Replace Error Code With Exception

Refactor by replacing **error codes** with **exceptions** to simplify error handling and improve readability.

- Error code, or returning null, is an easy way to express error.
- However, it makes code hard to debug.

Error code also requires **explicit checks** after every operation, makes code **cluttered** with error-handling logic, and can be **easily ignored**, leading to silent failures.

- In this case, we can refactor the code to use an exception.

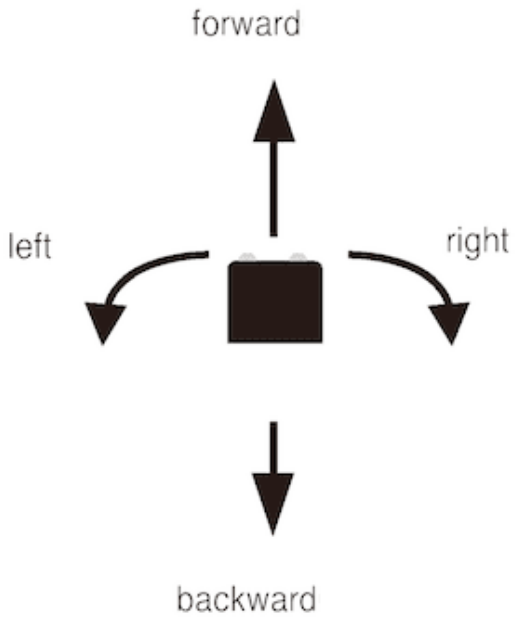
```
public static Command parseCommand(String name) {
    if (!_commandNameMap.containsKey(name)) {
        return null;
    }
    return _commandNameMap.get(name);
}
```



```
public static Command parseCommand(String name) throws InvalidCommandException {
    if (!_commandNameMap.containsKey(name)) {
        throw new InvalidCommandException(name);
    }
    return _commandNameMap.get(name);
}
```

Example: Robot

- We use command design pattern to control robot.



Command class

- The command class uses a dictionary to map a string to command object.

```
class Command:
    def __init__(self, name: str):
        self._name = name

    @staticmethod
    def parse_command(name: str) -> Optional['Command']:
        return Command._command_name_map.get(name)

    Command._command_name_map: Dict[str, Command] = {
      "forward": Command.FORWARD,
      "backward": Command.BACKWARD,
      "right": Command.TURN_RIGHT,
      "left": Command.TURN_LEFT
    }
```

- Each command is associated with a Command object.

```
# Create command constants
Command.FORWARD = Command("forward")
Command.BACKWARD = Command("backward")
Command.TURN_RIGHT = Command("right")
Command.TURN_LEFT = Command("left")
```

Direction class

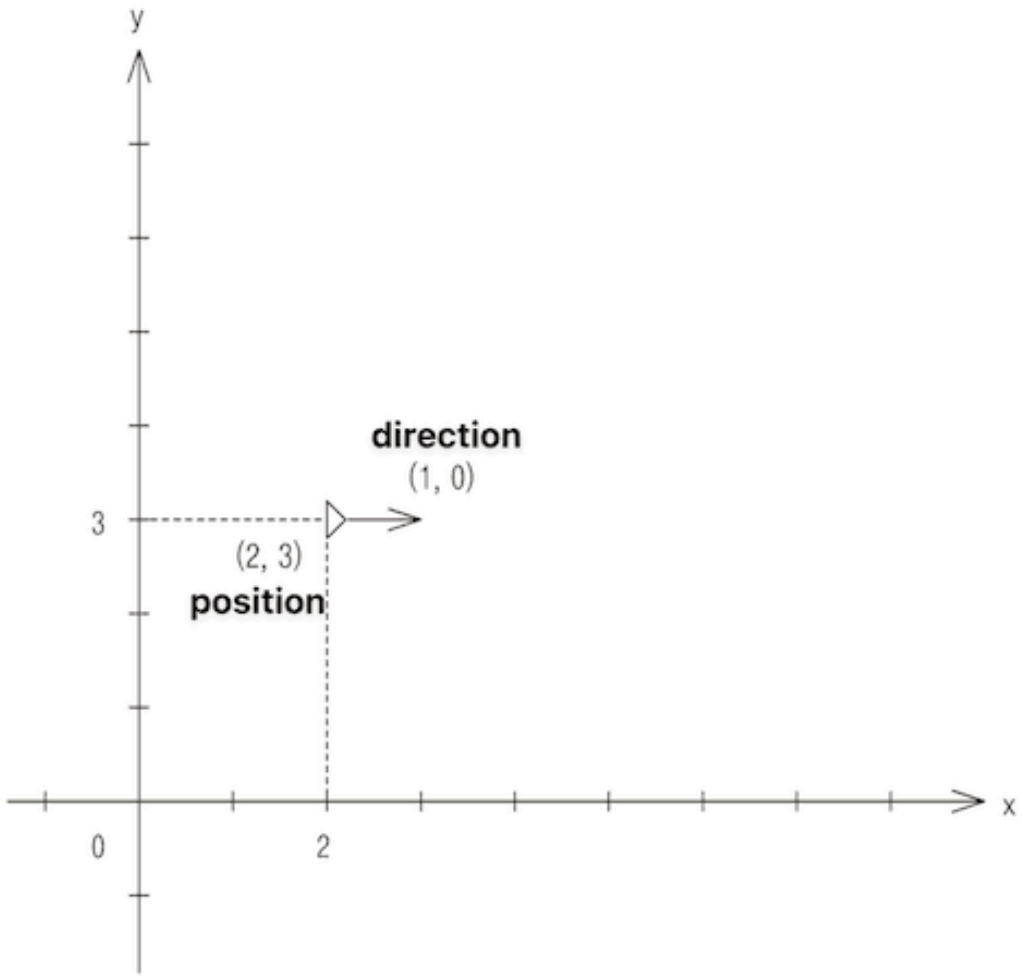
- This class specifies the direction of a robot.

```
class Direction:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def set_direction(self, x: int, y: int):
        self.x = x
        self.y = y
```

Position class

- This class specifies the position of a robot.

```
class Position:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def relative_move(self, dx: int, dy: int):
        self.x += dx
        self.y += dy
```


y
direction
(1, 0)
3
(2, 3)
position
x
0
2

Robot class

- A robot has a name, position, and direction.

```
class Robot:
    def __init__(self, name: str):
        self.name = name
        self.position = Position(0, 0)
        self.direction = Direction(0, 1)
```

- It executes a sequence of commands.

```
def execute(self, command_sequence: str):
    tokens = command_sequence.split()
    for token in tokens:
        if not self.execute_command(token):
            print(f"Invalid command: {token}")
            break
def execute_command(self, command_string: str) -> bool:
    if command is None: return False
    return self._execute_command(command)
```

- The core of execution is changing position or direction from the command.

```
def _execute_command(self, command: Command) -> bool:
    """Execute command object, returns False on error"""
    if command == Command.FORWARD:
        self.position.relative_move(self.direction.x, self.direction.y)
    elif command == Command.BACKWARD:
        self.position.relative_move(-self.direction.x, -self.direction.y)
    elif command == Command.TURN_RIGHT:
        self.direction.set_direction(self.direction.y, -self.direction.x)
    elif command == Command.TURN_LEFT:
        self.direction.set_direction(-self.direction.y, self.direction.x)
    else: return False
    return True
```

```python
def main():
    robot = Robot("Andrew")
    print(robot)
    robot.execute("forward right forward")
    print(robot)
    robot.execute("left backward left forward")
    print(robot)
    robot.execute("right forward forward farvard")  # 'farvard' is invalid
    print(robot)
```

```
[ Robot: Andrew position(0, 0), direction(0, 1) ]
[ Robot: Andrew position(1, 1), direction(1, 0) ]
[ Robot: Andrew position(0, 0), direction(-1, 0) ]
Invalid command: farvard
[ Robot: Andrew position(0, 2), direction(0, 1) ]
```

Code Smell

- However, this code has many code smells.
- It returns True or False to be processed somewhere in the code to propagate the error.

```
def parse_command(name: str):
    return ...
def execute_command(self, command_string: str) -> bool:
    if command is None: return False
def _execute_command(self, command: Command) -> bool:
    else: return False
```

Unit Tests

- This program has multiple modules, so we need to make unit tests for each module.
- There are many ways to run the unit tests, but using `pytest` is the easiest.

```
pytest test/ -v
```

Refactoring

- The first step is to make an Exception class.

```
class InvalidCommandException(Exception):
    def __init__(self, message: str = ""):
        super().__init__(message)
        self.message = message
```

- Instead of returning a value, we need to raise an exception.

```
class Command:
    @staticmethod
    def parse_command(name: str) -> 'Command':
        if name not in Command._command_name_map:
            raise InvalidCommandException(name)
        return Command._command_name_map[name]
```

- We catch the raised exception using try ... catch.

```
from Position import Position
from Direction import Direction
from Command import Command
from InvalidCommandException import InvalidCommandException

class Robot:
    def execute(self, command_sequence: str):
        try:
            for token in tokens:
                self.execute_command(token)
        except InvalidCommandException as e:
            print(f"Invalid command: {e.message}")
    def _execute_command(self, command: Command):
        ...
        else: raise InvalidCommandException()
```

Unit tests update

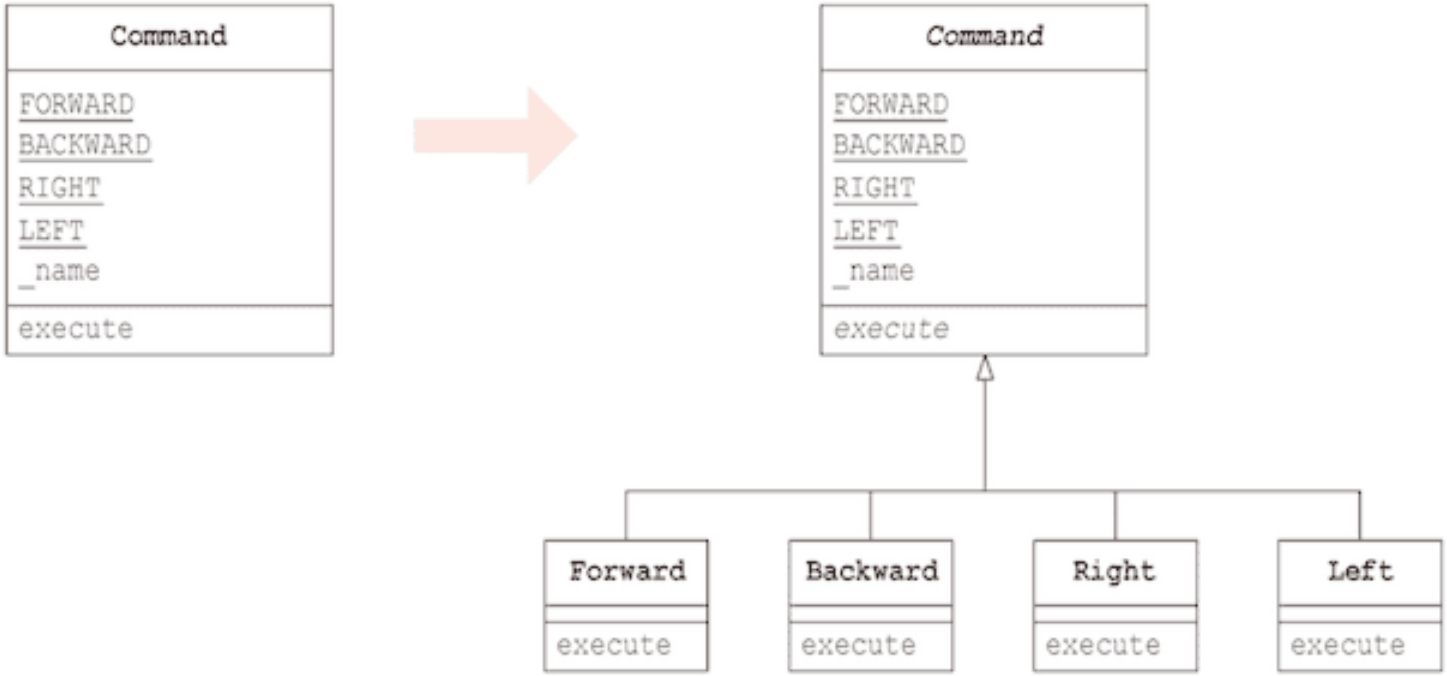
- In the unit tests, we should catch the `InvalidCommandException`.

```
with self.assertRaises(InvalidCommandException) as context:
    robot.execute_command("invalid")
```

Refactoring: Remove Type Code

- The refactored code still has a code smell: type code.
- We need to remove the if statement.

```
def _execute_command(self, command: Command):
    if command == Command.FORWARD:
        self.position.relative_move(self.direction.x, self.direction.y)
    elif command == Command.BACKWARD:
        self.position.relative_move(-self.direction.x, -self.direction.y)
```


Command
FORWARD
BACKWARD
RIGHT
LEFT
_name
execute
Command
FORWARD
BACKWARD
RIGHT
LEFT
_name
execute
Forward
execute
Backward
execute
Right
execute
Left
execute

Polymorphism

- Instead of using type code, we subclass Command class to make sub commands.

```
class Command:
    def execute(self, robot) -> None: pass

class Forward(Command):
    def __init__(self): super().__init__("forward")
    def execute(self, robot): robot.forward()
Command._command_name_map: Dict[str, Command] = {
    "forward": Forward(),
    "backward": Backward(),
    "right": Right(),
    "left": Left()
}
```

- The Robot class is updated to support commands for the sub classes.

```
class Robot:
    def execute_command(self, command_string: str):
        command = Command.parse_command(command_string)
        command.execute(self)

    def forward(self):
        self.position.relative_move(self.direction.x, self.direction.y)
    ...
    def left(self):
        self.direction.set_direction(-self.direction.y, self.direction.x)
```

Unit tests update

- `assertIs(a, b)` checks if `a` and `b` are the exact same object in memory (i.e., `a is b`).
  - `assertEqual(a, b)` checks if `a` and `b` are equal in value (i.e., `a == b`), which depends on the `__eq__` method implementation.

- So, this test returns false.

```
def test_command_identity(self):
    self.assertIs(
        Command.parse_command("forward"),
        Forward())
    self.assertIs(
        Command.parse_command("left"),
        Left())
```

- We should check the type identity instead.

```
def test_command_type_identity(self):
    self.assertEqual(
        type(Command.parse_command("forward")),
        Forward)
    self.assertEqual(
        type(Command.parse_command("left")),
        Left)
```

Tip

Checked and unchecked Exceptions

- Java has two types of exceptions:
  - Checked exceptions: Must be handled or declared (e.g., IOException)
  - Unchecked exceptions: Runtime exceptions you can choose to handle (e.g., NullPointerException)

```java
// This WON'T compile
// without try-catch or throws declaration
public void readFile() {
    // Compile error!
    FileReader file = new FileReader("data.txt");
}

// Must handle or declare
public void readFile() throws IOException {
    // OK
    FileReader file = new FileReader("data.txt");
}
```

- Python has only unchecked exceptions.
- All exceptions in Python are equivalent to Java's unchecked exceptions - you're never forced to handle them at compile time.

```
# This compiles fine,
# even though it might raise FileNotFoundError
def read_file():
    # No forced error handling
    with open("data.txt", "r") as file:
        return file.read()

# You CAN handle it, but you're not required to
def safe_read_file():
    try:
        with open("data.txt", "r") as file:
            return file.read()
    except FileNotFoundError:
        return "File not found"
```

Discussion

Benefits of Exceptions

1. **Cleaner code** - separates normal flow from error handling