Game Programming Patterns

The Command

Game Programming Patterns

The Command Design Pattern

- It’s a behavioral design pattern
- An **object** is used to **encapsulate** all **information needed** to perform an **action** or trigger an **event** at a **later time**
- In game development:
    - Useful when **dealing** with **raw user input**
    - To create **undo/redo functionality**

Description

Encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.

Design Patterns: Elements of Reusable Object-Oriented Software

Command Design Pattern

Example: User Input

- Every game has some kind of code to read raw user input
- A simple implementation looks like:

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();
  else if (isPressed(BUTTON_Y)) fireGun();
  else if (isPressed(BUTTON_A)) swapWeapon();
  else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

- **What if we want to allow the user to configure button mapping?**

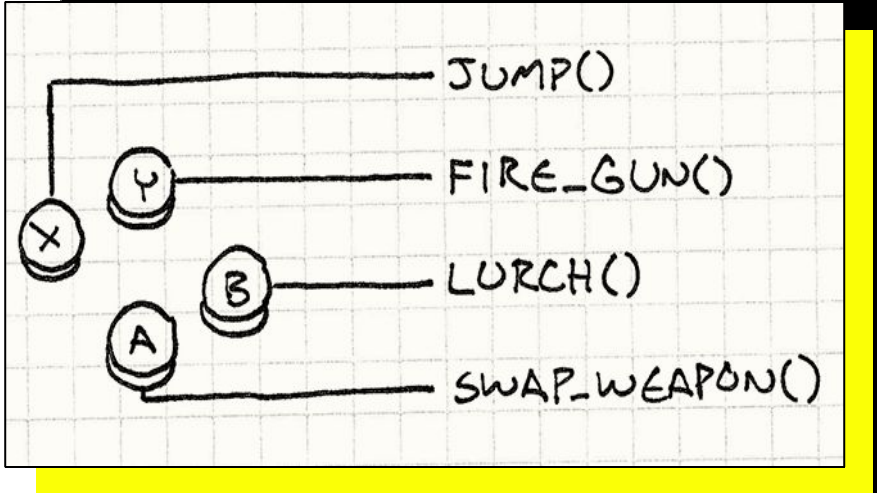


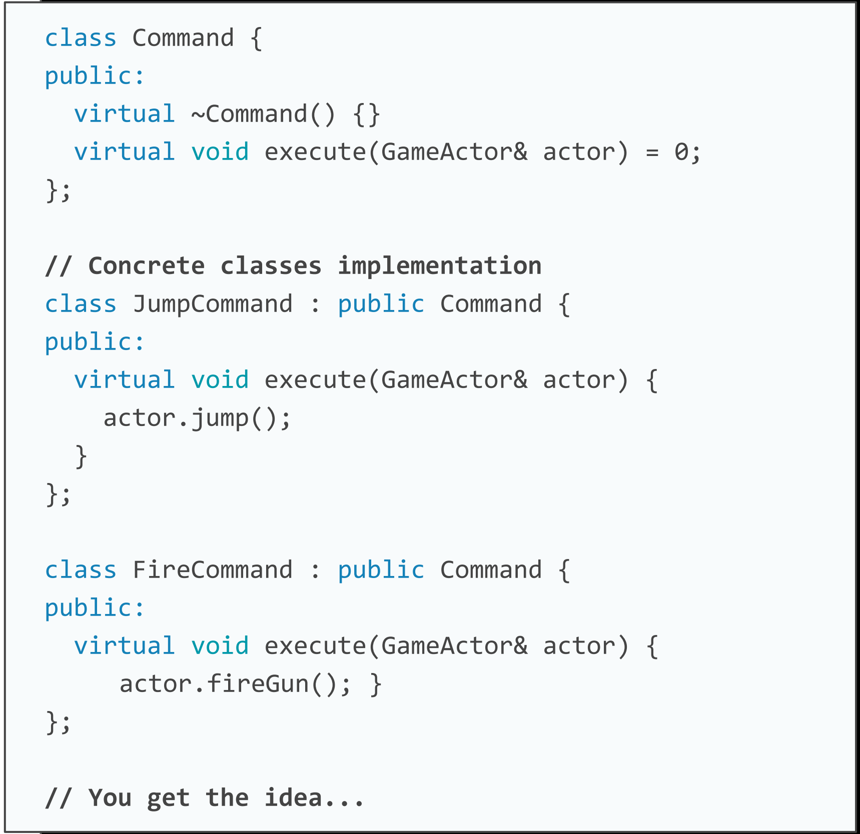
Illustration of common button mappings in a video game.

From: Robert Nystrom; “Game Programming Patterns”

Command Design Pattern

Applying the Pattern to a Base Class

- We start by defining a **base class** that represents a **triggerable game command**
- Then create the **concrete classes** for each **command**



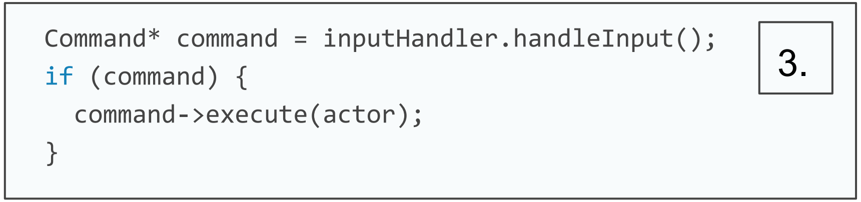
```
class Command {
public:
  virtual ~Command() {}
  virtual void execute(GameActor& actor) = 0;
};

// Concrete classes implementation
class JumpCommand : public Command {
public:
  virtual void execute(GameActor& actor) {
    actor.jump();
  }
};

class FireCommand : public Command {
public:
  virtual void execute(GameActor& actor) {
     actor.fireGun(); }
};

// You get the idea...
```

x4

Command Design Pattern

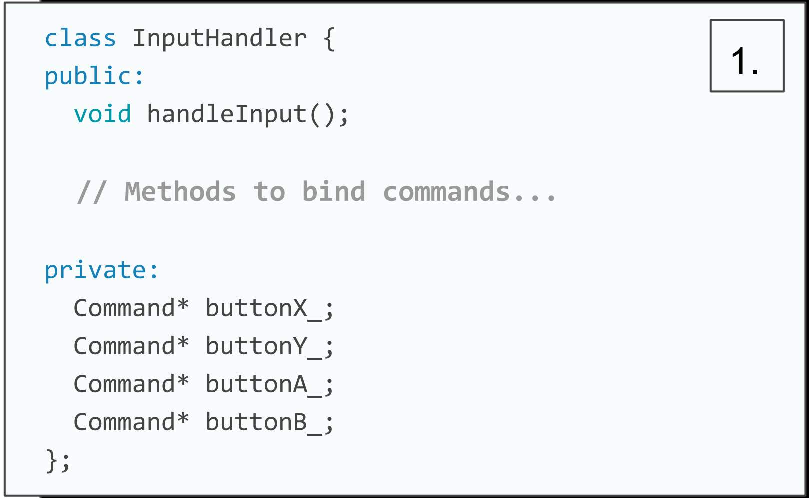
Creating the Input Handler

1. Our **input handler** stores a **pointer** to a ***Command*** for **each button**
2. Then our ***handleInput()*** method just **delegates to those pointers**
3. Finally, we can **check** for **input**. If positive, the **correspondent action** will be **executed**
    a. With this **layer of indirection**, between *Command* and *Actor*, the **player can** easily **control any Actor**



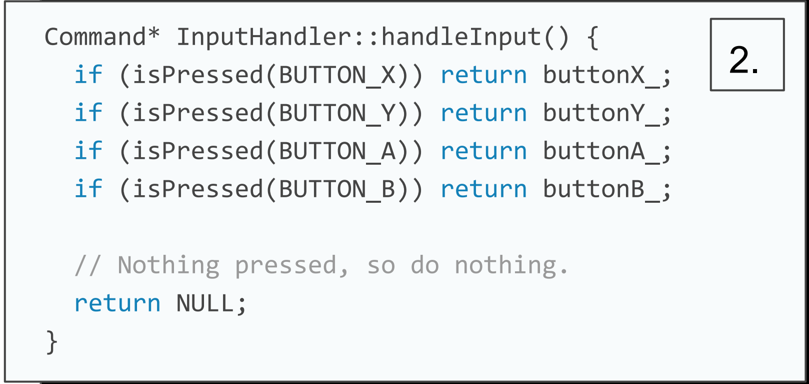
```
Command* command = inputHandler.handleInput();
if (command) {
  command->execute(actor);
}
```

3.



```
class InputHandler {
public:
  void handleInput();

  // Methods to bind commands...

private:
  Command* buttonX_;
  Command* buttonY_;
  Command* buttonA_;
  Command* buttonB_;
};
```

1.



```
Command* InputHandler::handleInput() {
  if (isPressed(BUTTON_X)) return buttonX_;
  if (isPressed(BUTTON_Y)) return buttonY_;
  if (isPressed(BUTTON_A)) return buttonA_;
  if (isPressed(BUTTON_B)) return buttonB_;

  // Nothing pressed, so do nothing.
  return NULL;
}
```

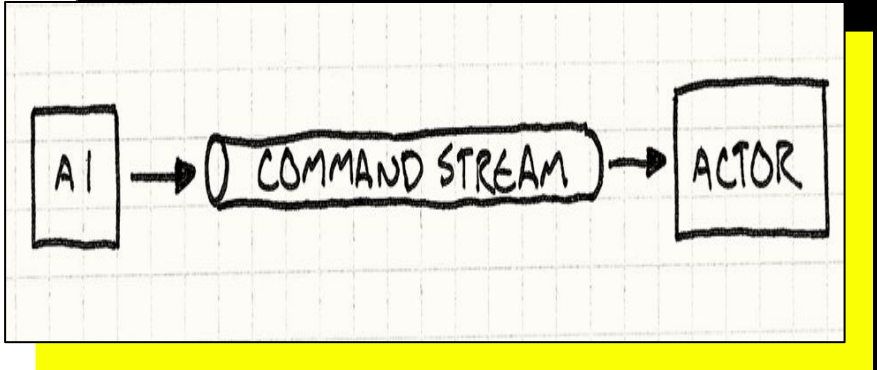
2.

x5

Command Design Pattern

AI Commands

- **This pattern** can also be used as an **interface** between the **AI engine** and the **Actors**
    - with the AI code emitting Command objects
- The **decoupling between** the **AI commands** and **Actor code**, gives a lot of **flexibility**
    - For instance, we can use **different AI modules** (e.g. difficulties) for **different actors**



Final Example: Undo and Redo

The most well-known use of this pattern

Command Pattern: Undo and Redo

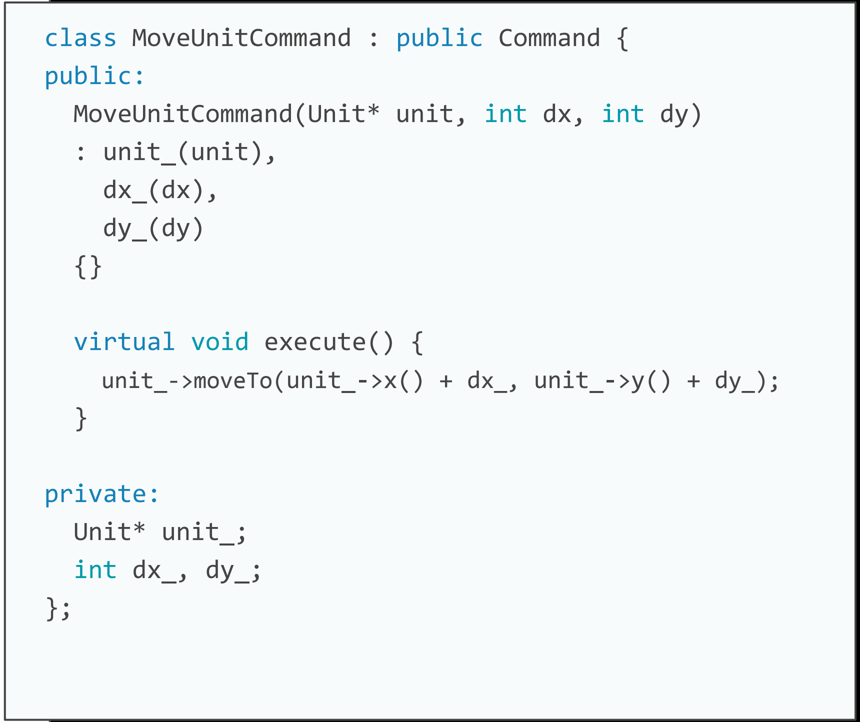
Example: Undo/Redo

- The most well-known use of this pattern.
  - If a **command** object **can do** things, it **can** also **undo**.
  - **Without** the **Command pattern** would be really **hard** to **implement** this feature
- Can be used in strategy games, **turn-based**, etc...

Command Pattern: Undo and Redo

Creating the Command

- This is a little **different** from the **previous example**
  - In the last example, we **wanted** to **abstract** the **Command** from the **actor** that it modified
  - **Now** we specifically **want** to **bind** it to the **unit** being **moved**
  - **This is a specific concrete move**

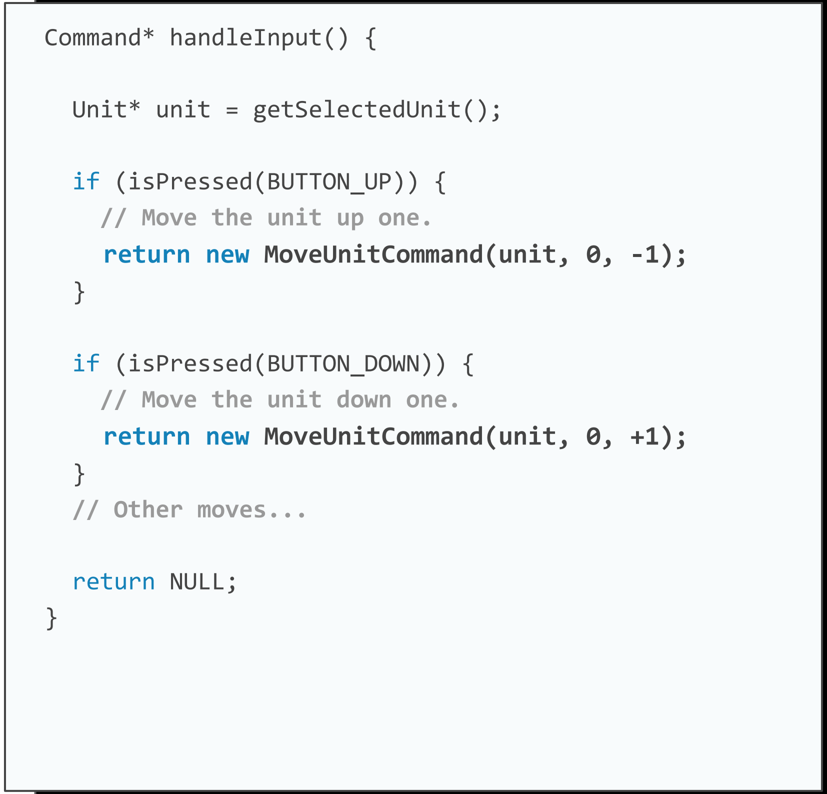


```cpp
class MoveUnitCommand : public Command {
public:
  MoveUnitCommand(Unit* unit, int dx, int dy)
  : unit_(unit),
    dx_(dx),
    dy_(dy)
  {}

  virtual void execute() {
    unit_->moveTo(unit_->x() + dx_, unit_->y() + dy_);
  }

private:
  Unit* unit_;
  int dx_, dy_;
};
```

Command Pattern: Undo and Redo

Input Handling

- In the **previous example** (i.e. User Input) we wanted an **object** to represent **“something”** that **could be done**
- **Now** we want **“something”** that can be **done** in a **specific point in time**
  - This **means** that the **input handling** code will be **creating** an **instance** of the ***MoveUnitCommand*** **everytime** the **player** chooses this **action**
  - This fact will come in handy to the Undo



```
Command* handleInput() {

  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
    // Move the unit up one.
    return new MoveUnitCommand(unit, 0, -1);
  }

  if (isPressed(BUTTON_DOWN)) {
    // Move the unit down one.
    return new MoveUnitCommand(unit, 0, +1);
  }
  // Other moves...

  return NULL;
}
```

Command Pattern: Undo and Redo

Undoable Command

1. To add the Undo feature, we define another rule to our Command
2. Finally, our **previous *MoveUnitCommand*** with the ***undo()*** method

1.

```
class Command {
public:
  virtual ~Command() {}
  virtual void execute() = 0;
  virtual void undo() = 0;
};
```

Note: in some cases, when the inverse operation is not trivial, it can be more efficient to store the previous state in the command object and then restore it
(e.g. when applying a matrix transformation)

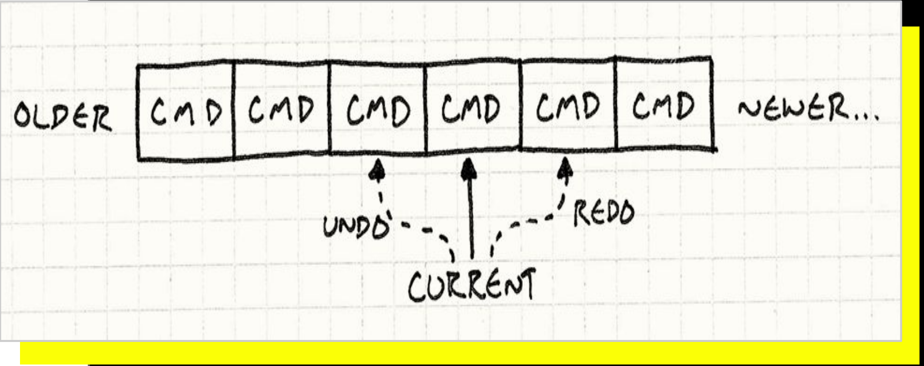
2.

```
class MoveUnitCommand : public Command {
public:
  MoveUnitCommand(Unit* unit, int dx, int dy)
  : unit_(unit),
    dx_(dx),
    dy_(dy)
  {}

  virtual void execute() {
    unit_->moveTo(unit_->x() + dx_, unit_->y() + dy_);
  }

  virtual void undo() {
    unit_->moveTo(unit_->x() - dx_, unit_->y() - dy_);
  }

private:
  Unit* unit_;
  int dx_, dy_;
};
```

Command Pattern: Undo and Redo

Multiple Levels of Undo

- Instead of remembering the last command, we **keep** a list of **commands** and a **reference** to the “**current**” one
- When the player chooses “**Undo**”, we undo the **current command** and move the **current pointer back**
- When they choose “**Redo**”, we **advance** the **pointer** and **then execute** that **command**
- If the player **chooses** a **new command after undoing** some, **everything after** the **current command** is **discarded**



Game Programming Patterns

The Command