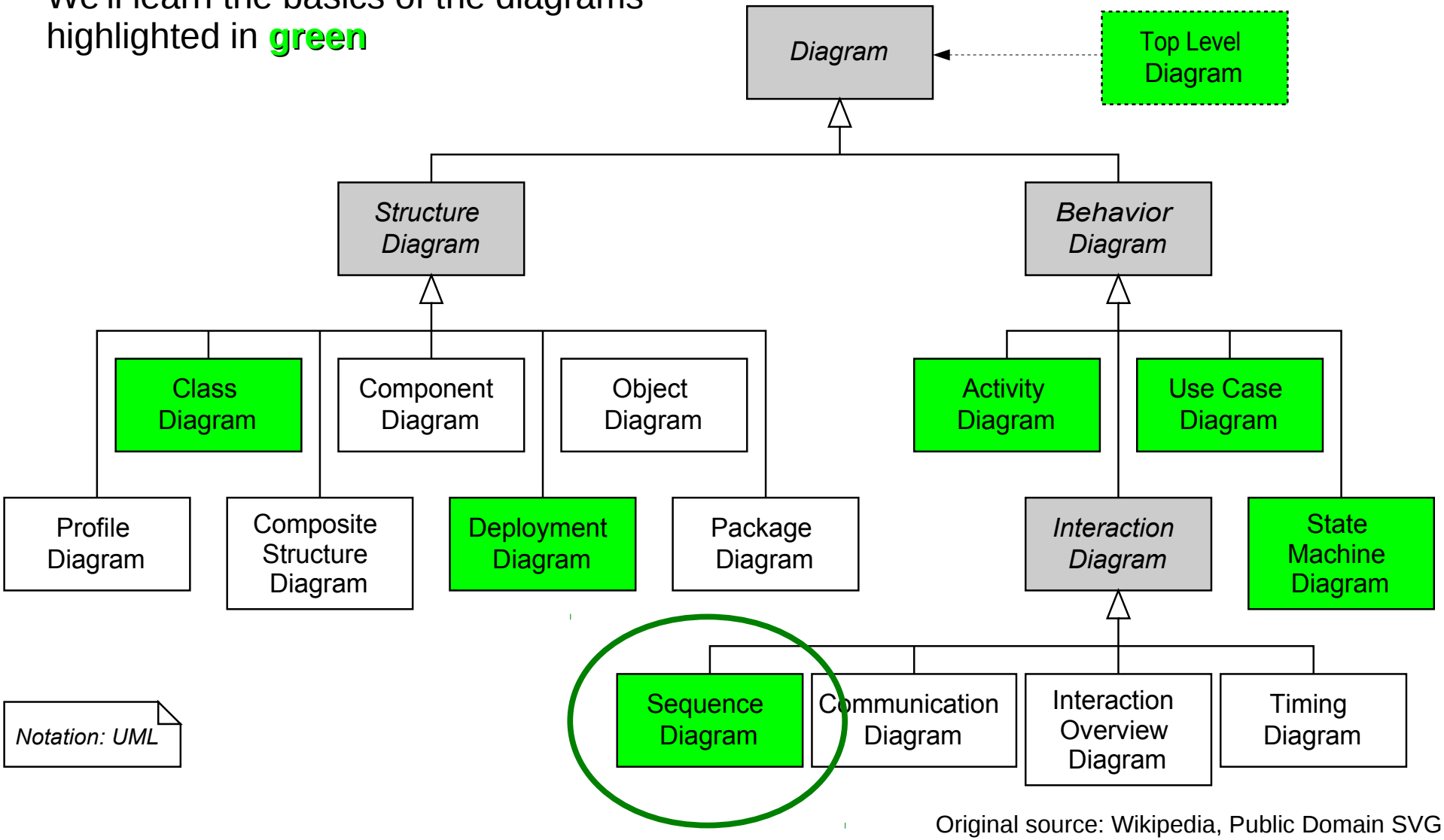


UML Sequence Diagram

- Sequence is the most common *interaction* diagram
  - Interaction is between users, objects (and static class methods), and external entities
  - Focus is on a *single program thread* and the message or method call interchange between a number of object lifelines
- Objects are spaced across the top, with descending dashed lines called *lifelines*
  - The Y axis is *time*, advancing downward
  - Messages or method calls are horizontal labeled arrows at a specific point in time
  - Processing is thick gray lines on the lifelines, consuming time
  - All UML conventions apply (comments, stereotypes, etc)

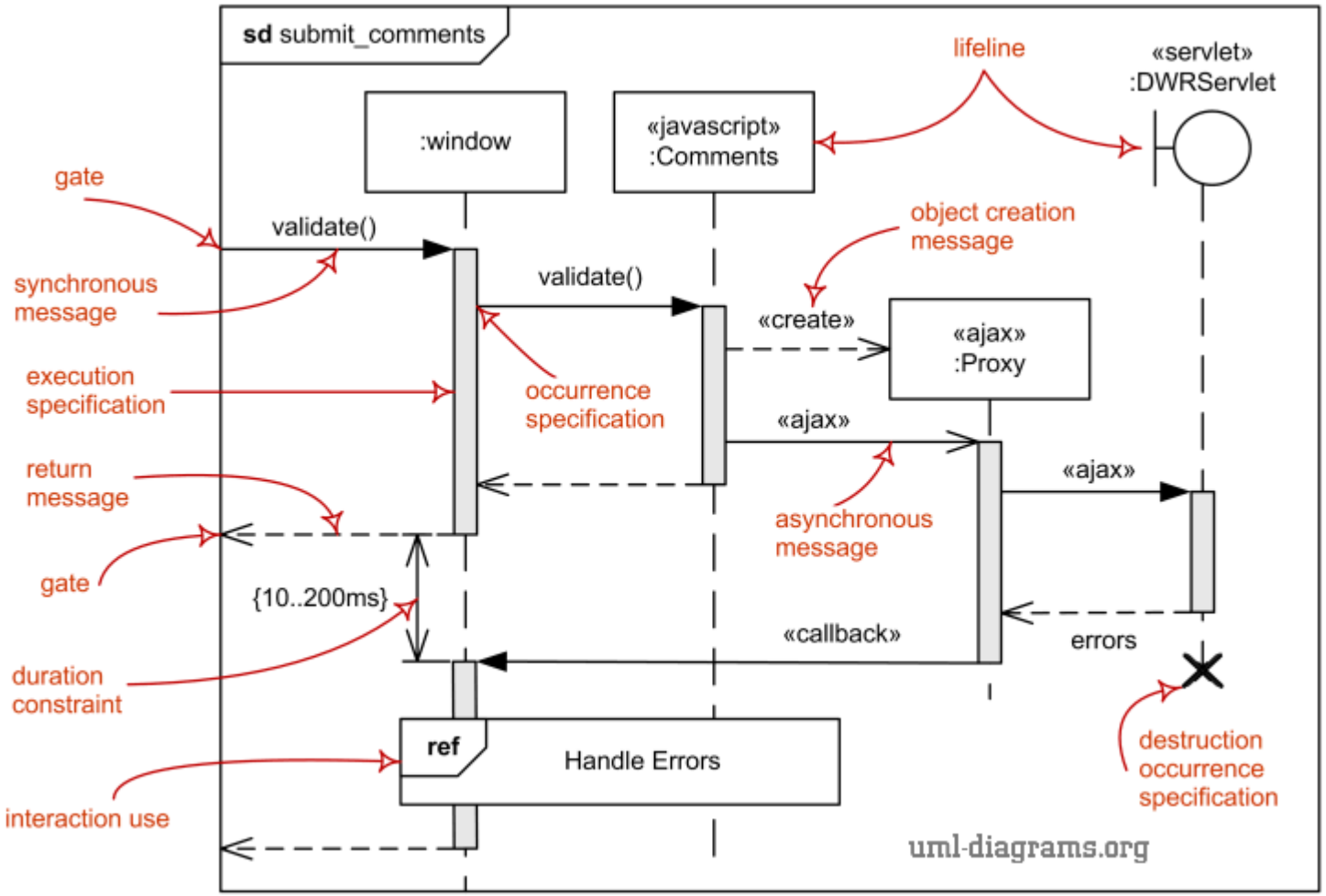
The Sequence Diagram in Context

We'll learn the basics of the diagrams highlighted in **green**



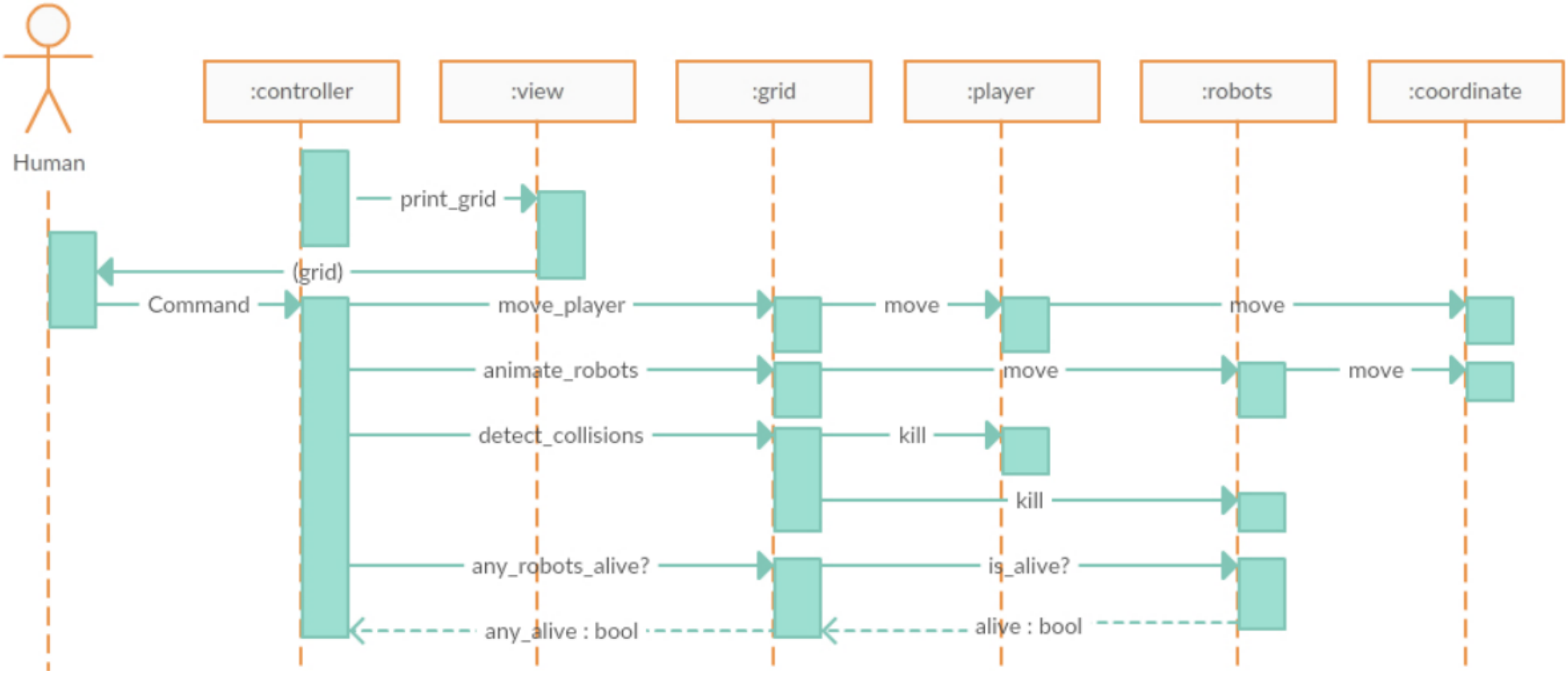
Original source: Wikipedia, Public Domain SVG

Basic Sequence Diagram Elements



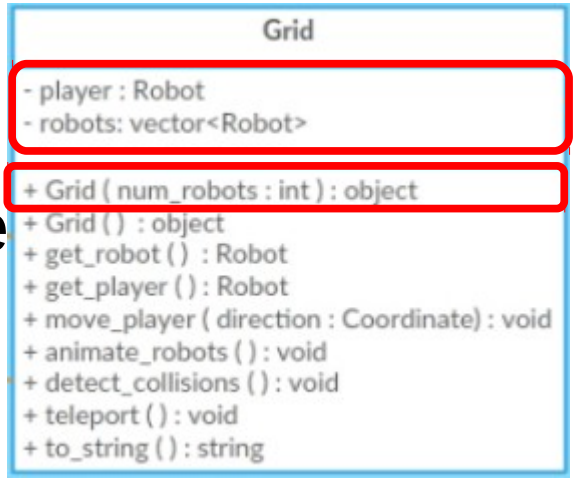
http://www.uml-diagrams.org/sequence-diagrams.html

UML Sequence Diagram for Roving Robots



Step 9: Many Robots, 1 Player

- First, let's redesign Grid's fields to track a distinct player and a lot of robots
  - We create the vector “robots” to manage the non-player robots
  - We also update constructors to create the desired number of non-player robots and add them our vector



grid.h

```
class Grid {
  private:
    Robot _player{"Ralph", Coordinate{MAX_X/2, MAX_Y/2}};
                            // The robot controlled by the player
    vector<Robot> robots;   // The robots wandering the grid

  public:
    Grid(int num_robots) {
      for (int i=0; i < num_robots; ++i)
        robots.push_back(Robot{});
    }
    Grid() : Grid(NUM_ROBOTS) { }
```

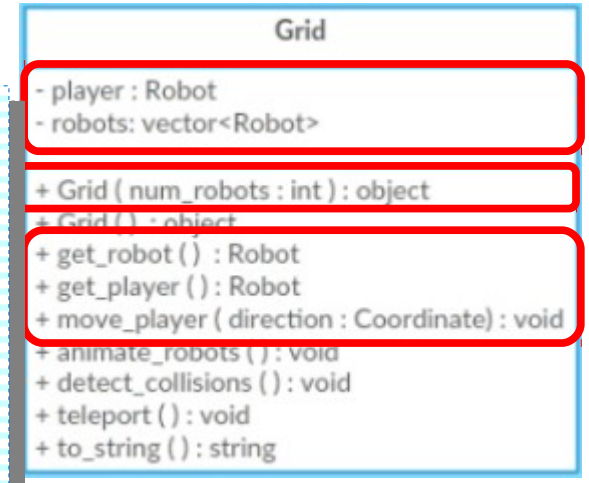
(public precedes private in the actual file...)

Step 9: Many Robots, 1 Player

- We update our globals.h...

globals.h

```
#ifndef GLOBALS
#define  GLOBALS 2016

const int MAX_X = 29; // x is in range [0,MAX_X]
const int MAX_Y = 29; // y is in range [0,MAX_Y]

const int NUM_ROBOTS = 10; // number of robots in grid

#endif
```



- ...add the methods to manage the robot horde...

grid.h

```
    Robot get_player( );
    void move_player(Coordinate direction);

    int get_num_robots();
    Robot get_robot(int robot_id);
    void move_robot(int robot_id, Coordinate direction);

    string to_string();
```

Step 9: Many Robots, 1 Player

- ...and rework to_string() to show the robots

grid.cpp

```
string to_string() {
  string grid = "";
  for (int y = 0; y < MAX_Y; ++y) {
    for (int x = 0; x < MAX_X; ++x) {
      char this_pos = '.';
      for(int i=0; i<robots.size(); ++i) {
        if (x == robots[i].get_coordinate().get_x() &&
            y == robots[i].get_coordinate().get_y())
          this_pos = 'X';
      }
      if (x == player.get_coordinate().get_x() &&
          y == player.get_coordinate().get_y())
        this_pos = 'R';
      grid += this_pos;
    }
    grid += '\n';
  }
  grid += player.to_string() + '\n';
  return grid;
}
```

Step 9: Complete grid.h

grid.h

```
#ifndef GRID
#define  GRID 2017
#include "robot.h"
#include <vector>

class Grid {
  public:
    Grid(int num_robots) {
      for (int i=0; i < num_robots; ++i)
        robots.push_back(Robot{});
    }
    Grid() : Grid(NUM_ROBOTS) { }

    Robot get_player( );
    void move_player(Coordinate direction);

    int get_num_robots();
    Robot get_robot(int robot_id);
    void move_robot(int robot_id, Coordinate direction);

    string to_string();
  private:
    Robot _player{"Ralph", Coordinate{MAX_X/2, MAX_Y/2}};
    vector<Robot> robots; // The robots wandering the grid
};
#endif
```

Step 9: Complete grid.cpp

grid.cpp

```
#include "grid.h"
#include "globals.h"
#include "robot.h"

Robot Grid::get_player( ) {return _player;}
void Grid::move_player(Coordinate direction) {_player.move(direction);}

int Grid::get_num_robots() {return robots.size(); }
Robot Grid::get_robot(int robot_id) {return robots[robot_id];}
void Grid::move_robot(int robot_id, Coordinate direction)
{robots[robot_id].move(direction);}

string Grid::to_string() {
  string grid = "";
  for (int y = 0; y < MAX_Y; ++y) {
    for (int x = 0; x < MAX_X; ++x) {
      char this_pos = '.';
      for(int i=0; i<robots.size(); ++i) {
        if (x == robots[i].get_coordinate().get_x() &&
            y == robots[i].get_coordinate().get_y())
          this_pos = 'X';
      }
      if (x == _player.get_coordinate().get_x() &&
          y == _player.get_coordinate().get_y())
        this_pos = 'R';
      grid += this_pos;
    }
    grid += '\n';
  }
  return grid;
}
```

Step 9: Regression Testing

```
#include "globals.h"
#include "robot.h"
#include "grid.h"
#include <string>
#include <iostream>
using namespace std;

int main() {
  srand(0xbadcafe);
  string expected =
".........................\n.........................\n.............
...........\n.........X.................\n.........................\n...
.....................\n.........................\n...................
......\n.........................\n.........................\n........
..................\n.........................\n.....................X...
..\n..X.....................\n............R............\n............
............\n.........................\n....X.....................\n.
........X......X.........\n......X..................\n................
.........\n...................X.....\n.........................\n.....
....................\n..........X..............\n....................
.....\n..................X.......\n.........................\n..........
................\n";
  string actual = "";
  Grid grid{};
  actual += grid.to_string( );
  if (expected != actual) {
    cerr << "fail: grid.h: initialization test #1" << endl;
    cerr << "expected: \"" << expected << "\"" << endl;
    cerr << "actual:   \"" << actual << "\"" << endl << endl;
    return -1;
  }
```

Step 9: Tests

- Don't forget to test!

```
ricegf@pluto:~/dev/cpp/201701/06/09-add_mult_robots$ make grid
g++ -std=c++11 test_grid.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/09-add_mult_robots$ ./a.out
ricegf@pluto:~/dev/cpp/201701/06/09-add_mult_robots$
```

Regression

```
ricegf@pluto:~/dev/cpp/201701/06/09-a
============
ROVING ROBOT
============


........................X....
.........................X...
.............................
.........................X...
.............................
.............................
.............................
.............................
.........X...................
.............................
.............................
.............................
.............................
..X..........................
..............R..............
...............X.............
.............................
...X..............X..........
............................X
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.......................X.....

(14,14) Command?
```

Interactive

(other interactive test results not shown)

Step 10: Detect Collisions

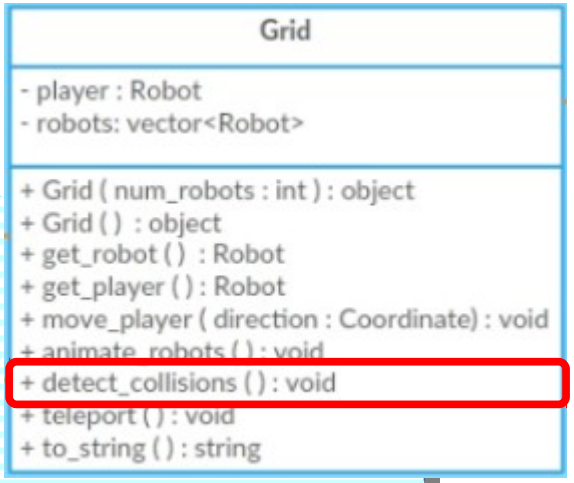
- The spec says that robots who collide die
  - We need to track whether each robot (and the player – same class) is alive or dead
  - We need to ensure that dead robots don't move

```
class Robot {
  public:
    Robot(string robot_id, Coordinate coordinate) :
      _robot_id{robot_id}, _coordinate{coordinate}, _alive{true} { }
    Robot() : Robot("Robot"+std::to_string(generateID()), Coordinate()) { }

    string get_robot_id( );
    Coordinate get_coordinate( );
    string to_string( );

    void move(Coordinate direction);
    void kill();
    bool is_alive();

  private:
    string _robot_id; // "Name" of the robot to display on-screen
    Coordinate _coordinate;  // location of robot on an X-Y grid
    bool _alive;
```

Step 10: Detect Collisions

- Check for collisions, and mark colliding robots as dead

```
void Grid::detect_collisions( ) {
  for (int i=0; i<robots.size(); ++i) {
    if (_player.get_coordinate().get_x()
     == _robots[i].get_coordinate().get_x() &&
        _player.get_coordinate().get_y()
     == _robots[i].get_coordinate().get_y()) {
      _player.kill();
      _robots[i].kill();
    }
    for (int j=i+1; j<robots.size(); ++j) {
      if (_robots[j].get_coordinate().get_x()
       == _robots[i].get_coordinate().get_x() &&
          _robots[j].get_coordinate().get_y()
       == _robots[i].get_coordinate().get_y()) {
        _robots[j].kill();
        _robots[i].kill();
      }
    }
  }
}
```



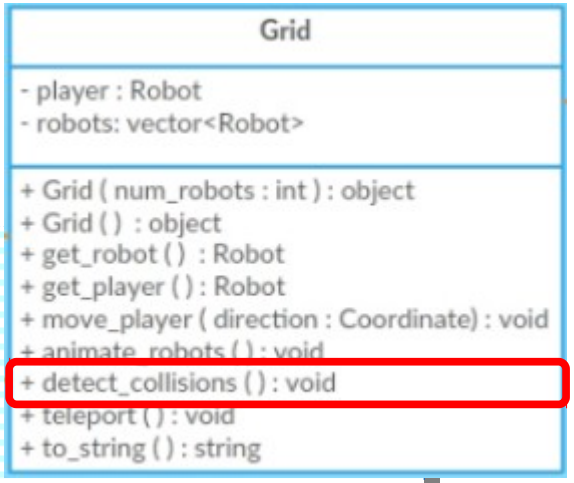
– Shouldn't the Coordinate or Robot class know how to compare coordinates? What are our options?

– If the player was always simply the first element of the robots vector, i.e., robots[0], wouldn't this simplify our code? Any downsides?

– Are we wasting CPU cycles by comparing already dead robots to each other every move?

Step 10: Detect Collisions

- Theoretical version with a Robot collide(Robot) method

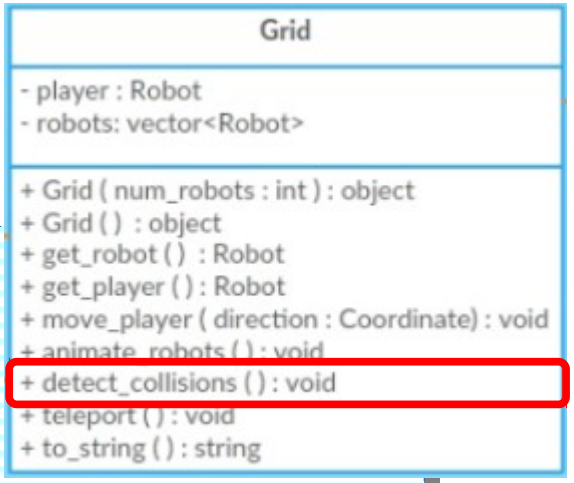



```
void Grid::detect_collisions( ) {
  for (int i=0; i<robots.size(); ++i) {
    if (_player.collide(_robots[i]) {
      _player.kill();
      _robots[i].kill();
    }
    for (int j=i+1; j<robots.size(); ++j) {
      if (_robots[j].collide(_robots[i]) {
        _robots[j].kill();
        _robots[i].kill();
      }
    }
  }
}
```

Would it be premature optimization to clean this up now?
Or wait for the refactoring?

Step 10: Detect Collisions

- We'll represent dead robots with an '*'




```
string Grid::to_string() {
  string grid = "";
  for (int y = 0; y < MAX_Y; ++y) {
    for (int x = 0; x < MAX_X; ++x) {
      char this_pos = '.';
      for(int i=0; i<_robots.size(); ++i) {
        if (x == _robots[i].get_coordinate().get_x() &&
            y == _robots[i].get_coordinate().get_y())
          this_pos = _robots[i].is_alive() ? 'X' : '*';
      }
      if (x == _player.get_coordinate().get_x() &&
          y == _player.get_coordinate().get_y())
        this_pos = _player.is_alive() ? 'R' : '*';
      grid += this_pos;
    }
    grid += '\n';
  }
  return grid;
}
```

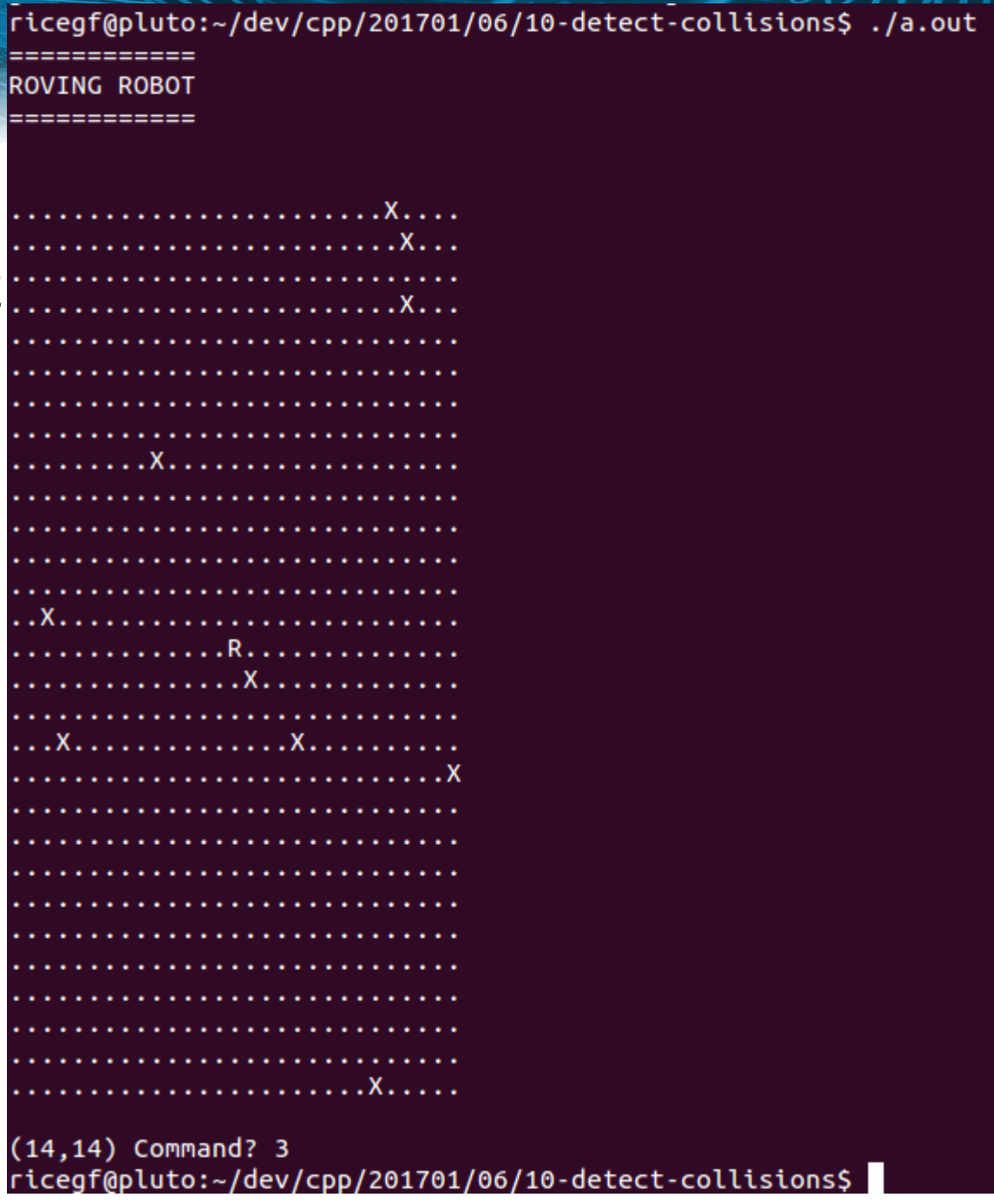
Step 10: Detect Collisions

- If the player collides with a robot, dead or alive, the game is over

```
void Controller::cli() {
    _view.print_banner();
    string cmd = "x";
    while (cmd[0] != '0' && _grid.get_player().is_alive()) {
        _view.print_grid();
        cout << _grid.get_player().get_coordinate().to_string() << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
        _grid.detect_collisions();
    }
};
```

Step 10: Test

- Three problems
  - The robots always start in the same place
  - The game exits without showing the final grid
  - Robots can (still) move off the grid
- We'll fix those bugs in the same sprint in which we animate our robots



Step 11: Fix 2 bugs...

- Keep the robots on the grid

```
void Robot::move(Coordinate direction) {
    int x = min(max(_coordinate.get_x() + direction.get_x(), 0), MAX_X);
    int y = min(max(_coordinate.get_y() + direction.get_y(), 0), MAX_Y);
    _coordinate = Coordinate{x, y};
}
```

- Show the grid before exiting

```
void Controller::cli() {
    _view.print_banner();
    string cmd = "x";
    while (cmd[0] != '0' && _grid.get_player().is_alive()) {
        _view.print_grid();
        cout << _grid.get_player().get_coordinate().to_string() << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
        _grid.detect_collisions();
    }
    _view.print_grid();
};
```

Step 11: ...add srand...

- Seed the pseudo-random number generator

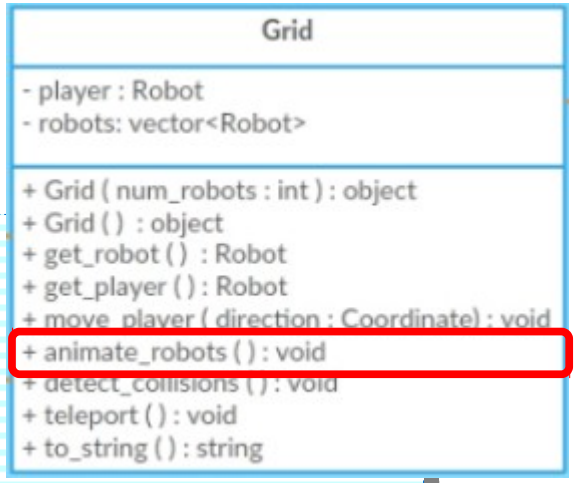
```
class Grid {
  public:
    Grid(int num_robots) {
      srand(time(NULL)); // seed rand()
      for (int i=0; i < num_robots; ++i)
        _robots.push_back(Robot{});
    }
```

Step 11: ...and Animate the Robots

- Write animate_robots()

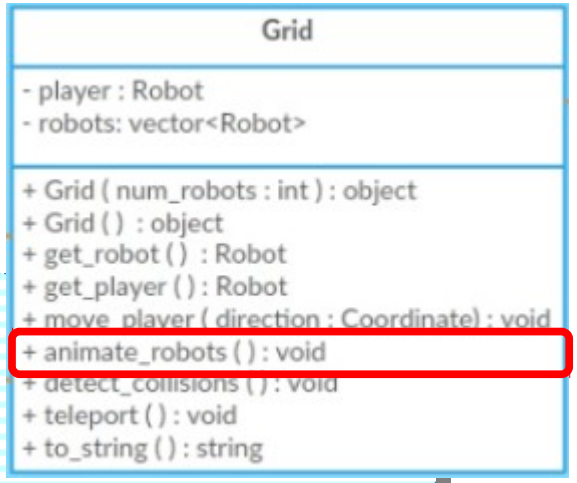
```
void Grid::animate_robots( ) {
  for (Robot& robot : _robots) {
    if (robot.is_alive()) {
      int delta_x = robot.get_coordinate().get_x() -
                    _player.get_coordinate().get_x();
      int delta_y = robot.get_coordinate().get_y() -
                    _player.get_coordinate().get_y();
      robot.move(Coordinate(delta_x < 0 ? 1 : (delta_x > 0 ? -1 : 0),
                            delta_y < 0 ? 1 : (delta_y > 0 ? -1 : 0)));
    }
  }
}
```

Is this necessary?



Step 11: ...

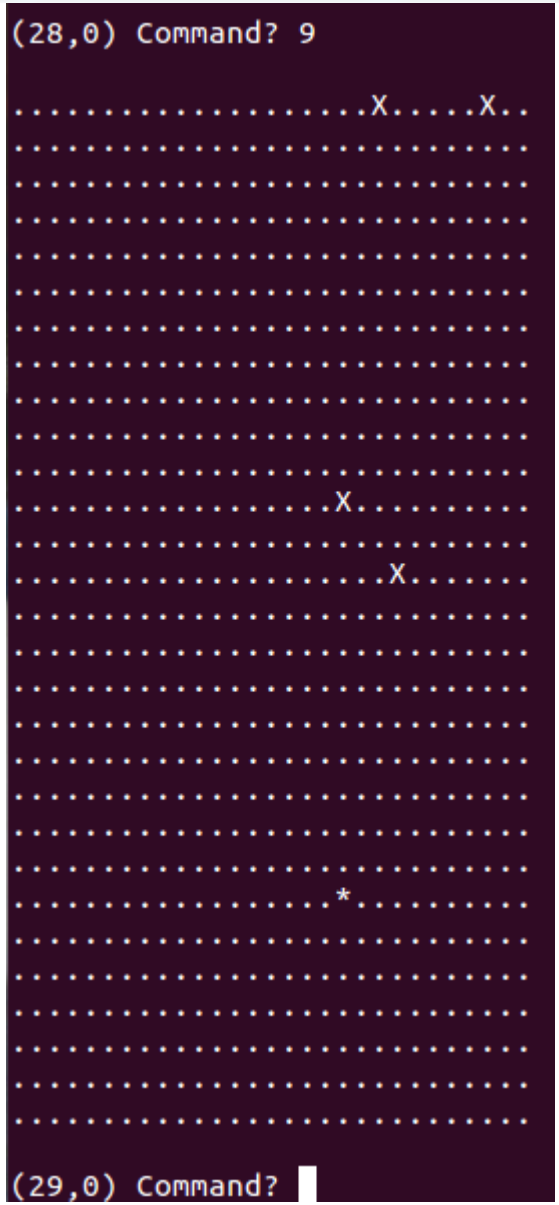
- Call animate_robots() before detect_collisions()




```
void Controller::cli() {
    _view.print_banner();
    string cmd = "x";
    while (cmd[0] != '0' && _grid.get_player().is_alive()) {
        _view.print_grid();
        cout << _grid.get_player().get_coordinate().to_string() << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
        _grid.animate_robots();
        _grid.detect_collisions();
    }
    _view.print_grid();
};
```

Step 11: … and Test

- Wait – where'd Ralph go?
  - Woops – a robot can step one dot off of the grid to the right!
- This is called an “edge case”
  - Most bugs occur “on the edges” of inputs and variables - minimum or maximum values and unusual values
  - Thorough testing of edge cases should have caught this bug sooner



Step 11: Prepare to Debug!

- First, we set up the conditions for the bug
  - This just saves time (but beware of Observer Effect)

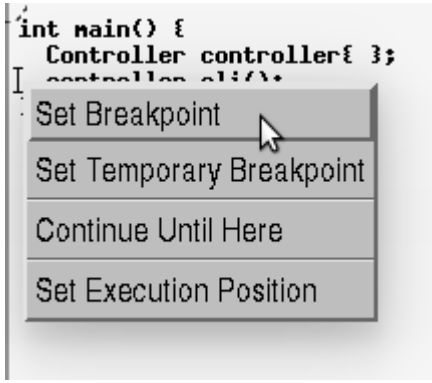
```
private:
  // Robot _player{"Ralph", Coordinate{MAX_X/2, MAX_Y/2}}; // The robot contro
  Robot _player{"Ralph", Coordinate{28, 28}}; // TODO: Just for quick testing
  vector<Robot> _robots; // The robots wandering the grid
```

- Then rebuild using the -g option
  - This inserts extra debug info into the executable

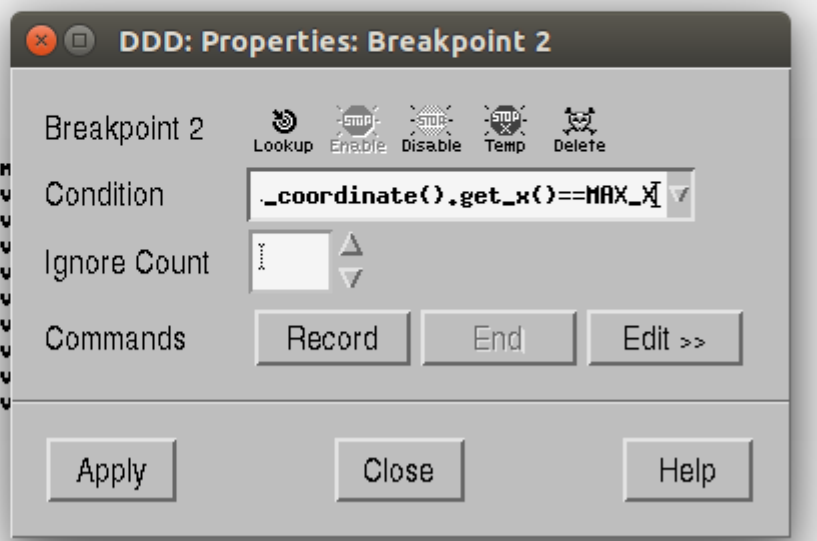
```
ricegf@pluto:~/dev/cpp/201701/06/11-animate_robots$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/11-animate_robots$ make debug
g++ -std=c++11 -g -c main.cpp
g++ -std=c++11 -g -c controller.cpp
g++ -std=c++11 -g -c view.cpp
g++ -std=c++11 -g -c grid.cpp
g++ -std=c++11 -g -c robot.cpp
g++ -std=c++11 -g -c coordinate.cpp
g++ -std=c++11 -g main.o controller.o view.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/11-animate_robots$ ddd a.out
```

Step 11: Debug

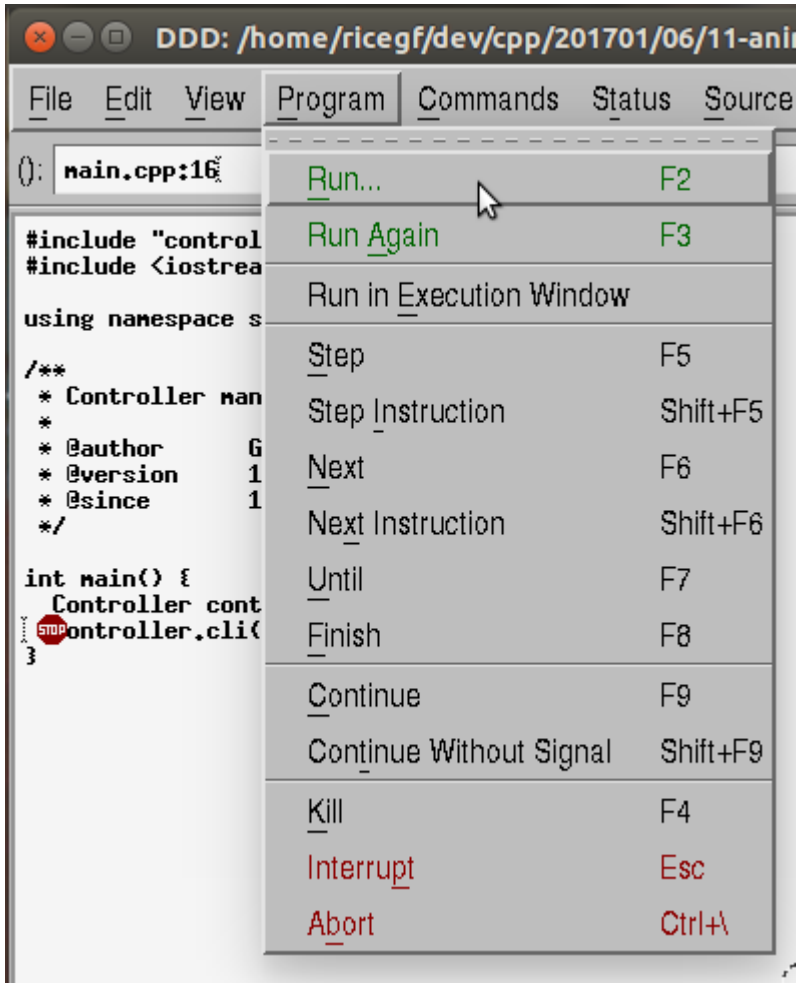
- Launch the debugger with 'ddd a.out'
  - Set a breakpoint with right-click → Set Breakpoint
  - The breakpoint conditions are configurable

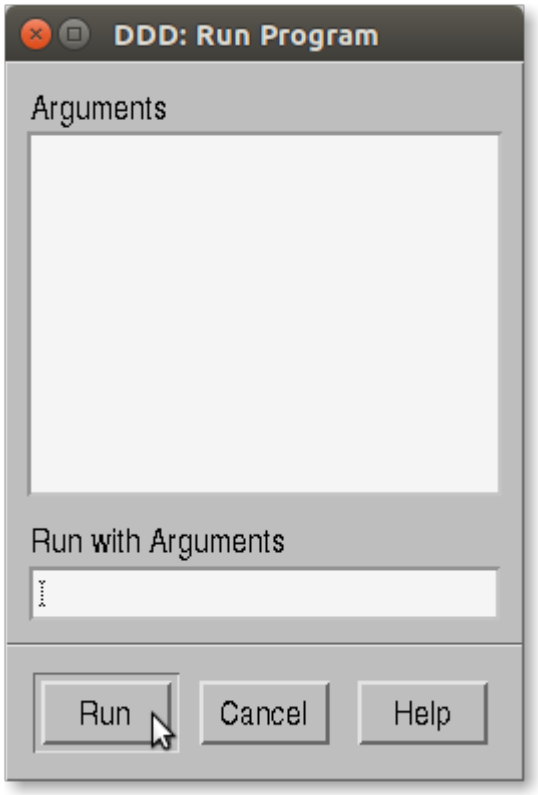



```
View Controller::get_view() {return _view;}

void Controller::cli() {
    _view.print_banner();
    string cmd = "x";
    while (cmd[0] != '0' && _grid.get_player().is_alive()) {
        _view.print_grid();
        cout << _grid.get_player().get_coordinate().to_string() << " Command? ";
        cin >> cmd;
        execute_cmd(cmd);
        _grid.animate_robots();
        _grid.detect_collisions();
    }
    _view.print_grid();
};
void Controller::execute_cmd(string cm
        if (cmd[0] == '1') { _grid.mov
        if (cmd[0] == '2') { _grid.mov
        if (cmd[0] == '3') { _grid.mov
        if (cmd[0] == '4') { _grid.mov
        if (cmd[0] == '5') { _grid.mov
        if (cmd[0] == '6') { _grid.mov
        if (cmd[0] == '7') { _grid.mov
        if (cmd[0] == '8') { _grid.mov
        if (cmd[0] == '9') { _grid.mov
};
```



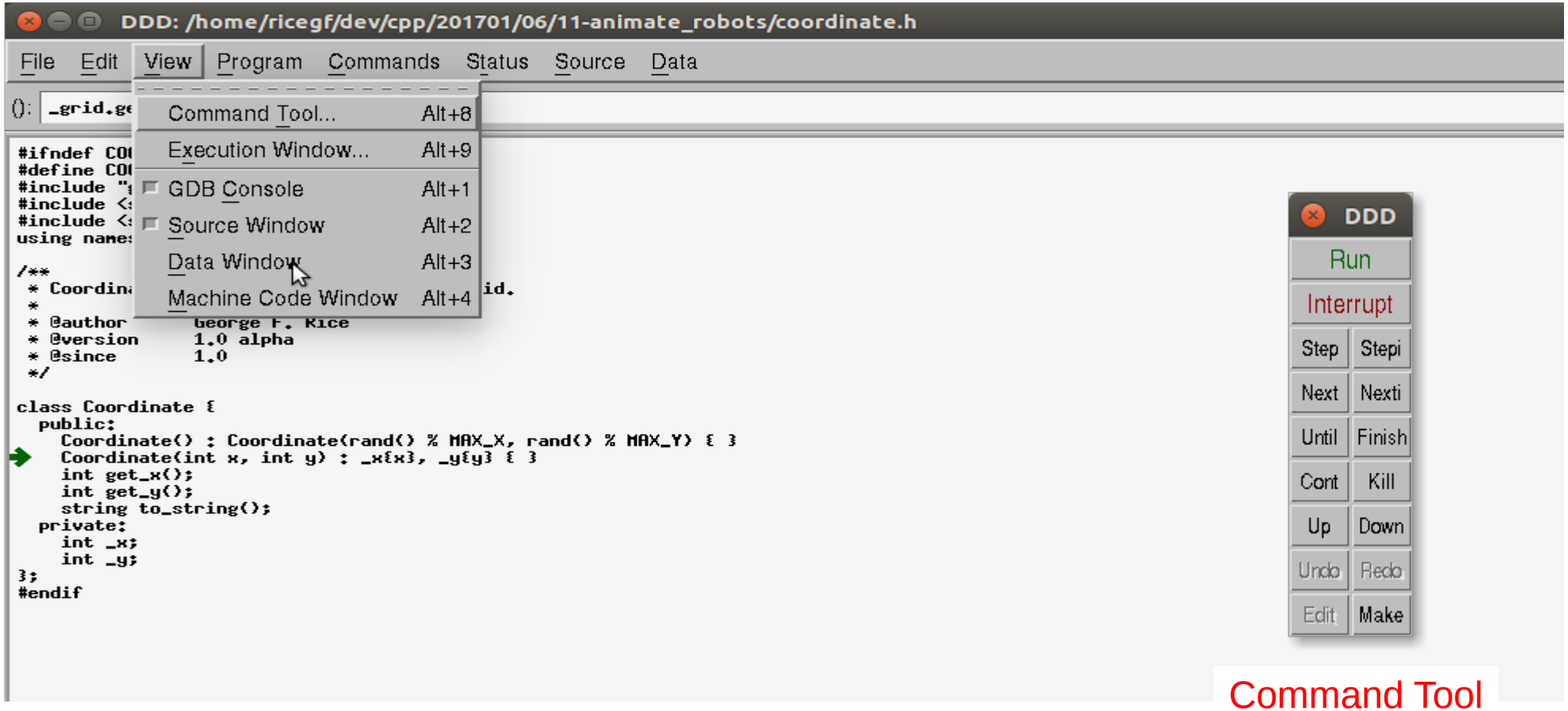
Step 11: Run






```
GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea LReading symbols from a.out...done.
(gdb) break main.cpp:16
Breakpoint 1 at 0x401779: file main.cpp, line 16.
(gdb) set args
(gdb) run
Starting program: /home/ricegf/dev/cpp/201701/06/11-animate_robots/a.out

Breakpoint 1, main () at main.cpp:16
(gdb)
```

Step 11: Open the Command Tool and the Data Window

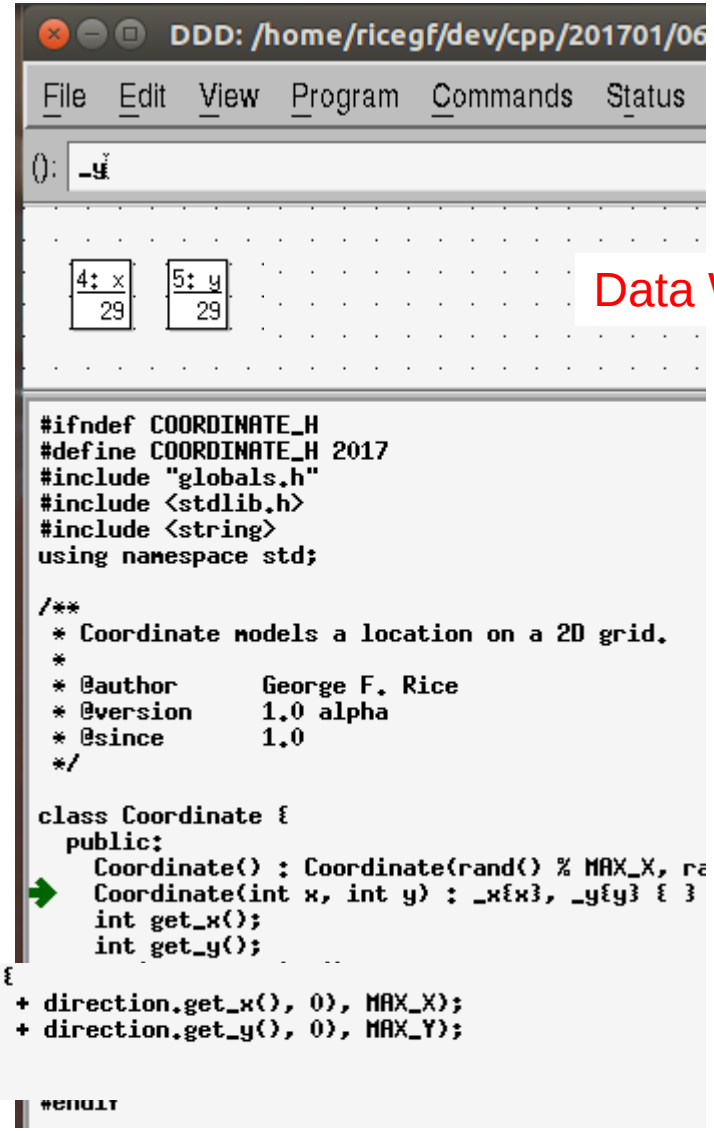


Command Tool

Step 11: Check the _x, _y Values

```
(gdb) next
(gdb) next
(gdb) next
```

```
.............X..............
............................
............................
...............X............
............................
............................
............................
............................
............................
............................
...................X........
.....................X......
............................
............................
..............X.............
.............X..............
............................
............................
............................
............................
........X...................
............................
............................
..................X.........
............................
............................
................X...........
............................
.........................X..R
```




```
void Robot::move(Coordinate direction) {
   int x = min(max(_coordinate.get_x() + direction.get_x(), 0), MAX_X);
   int y = min(max(_coordinate.get_y() + direction.get_y(), 0), MAX_Y);
_coordinate = Coordinate{x, y};
}
```

Step 12: Fix Edge Case Bug

- Fix the “off-grid” bug in class Robot

```
void Robot::move(Coordinate direction) {
    // int x = min(max(_coordinate.get_x() + direction.get_x(), 0), MAX_X);
    // int y = min(max(_coordinate.get_y() + direction.get_y(), 0), MAX_Y);
    int x = min(max(_coordinate.get_x() + direction.get_x(), 0), MAX_X-1);
    int y = min(max(_coordinate.get_y() + direction.get_y(), 0), MAX_Y-1);
    _coordinate = Coordinate{x, y};
}
```

- We found a bug via interactive test.
- Now we add a regression test to detect it.
  - Like zombies, bugs tend to return to life...

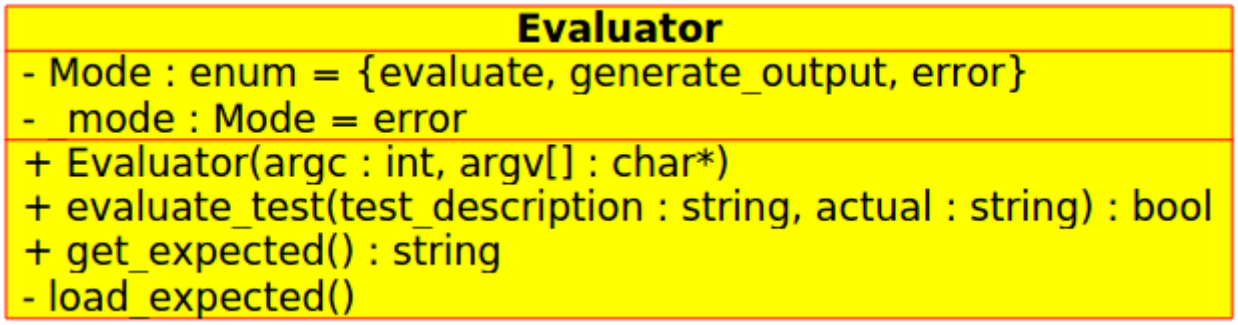
Step 12: Fix Edge Case Bug

- How to test, since Ralph starts in the middle
  - It's a long way to the edge
  - So we add a “test point”, another grid constructor that accepts an initial coordinate for Ralph
  - And we complete the set (if you're a little OCD) – 4 constructors total

```
class Grid {
  public:
    Grid(int num_robots, Coordinate ralphs_coord)
        : _player{Robot{"Ralph", ralphs_coord}}  {
      for (int i=0; i < num_robots; ++i)
        _robots.push_back(Robot{});
    }
    Grid(int num_robots) : Grid(num_robots, Coordinate{MAX_X/2, MAX_Y/2}) { }
    Grid(Coordinate ralphs_coord) : Grid(NUM_ROBOTS, ralphs_coord) { }
    Grid() : Grid(NUM_ROBOTS) { }
```

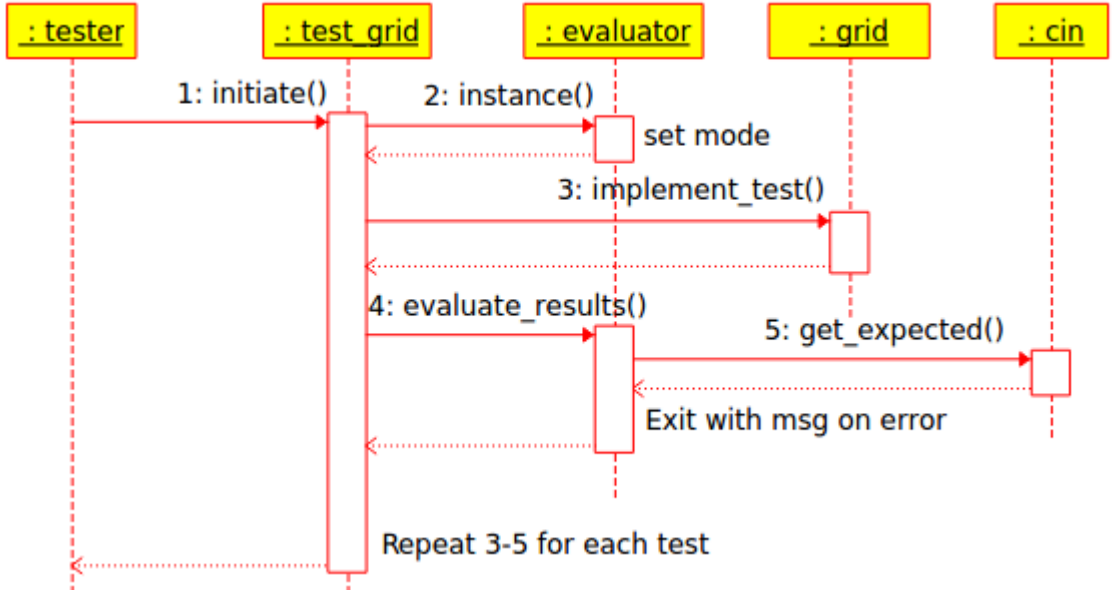
Step 12: Test Framework

- We should also test for moving off the upper left
- Copying “expected” into code is getting to be a bit of a pain
  - A framework could help with this
  - Thus we whip out the Evaluator



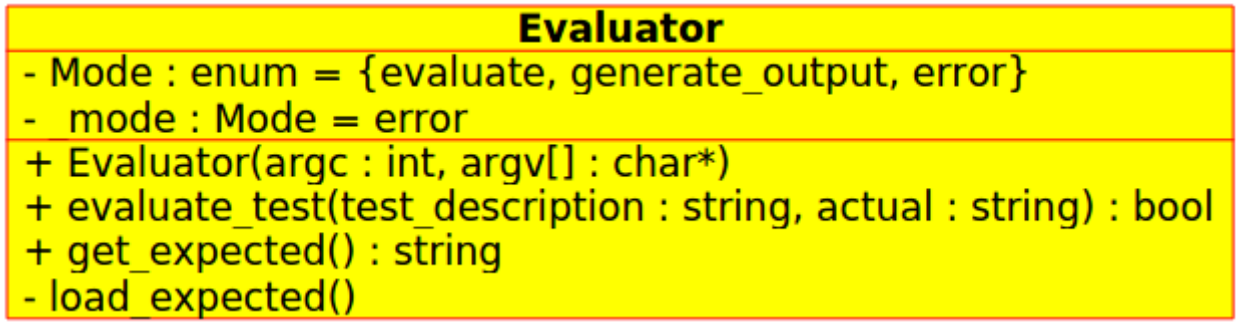
Step 12: Evaluator Class

- Evaluator handles the rote work of testing
  - Parse command line: -v outputs expected grids (./a.out -v > expected.txt), otherwise reads expected from cin and compares ('./a.out < expected.txt')
  - Results of grid manipulation sent to evaluate_results(), which exits with msg on error



Step 12: Evaluator Interface

- Our usual UML-to-C++ translation




```
#include <string>
using namespace std;

class Evaluator {
  public:
    Evaluator(int argc, char* argv[]);  // Sets the mode based on parameters
    void evaluate_test(string test_description,
                       string actual);  // Load and compare expected to actual
    string get_expected();              // Return the expected grid
  private:
    void load_expected(string test_description);  // read _expected from stdin
    string _expected;
    enum Mode {generate_output, evaluate, error}; // can generate expected or
    Mode _mode = error;                           // compare to actuals
};
```

(Implementation is on Blackboard)

Step 12: New Test_Grid

- Evaluator makes our testing much more compact

```
#include "globals.h"
#include "grid.h"
#include "coordinate.h"
#include "evaluator.h"
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  srand(0xbadcafe);
  Evaluator eval{argc, argv};

  Grid grid{};
  eval.evaluate_test("Default initialization", grid.to_string( ));

  grid = Grid{33};
  eval.evaluate_test("Initialize to 33 robots", grid.to_string( ));

  grid = Grid{Coordinate{0, 0}};
  grid.move_player(Coordinate{-1, -1});
  eval.evaluate_test("Move off upper left of grid", grid.to_string( ));

  grid = Grid{Coordinate{MAX_X-1, MAX_Y-1}};
  grid.move_player(Coordinate{1, 1});
  eval.evaluate_test("Move off lower right of grid", grid.to_string( ));
}
```

4 tests in 12 lines of code
Additional tests become “cheap”

Step 12: Test_Grid in Makefile

- Evaluator makes our testing much more compact

```
# ... earlier lines omitted
grid: test_grid.o grid.o robot.o coordinate.o evaluator.o globals.h
        $(CXX) $(CXXFLAGS) test_grid.o grid.o robot.o coordinate.o evaluator.o
showgrid: show_grid.o grid.o robot.o coordinate.o globals.h
        $(CXX) $(CXXFLAGS) show_grid.o grid.o robot.o coordinate.o
evaluator.o: evaluator.cpp evaluator.h grid.h globals.h coordinate.h
        $(CXX) $(CXXFLAGS) -c evaluator.cpp
grid.o: grid.cpp grid.h globals.h robot.h
        $(CXX) $(CXXFLAGS) -c grid.cpp
robot: test_robot.o robot.o coordinate.o globals.h
        $(CXX) $(CXXFLAGS) test_robot.o robot.o coordinate.o
robot.o: robot.cpp robot.h globals.h coordinate.h
        $(CXX) $(CXXFLAGS) -c robot.cpp
coordinate: test_coordinate.o coordinate.o
        $(CXX) $(CXXFLAGS) test_coordinate.o coordinate.o
test_coordinate.o: coordinate.o coordinate.h
        $(CXX) $(CXXFLAGS) -c test_coordinate.cpp
coordinate.o: coordinate.cpp coordinate.h globals.h
        $(CXX) $(CXXFLAGS) -c coordinate.cpp
diff:
        for f in $$( ls ../$(PREV)/*.cpp ../$(PREV)/*.h ); do echo '######## ' $
$f ; diff $$f . ; done
clean:
        -rm -f *.o a.out
```

Step 12: Regression Test

```
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ make grid
g++ -std=c++11   -c -o test_grid.o test_grid.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 -c evaluator.cpp
g++ -std=c++11 test_grid.o grid.o robot.o coordinate.o evaluator.o
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ ./a.out < test_grid_expected.txt
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ ./a.out -v
.............................
.............................
.............................
..........X..................
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.....................X.......
..X..........................
............R................
.............................
.............................
.....X.......................
..........X.......X..........
.......X.....................
.............................
.....................X.......
.............................
.............................
          X
```

Step 12: Interactive Test

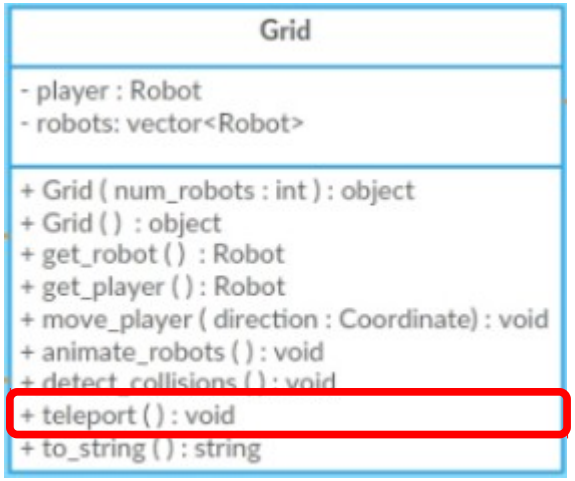
- It works!
- Awfully hard to win, though
  - We need... teleporting!

```
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ make clean
rm -f *.o a.out
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ make
g++ -std=c++11 -c main.cpp
g++ -std=c++11 -c controller.cpp
g++ -std=c++11 -c view.cpp
g++ -std=c++11 -c grid.cpp
g++ -std=c++11 -c robot.cpp
g++ -std=c++11 -c coordinate.cpp
g++ -std=c++11 main.o controller.o view.o grid.o robot.o coordinate.o
ricegf@pluto:~/dev/cpp/201701/06/12-fix_offgrid_bug$ ./a.out
============
ROVING ROBOT
============


.........................X....
............................X...
.............................
............................X...
.............................
.............................
.............................
.............................
.........X...................
.............................
.............................
.............................
.............................
..X..........................
..............R..............
...............X.............
.............................
...X..............X..........
.............................X
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.............................
.........................X.....

Ralph (14,14) command? 7
```

Step 13: Teleporting

- Literally a trivial implementation
  - To Grid, we add

```
void teleport( ) {_player =
                        Robot{_player.get_robot_id()};}
```

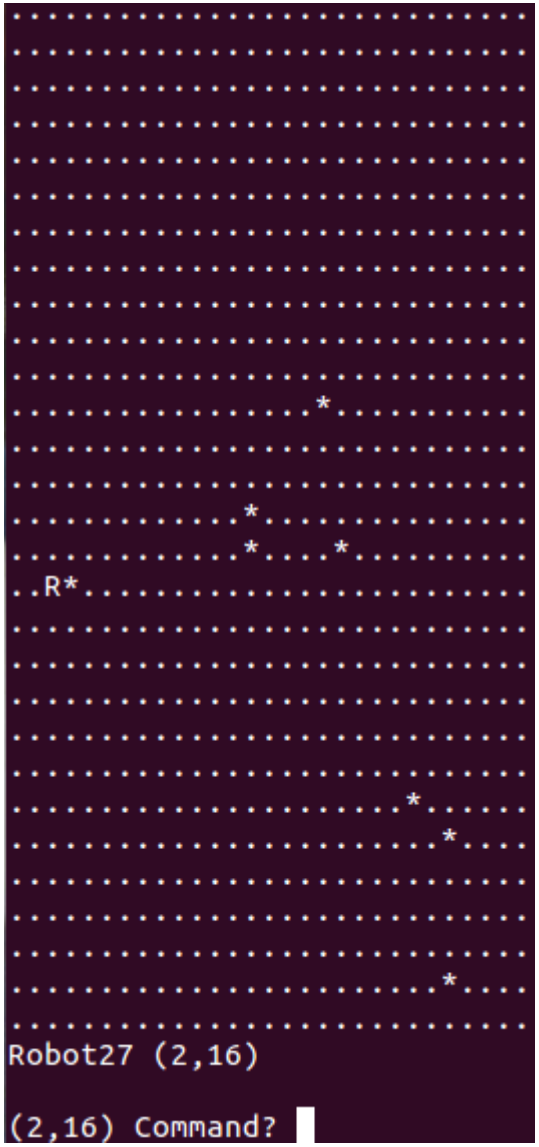


  - This just gives us a brand new player at a random location
- In Controller, we add (in the cmd handler)

```
if (cmd[0] == '.') { _grid.teleport(); }
```

Step 13: … and Test

- Now it's winnable
- But the game never ends - because it doesn't detect when all of the robots have been killed
  - Sounds like we need a Grid method that returns true if any robots are alive, and false if not



Step 14: Detecting the End

- Detecting any live robot is straightforward
  - To Grid, we add

```
bool Grid::any_robots_alive( ) {
  for (Robot robot: _robots) {
    if (robot.is_alive()) return true;
  }
  return false;
}
```

  - We exit our main loop in Controller if all robots killed

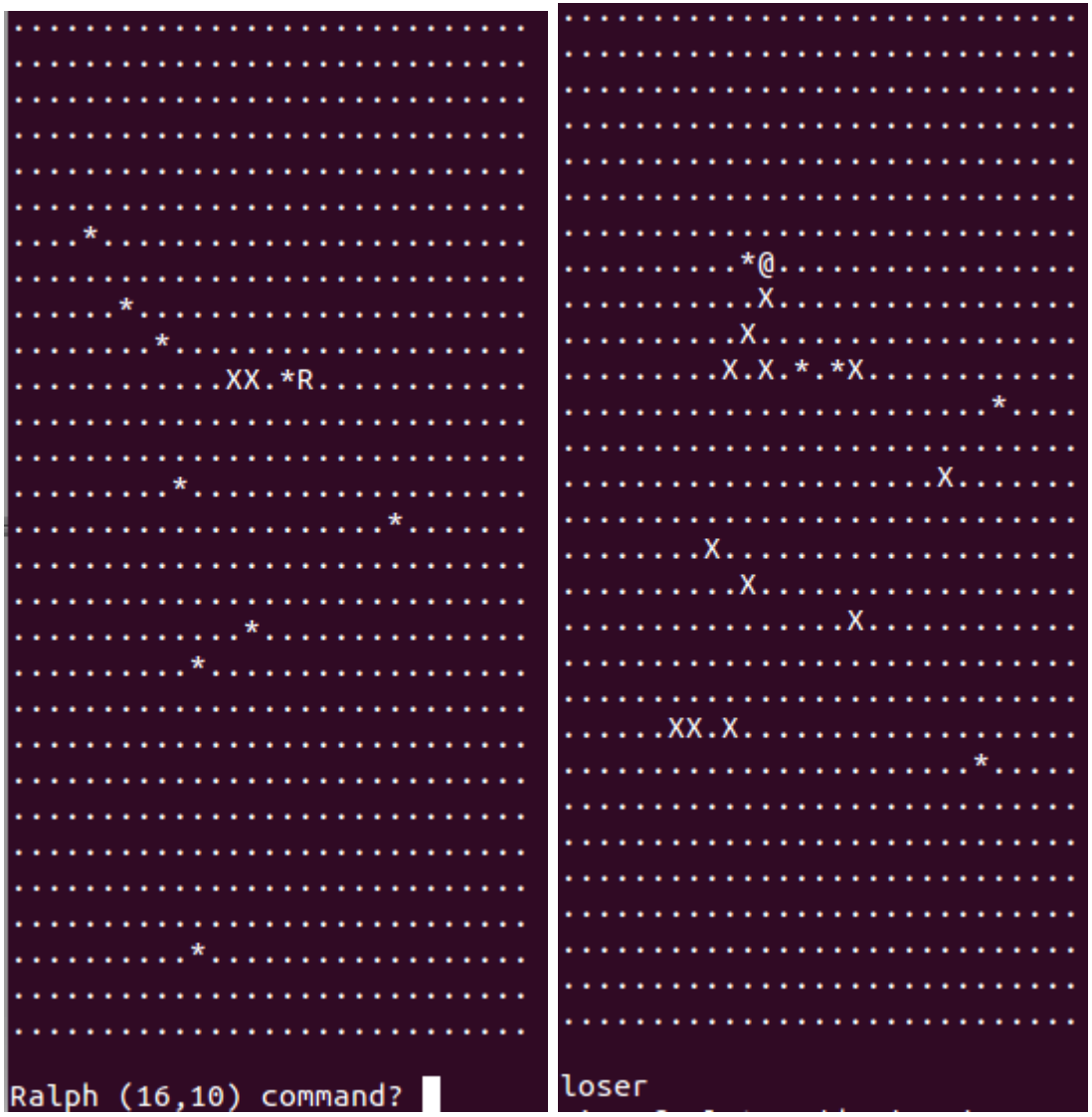
```
// was  while (cmd[0] != '0' && grid.get_player().is_alive()) {
        while (cmd[0] != '0' && grid.get_player().is_alive() &&
               grid.any_robots_alive()) {
```

  - Finally, after the main loop, we report the results

```
      if (grid.get_player().is_alive())
        cout << "### WINNER ###" << endl << endl;
      else
        cout << "loser" << endl;
```

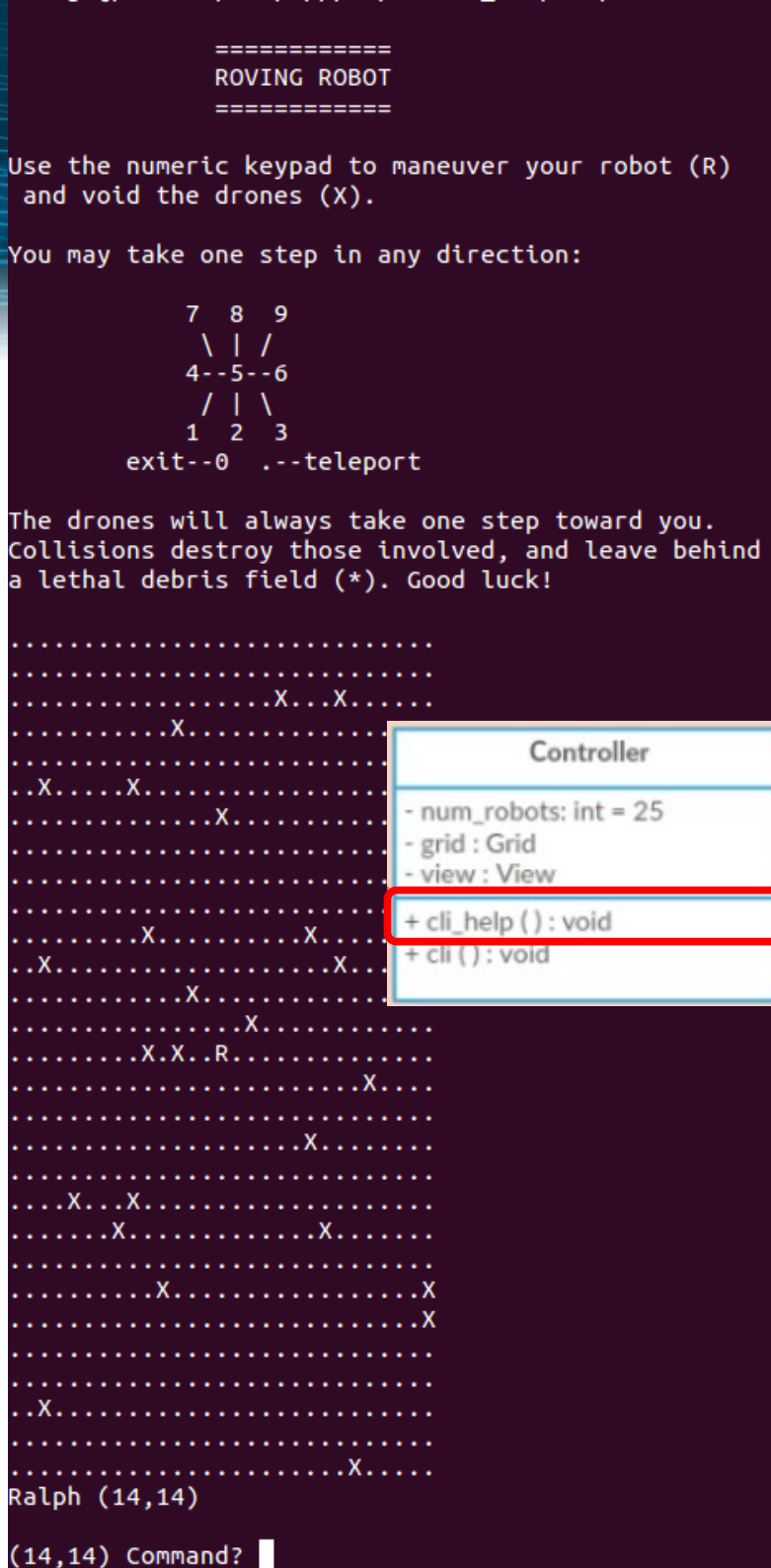
Step 14: … and Test

- Done!
- OK, NOT done
- The hardest job in programming is to *finish the details*
  - Refactoring
  - User docs
  - Packaging



Step 15: Wrapping up - Docs

- Keys to writing good docs
  - Know your goal
  - Know your audience
  - Avoid jargon
  - Keep it concise and simple
  - Clear and on-topic graphics may help a lot
  - White space is your friend
  - Provide search and an index
  - If you write poorly, hire a writer



Step 15: Help

- Detecting any live robot is straightforward
  - To View, we add concise, visual instructions

```
void View::print_help() {
  cout << "Use the numeric keypad to maneuver your robot (R)" << endl
       << " and avoid the drones (X)." << endl
       << endl
       << "You may take one step in any direction:" << endl
       << "             7  8  9" << endl
       << "              \\ | /" << endl
       << "             4--5--6" << endl
       << "              / | \\" << endl
       << "             1  2  3" << endl
       << "        exit--0  .--teleport " << endl
       << endl
       << "The drones will always take one step toward you." << endl
       << "Collisions destroy those involved, and leave behind" << endl
       << "a lethal debris field (*). Good luck!" << endl << endl;
}
```

```
void Controller::cli_help() {
    _view.print_help();
};
```

Step 15: Help

- We make one design change
  - Controller::execute_cmd returns a bool
    - True if the robots should move, false otherwise (e.g., the user presses ? for help or an invalid key)

```
bool Controller::execute_cmd(string cmd) { // Returns true if robots should move
        if (cmd[0] == '1') { _grid.move_player(Coordinate(-1, 1)); return true;}
        if (cmd[0] == '2') { _grid.move_player(Coordinate( 0, 1)); return true;}
        if (cmd[0] == '3') { _grid.move_player(Coordinate( 1, 1)); return true;}
        if (cmd[0] == '4') { _grid.move_player(Coordinate(-1, 0)); return true;}
        if (cmd[0] == '5') { _grid.move_player(Coordinate( 0, 0)); return true;}
        if (cmd[0] == '6') { _grid.move_player(Coordinate( 1, 0)); return true;}
        if (cmd[0] == '7') { _grid.move_player(Coordinate(-1,-1)); return true;}
        if (cmd[0] == '8') { _grid.move_player(Coordinate( 0,-1)); return true;}
        if (cmd[0] == '9') { _grid.move_player(Coordinate( 1,-1)); return true;}
        if (cmd[0] == '.') { _grid.teleport(); return true;}
        if (cmd[0] == '?') { cli_help(); }
        return false;
};
```

Step 15: Help

- And adapt Controller::cli to match

```
void Controller::cli() {
    _view.print_banner();
    cli_help();
    string cmd = "x";
    while (cmd[0] != '0' && _grid.get_player().is_alive() &&
           _grid.any_robots_alive()) {
        _view.print_grid();
        cout << _grid.get_player().get_robot_id() << " "
             << _grid.get_player().get_coordinate().to_string()
             << " command? ";
        cin >> cmd;
        if (execute_cmd(cmd)) {
          _grid.animate_robots();
          _grid.detect_collisions();
        }
    }
    _view.print_grid();
    if (_grid.get_player().is_alive())
      cout << "### WINNER ###" << endl << endl;
    else
      cout << "loser" << endl;
};
```

Final Code

- We adjust the number of robots to 25 based on experience
- 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”)

```
ricegf@pluto:~/dev/cpp/201701/06/15-add_help$ ./a.out
============
ROVING ROBOT
============

Use the numeric keypad to maneuver your robot (R)
 and avoid the drones (X).

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

The drones 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...........X..
......X....................
..........................
..........................
......................X.....
..........................
..........................
..........................
..........X.XX.............
..................X.........
..........................
......X....................
..................X.........
.....X......X..............
..................X.........

Ralph (14,14) command?
```

Step 15: Wrapping Up - Packaging

- Installation packaging varies by OS
  - For Windows, .msi or a setup.exe is common
    - Commercial tools are best – Visual Studio includes one
    - The Windows Store is... sparse
  - Mac OS X is somewhat similar to Windows
    - Mac supports flat, meta-, distribution, and hybrid packages that work on multiple OS versions
  - For Linux, snap packages are the latest thing
    - Legacy systems use Red Hat Package Manager (.rpm) or Debian (.deb); setup.exe equivalents (.tgz) are still sometimes used
    - Repositories of packages (“repos”) are usually massive with >10,000 programs available to install
  - Enterprise package management solutions are commonly deployed in the corporate world for patch and app deployment

Quick Review

- The _____ diagram is the most common UML interaction diagram. On this diagram, the Y direction represents time.
- A disproportionate number of bugs are usually found from _____ _____.
- _____ is rewriting existing code for better maintainability without adding new functionality.
- Good documentation is always written from the perspective of and to meet the needs of the _____.

Lecture #6
Quick Review

- A series of interactions that enable an actor to achieve a goal is a **use case**.
- True or false: In UML, a use case is treated as a class. **True**
- The pattern that separates the business logic, data visualization, and human or machine user is the **Model** – **View** – **Controller** (**MVC**) pattern.
- An object-oriented design is usually implemented starting with the class with the fewest **dependencies**. We implement a small set of functionality along with a **regression test**. A **test framework** can make this task easier.
- Code for which specified assertions are guaranteed to be true is **invariant**.
- To retain the value of a method-level variable between calls to that method, we declare the variable to be **static**.
- Rather than a to_string method, a more C++-like approach to providing a string representation of an object is to overload the **stream operator**.

Lecture #7

Quick Review

- The **sequence** diagram is the most common UML interaction diagram. On this diagram, the Y direction represents time.
- A disproportionate number of bugs are usually found from **edge cases**.
- **Refactoring** is rewriting existing code for better maintainability without adding new functionality.
- Good documentation is always written from the perspective of and to meet the needs of the **user**.

For Next Class

- **Exam #1** (20% of final grade)
- Study sheet and practice exam are on Blackboard

Questions?