Flexibility ⟷ Speed

make fun game fast > make fast game fun

Command — A command is a reified method call

a method call wrapped in an object

usually called once per frame

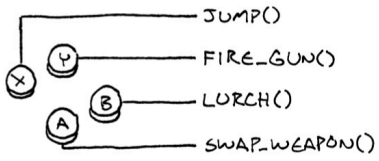

Figure 2.1 – Buttons mapped to game actions

A dead 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();
}
```

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();
  else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

Where each input used to directly call a function, now there's a layer of indirection.

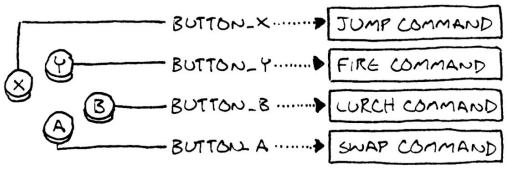

Figure 2.2 – Buttons mapped to assignable commands

This is the Command pattern in a nutshell. If you can see the merit of it already, consider the rest of this chapter a bonus.

We define a base class that represents a triggerable game command:

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

Then we create subclasses for each of the different game actions:

```
class JumpCommand : public Command
{
public:
  virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
  virtual void execute() { fireGun(); }
};

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

In our input handler, we store a pointer to a command for each button:

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

  // Methods to bind commands...

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

Now the input handling just delegates to those: