Ollscoil Teicneolaíochta an Oirdheiscirt

South East Technological University

Command Pattern

Presented by Jack Sinnott and Adrien Dudon.

Authors and Presenters

Jack Sinnott

Adrien Dudon

Brief Description

- Behavioural design pattern
  - *Concerned with algorithms and assignment of responsibilities between objects*
- Encapsulates an object with all information needed to perform an action or trigger at a later time.
- Commands are object-oriented replacements for **callback functions**.

Surprise Question

What is a callback?

Callback - Brief Description

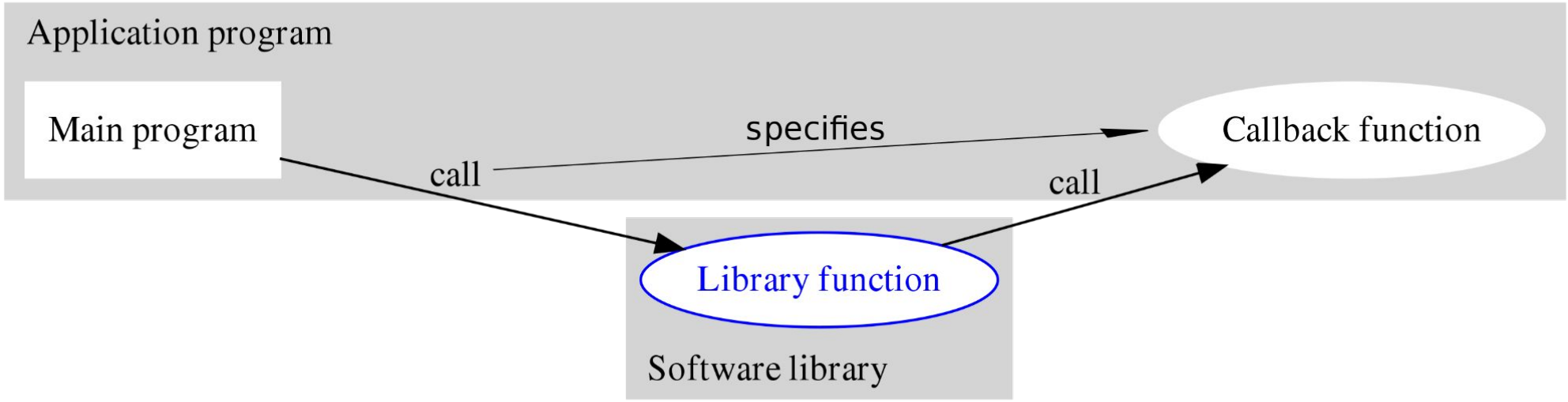
What is a callback?

- Function passed as argument to another piece of code (like another function), also called a **function pointer** in C++
- This other piece of code will **call back** (*execute*) the function passed as parameter
- The function parameter is not called directly and only defined, then, called circumstantially

Generally used for asynchronous programming.

More information: https://en.wikipedia.org/wiki/Callback_(computer_programming)

Callback - Diagram

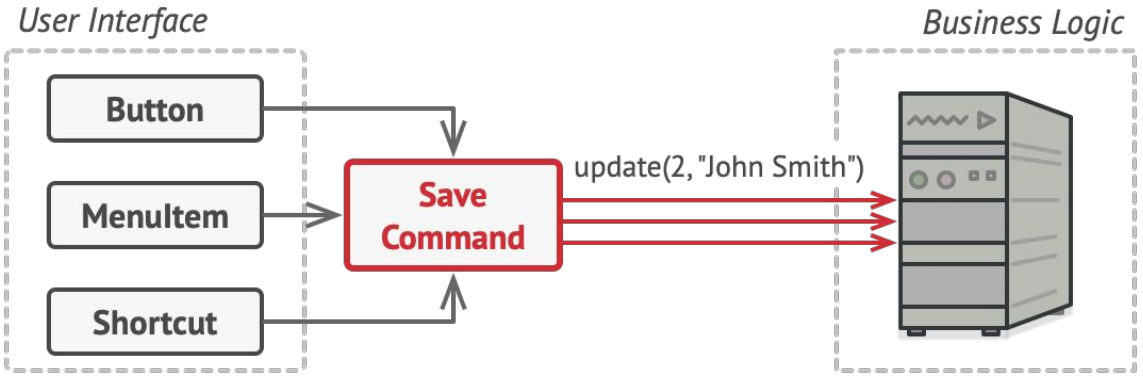


Callback - Python example

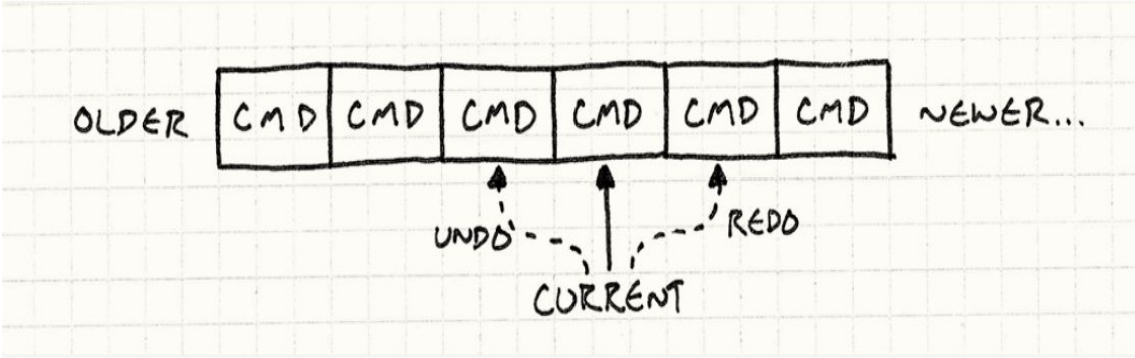
```python
def get_square(val):
    """The callback."""
    print("get_square() called back")
    return val ** 2

def caller(func, val):
    print("Execute callback")
    return func(val) # Then call back the function passed as parameter

if __name__ == '__main__':
    caller(get_square, 5)

# Output
# > "Execute callback"
# > "get_square() called back"
# > 25
```

Purpose

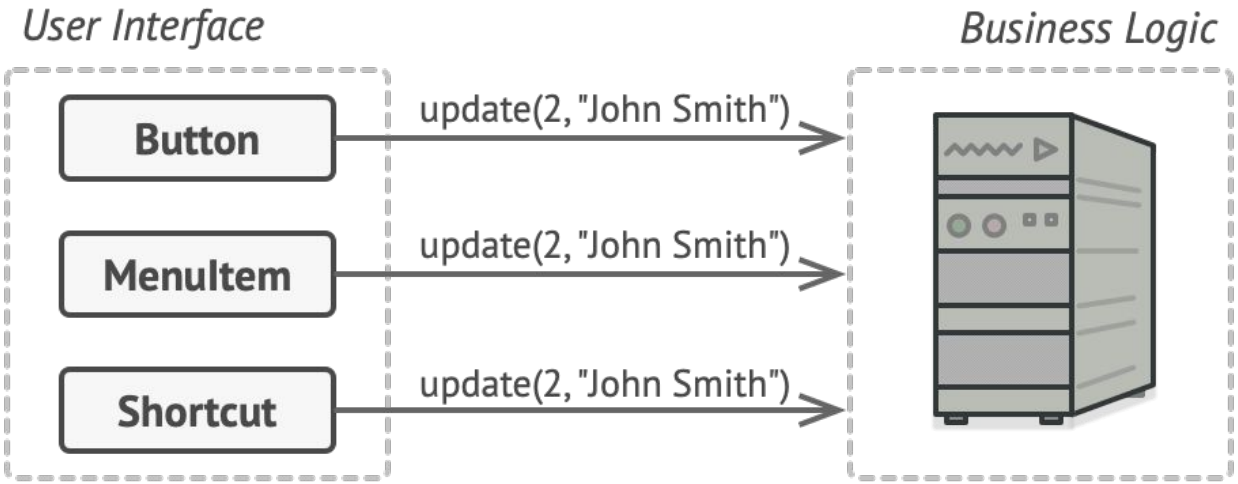
- Decouples classes that invoke an operation from the object that knows how to execute the operation



- Allows creation of a sequence of commands by providing a queue system



Problem - Saving App Data

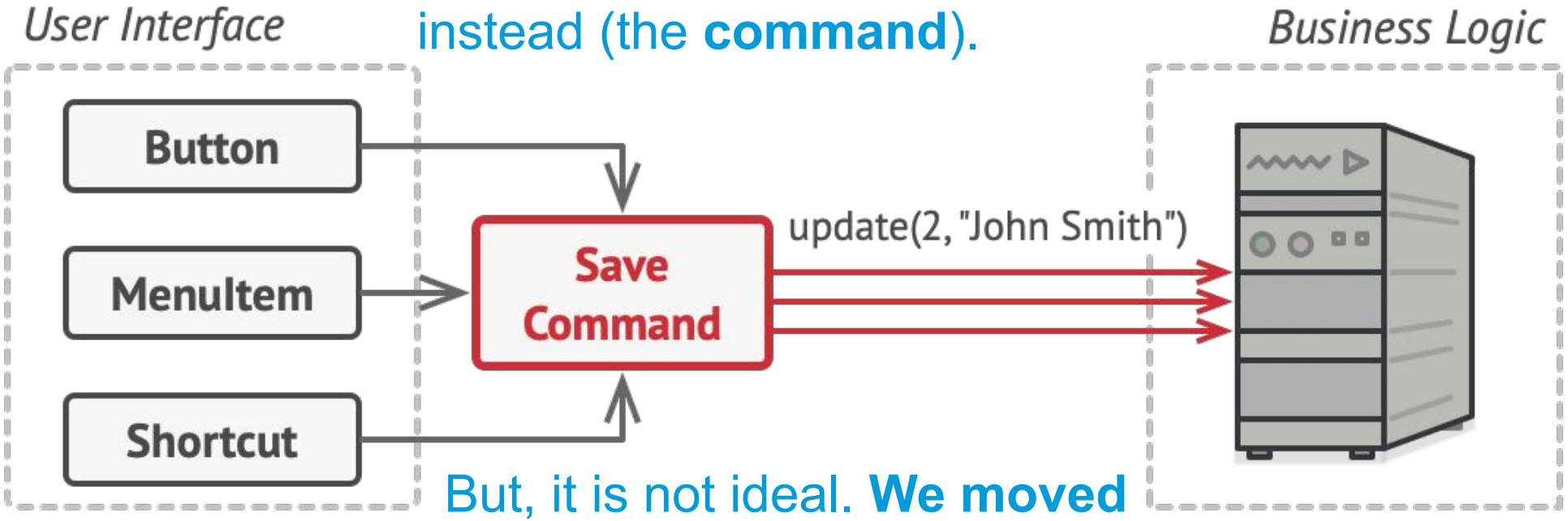


Question: **why is it bad?**

The update() function is called from everywhere.
Code repetition is bad, and *hard-to-maintain*.

Saving App Data - Solution?

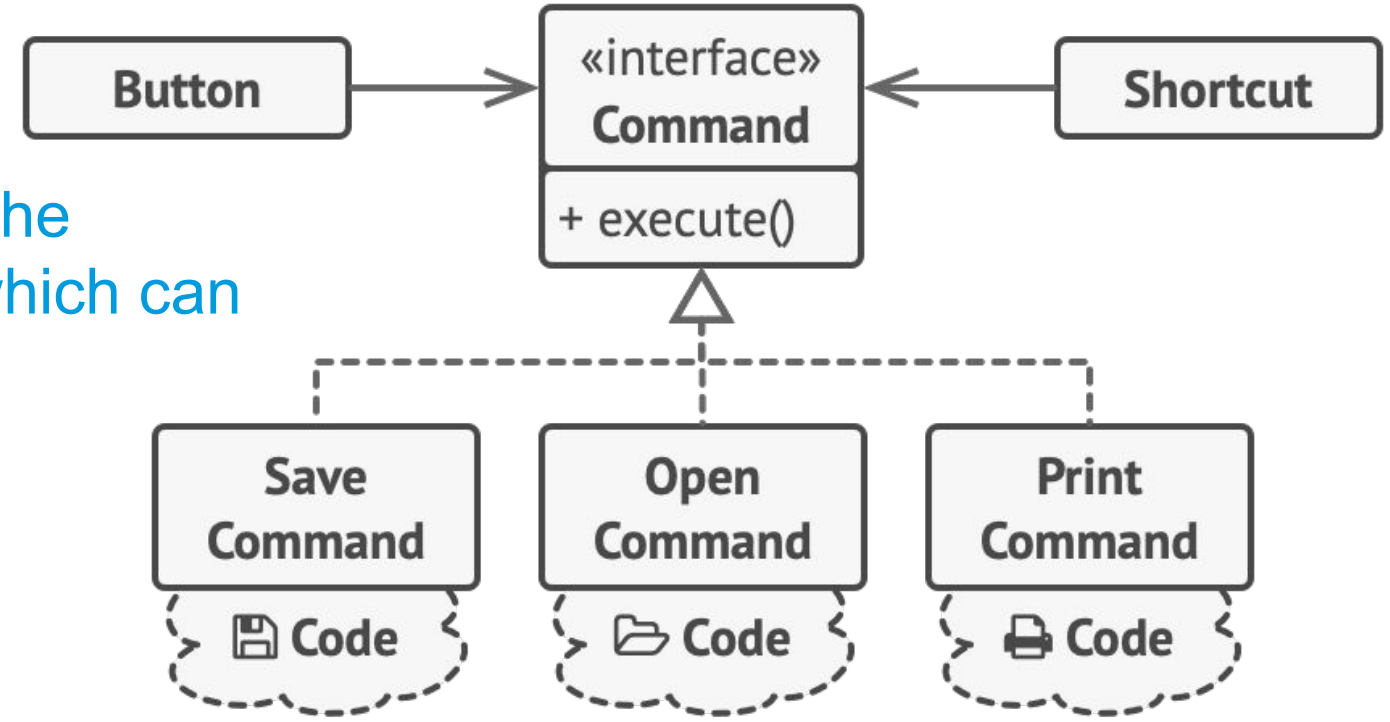
Better: call a wrapper class instead (the **command**).



But, it is not ideal. **We moved the problem.** The Command cannot be easily changed.

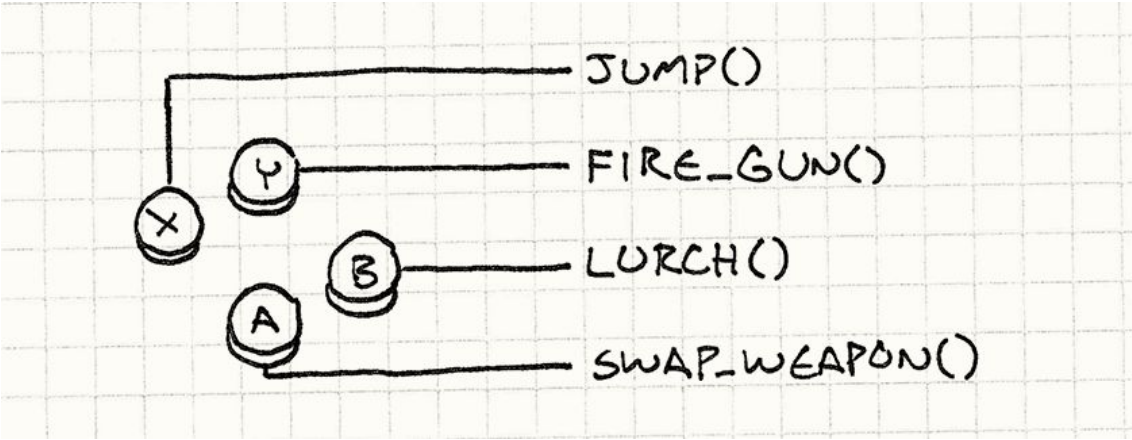
Saving App Data - Better solution!

Each button now calls the Interface **Command**, which can *bind* to any command.



Problem 2 - Game Inputs

- Bunch of methods called directly




```
void handleInput(ControllerInput input)
{
  switch (input)
  {
    case ControllerInput.X:
      Jump();
      break;
    case ControllerInput.Y:
      FireGun();
      break;
    case ControllerInput.A:
      SwapWeapon();
      break;
    case ControllerInput.B:
      Lurch();
      break;
  }
}
```

Problem: each inputs are hard-mapped to an action.

Game Inputs

```
def handle_input():
  if controller.x.is_pressed:
    if player.check_if_grounded():
      player.jump()
      start_jump_animation()
      # ...

  elif controller.y.is_pressed:
    player.fire_gun()
    # ...
  # ...
```

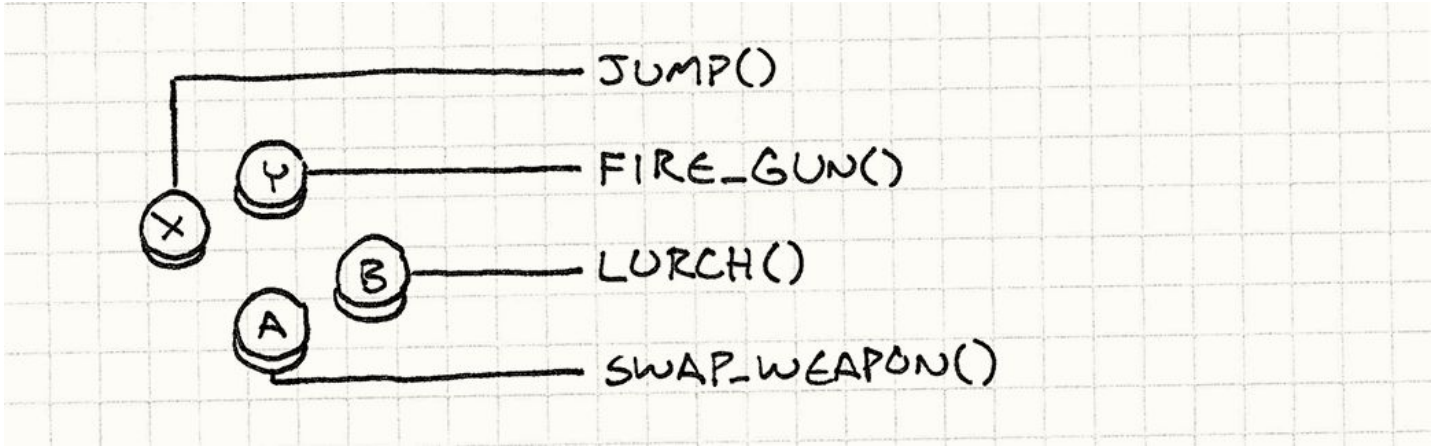
Directly calling implementation logic

=

But, why?

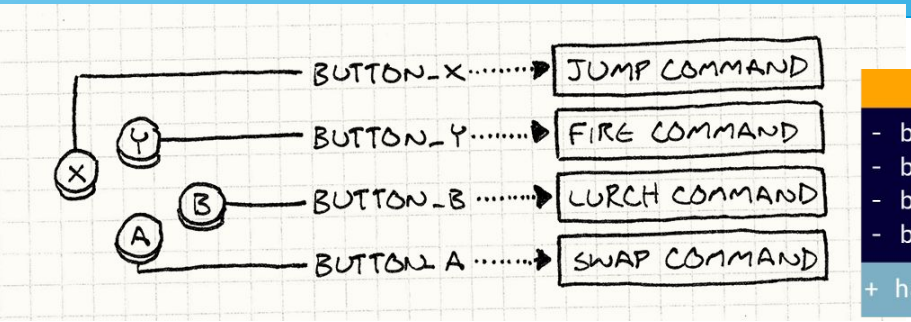
Each input is hard-wired to its implementation. You cannot easily change the binding at runtime (**bad**).

Game Inputs



- What if “Jump” is **more** than just calling a jump() method?
- What if we want the player to be able to **rebind** the fire action ?

Game Inputs - Solution



Real-world analogies

Game development

- Unity use the Command Pattern for their **new Input System**
- Queueing actions in the Sims or similar games

Application/Software Development

- Software like Photoshop lets you cancel previous actions (**CTRL+Z**). Command Pattern works great in this case!

Pros

Advantages of using Command

- **Loose Coupling:** Command pattern loosely couples an object that invokes an operation & the object that performs the operation.

- **Undo/Redo:** Command pattern provides the ability to easily create a undo/redo feature, using a queue system.

- **Extensibility:** Adding a new command is easy & doesn't require changing existing code!

Cons

Issues associated with Pattern

- **Each Command is in its own class:** This can lead to difficulty maintaining the codebase when there is many Commands

- Every Individual command is a ConcreteCommand class that increases the volume of classes for Implementation and maintenance.

Implementation Instructions

Back to our Game Inputs problem

- Let's see how to properly implement the Command System in the case of an Input System for a game.

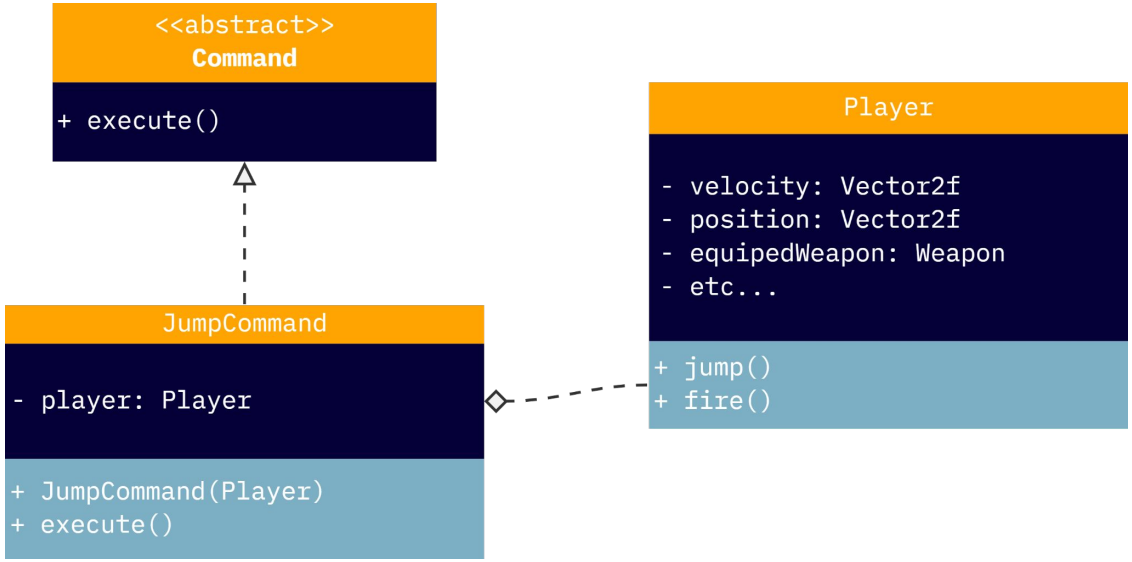
- Let's use C++, because it can be easily translated into higher level language if needed.

Code Implementation Example

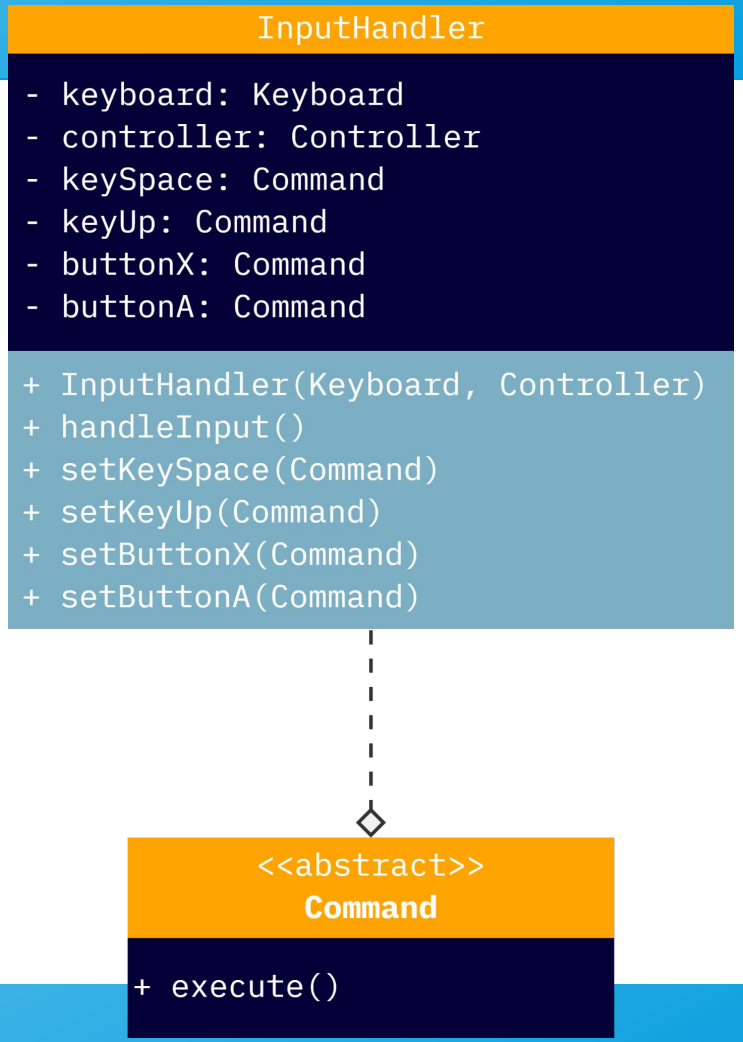
```
class Command // Pure Virtual
{lass
  virtual void execute() = 0;
}
```

Code Implementation Example

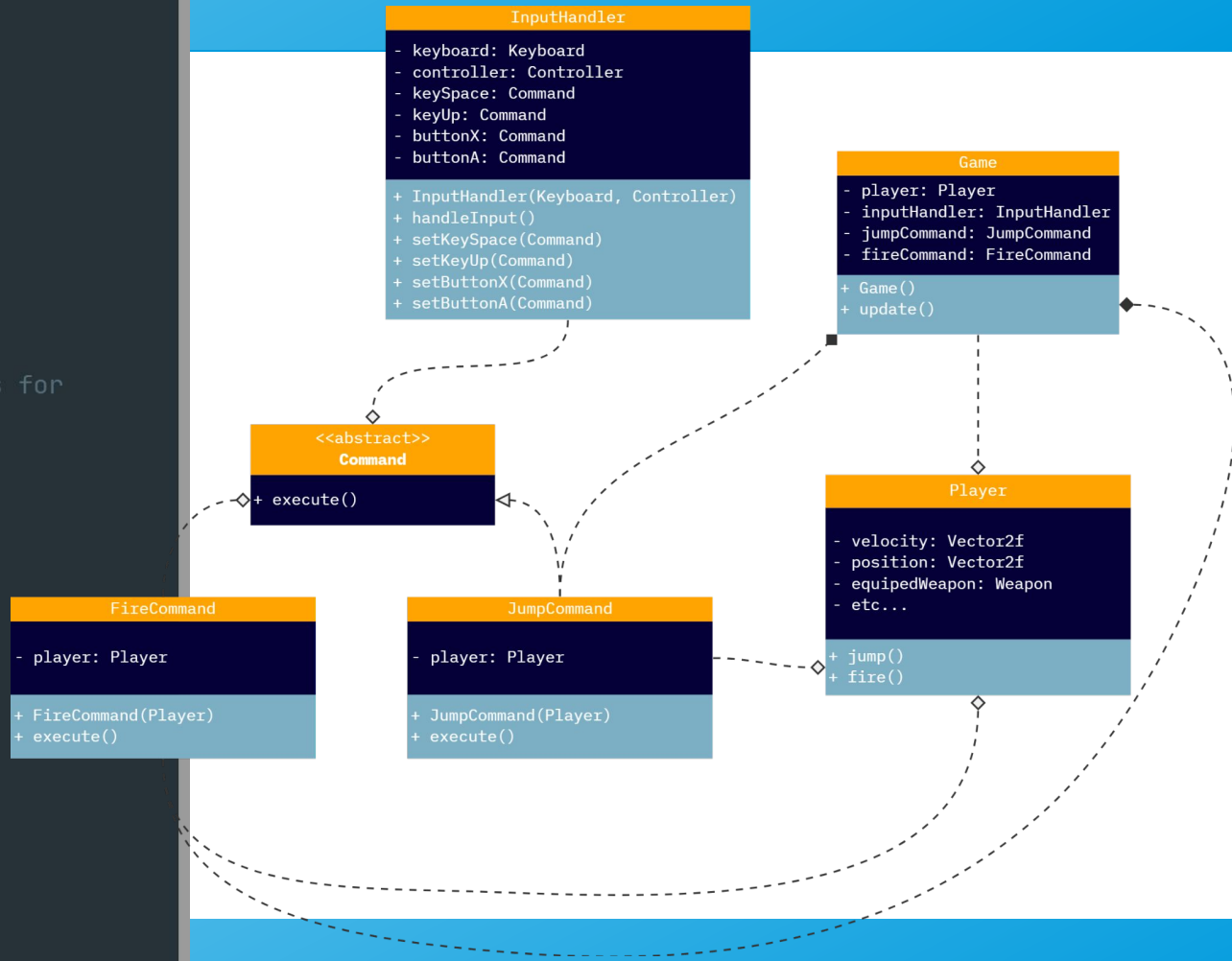
```
class JumpCommand : public Command
{
public:
    JumpCommand(const Player& t_player) :
        m_player(t_player)
    {
    }

    void execute() override
    {
        m_player.jump();
        m_player.starJumpAnimation();
    }

private:
    Player m_player;
}
```



```cpp
class InputHandler
{
public:
    InputHandler(const Keyboard& t_keyboard, const Controller& t_controller)
:       m_keyboard(t_keyboard),
        m_controller(t_controller)
    {
    }

    void setKeySpace(const Command& t_command)
    {
      m_keySpace = t_command;
    }

    void setKeyUp(const Command& t_command)
    {
      m_keyUp = t_command;
    }

    void setButtonX(const Command& t_command)
    {
      m_buttonX = t_command;
    }

    void setButtonA(const Command& t_command)
    {
      m_buttonA = t_command;
    }

private:
    Keyboard m_keyboard;
    Controller m_controller;

    Command m_keySpace;
    Command m_keyUp;

    Command m_buttonX;
    Command m_buttonA;

    // etc...
}
```



```cpp
class Game
{
public:
  Game()
  {
    m_jumpCommand = new JumpCommand(m_player);
    m_fireCommand = new FireCommand(m_player);

    m_inputHandler->setButtonX(&m_jumpCommand);
    m_inputHandler->setKeySpace(&m_jumpCommand);

    // Command can be changed at runtime, now X is for
firem_inputHandler->setButtonX(&m_fireCommand);
  }

  void update()
  {
    while (true)
    {
      m_inputHandler->handleInputs();
    }
  }

private:
  Player m_player;
  InputHandler* m_inputHandler;

  JumpCommand* m_jumpCommand;
  FireCommand* m_jumpCommand;
}
```



Sources

- https://refactoring.guru/design-patterns/command
- https://gameprogrammingpatterns.com/command.html
- https://www.cs.unc.edu/~stotts/GOF/hires/pat5bfso.htm