Basics of Design Patterns

Design patterns play a crucial role in software engineering by offering a standard approach to solving common problems, improving code **readability, reusability, and maintainability**. Although astronomy typically involves straightforward data analysis with limited lines of code, learning established methods to organize code can be advantageous. In this note, I will introduce three design patterns, which are a simplified version of the first chapter of the book Game Programming Patterns, to provide a basic understanding of their benefits. Interested individuals can seek additional resources such as the original book to delve deeper into the topic. (The code is written in C++, and a basic understanding of object-oriented programming may be necessary.)

Command

In the book, the author defines a command as **"a reified method call."** The command pattern's general concept is to **transform an action into a piece of data** - an object - that can be stored in a variable, passed to a function, and so on. This decouples the input handler from the actual execution of the action.

Using an example from the book, in every game, there is a code segment that reads and interprets raw user input, such as button presses, keyboard events, and mouse clicks. It interprets each input and transforms it into a meaningful game action. To implement this, the most straightforward approach is to

```
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();
}
```

In this implementation, the **input system is tightly coupled with the output actions**, which presents several drawbacks. We need to modify the code every time we want to change the mapping of input signals to output actions, and errors in the action function might hamper the input handler from running, among other issues. To decouple the input handler from the output actions, we can utilize the command pattern.

The first step is to define a class for each action:

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

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

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

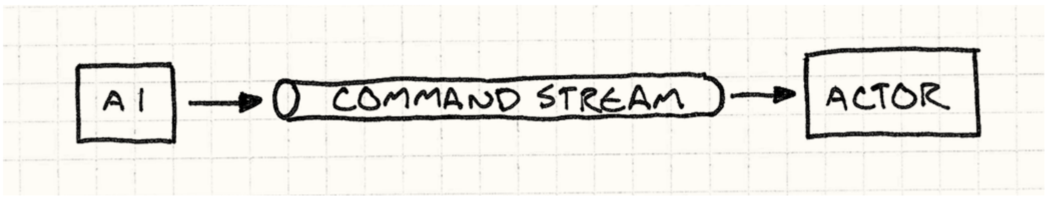
Then the input handler can be implemented as

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

  // Methods to bind commands...

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

Command* InputHandler::handleInput()
{
  //return a command oject according to the button pressed
  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;
}
```

After taking some inputs, we can execute them using

```
Command* command = inputHandler.handleInput();
if (command)
{
  command->execute();
  //we can also pass some arguments to the action here if necessary
  // for example, command->execute(speed=1);
}
```

You can see that we decouple the codes into separate parts:



In the command pattern, we e**ssentially encapsulate each input request as a command object**. A significant advantage of this approach is that it enables us to implement an "**undo**" operation easily.

To illustrate this, we can examine the implementation of a particular command:

```
class MoveUnitCommand : public Command
{
public:
  MoveUnitCommand(Unit* unit, int x, int y)
  : unit_(unit),
    xBefore_(0),
    yBefore_(0),
    x_(x),
    y_(y)
  {}

  virtual void execute()
  {
    // Remember the unit's position before the move
    // so we can restore it.
    xBefore_ = unit_->x();
    yBefore_ = unit_->y();

    unit_->moveTo(x_, y_);
  }

  virtual void undo()
  {
    unit_->moveTo(xBefore_, yBefore_);
  }

private:
  Unit* unit_;
  int xBefore_, yBefore_;
  int x_, y_;
};
```

Note that the key difference here is that we save the previous state in the command object. Therefore after executing it, we can restore previous state:

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

That's all regarding Command Pattern. To summarize, here is a list of advantages that this pattern provides according to ChatGPT:

1. Decoupling: This pattern decouples the object that triggers, or "invokes," the action from the actual object that performs the action, called the "receiver." This reduces the dependencies between objects, making code more maintainable and easier to modify.

2. Flexibility: Commands can be easily added, removed, or replaced without modifying the code that uses them. This makes the system more flexible and easier to extend.
3. Logging and Undo/Redo: Commands provide a natural way to log actions and to perform undo and redo operations, by storing the information needed to reverse the actions.
4. Abstraction: The Command Design Pattern allows higher-level abstractions to be built on top of individual commands, making complex actions easier to perform and manage.
5. Security: Commands can be used to implement secure systems by verifying user permissions before executing certain actions. This improves security by preventing unauthorized access to sensitive operations.

Observer

The Observer design pattern is a software design pattern that establishes a one-to-many dependence between objects. This allows for **automatic notification and update of all the observers when the state of the subject changes**. The Observer pattern promotes decoupling of objects, leading to loose coupling and easy maintenance and scalability.

As an example, let's consider adding an achievements system to our game. The system will have various badges that players can earn for completing specific milestones, such as "Kill 100 Monkey Demons," "Fall off a Bridge," or "Complete a Level Wielding Only a Dead Weasel."

To implement the "Fall off a Bridge" badge, we may be tempted to add achievement code directly into the game code.

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    unlock_achievement()
  }
}
```

The implementation mentioned earlier has various limitations. It tightly couples the achievement system with the physics system, which can cause design issues. Moreover, if multiple systems rely on the falling event such as animation, score tracking, etc., it will introduce complexities and additional code changes.

A better approach to handle such scenarios is to implement it using the Observer pattern:

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    notify(entity, EVENT_START_FALL);
  }
}
```

All it does is say, "Uh, I don't know if anyone cares, but this thing just fell. Do with that as you will." This is just to give a basic picture of the observer pattern to you. In the following codes, we will walk through how to implement it. We'll start with the nosy class that wants to know when another object does something interesting. We use implement the general observer class as :

```
class Observer
{
public:
  virtual ~Observer() {}
  virtual void onNotify(const Entity& entity, Event event) = 0;
};
```

Any concrete class that implements this becomes an observer. In our example, that's the achievement system, so we'd have something like so:

```
class Achievements : public Observer
{
public:
  virtual void onNotify(const Entity& entity, Event event)
  {
    switch (event)
    {
    case EVENT_ENTITY_FELL:
      if (entity.isHero() && heroIsOnBridge_)
      {
        unlock(ACHIEVEMENT_FELL_OFF_BRIDGE);
      }
      break;

      // Handle other events, and update heroIsOnBridge_...
    }
  }

private:
  void unlock(Achievement achievement)
  {
    // Unlock if not already unlocked...
  }

  bool heroIsOnBridge_;
};
```

The notification method is invoked by the object being observed. That object is called the “subject”. It has two jobs. First, it holds the list of observers that are waiting oh-so-patiently for a missive from it:

```
class Subject
{
private:
  Observer* observers_[MAX_OBSERVERS];
  int numObservers_;
};
```

The important bit is that the subject exposes a *public* API for modifying that list:

```
class Subject
{
public:
  void addObserver(Observer* observer)
  {
    // Add to array...
  }

  void removeObserver(Observer* observer)
  {
    // Remove from array...
  }

  // Other stuff...
};
```

That allows outside code to control who receives notifications. The subject communicates with the observers, but it isn’t *coupled* to them. In this example, no line of physics code will mention achievements. Yet, it can still talk to the achievements system. That’s the clever part about this pattern.

The other job of the subject is sending notifications:

```
class Subject
{
protected:
  void notify(const Entity& entity, Event event)
  {
    for (int i = 0; i < numObservers_; i++)
    {
      observers_[i]->onNotify(entity, event);
    }
  }

  // Other stuff...
};
```

To let one class to send out notifications, we can just let it inherit `Subject`:

```
class Physics : public Subject
{
public:
  void updateEntity(Entity& entity);
};
```

Then when the physics engine does something noteworthy, it calls `notify()` like in the motivating example before. That walks the observer list and gives them all the heads up.

Same as before, I will end this part with a summary of advantages of the observer pattern given chatGPT:

1. Decoupling of objects: The observer pattern promotes loose coupling between objects, allowing for changes to be made to one object without affecting the others. This results in more modular and maintainable code.
2. Scalability: The observer pattern makes it easy to add or remove observers without affecting the subject object or other observers. This makes the pattern scalable and adaptable to changing requirements.
3. Reusability: Because of the decoupling of objects, the observer pattern promotes code reusability. Observers can be reused in different contexts, and subjects can be reused with different sets of observers.
4. Customizability: The observer pattern allows for customization of behavior based on the needs of the observer. Each observer can choose what actions to take when notified of changes in the subject object.
5. Flexibility: The observer pattern allows for flexible communication between objects, which can facilitate complex interactions between components of a system.

State

State design pattern is a behavioral design pattern that **allows an object to change its behavior based on its internal state.** In essence, the pattern enables an object to alter its behavior(input handler) when its internal state changes.

Say we are implementing the heroine and making her respond to user input. Push the **B** button and she should jump. Simple enough, we can do:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

However, there’s nothing to prevent “air jumping”—keep hammering B while she’s in the air, and she will float forever. The simple fix is to add an `isJumping_` Boolean field to `Heroine` that tracks when she’s jumping, and then do:

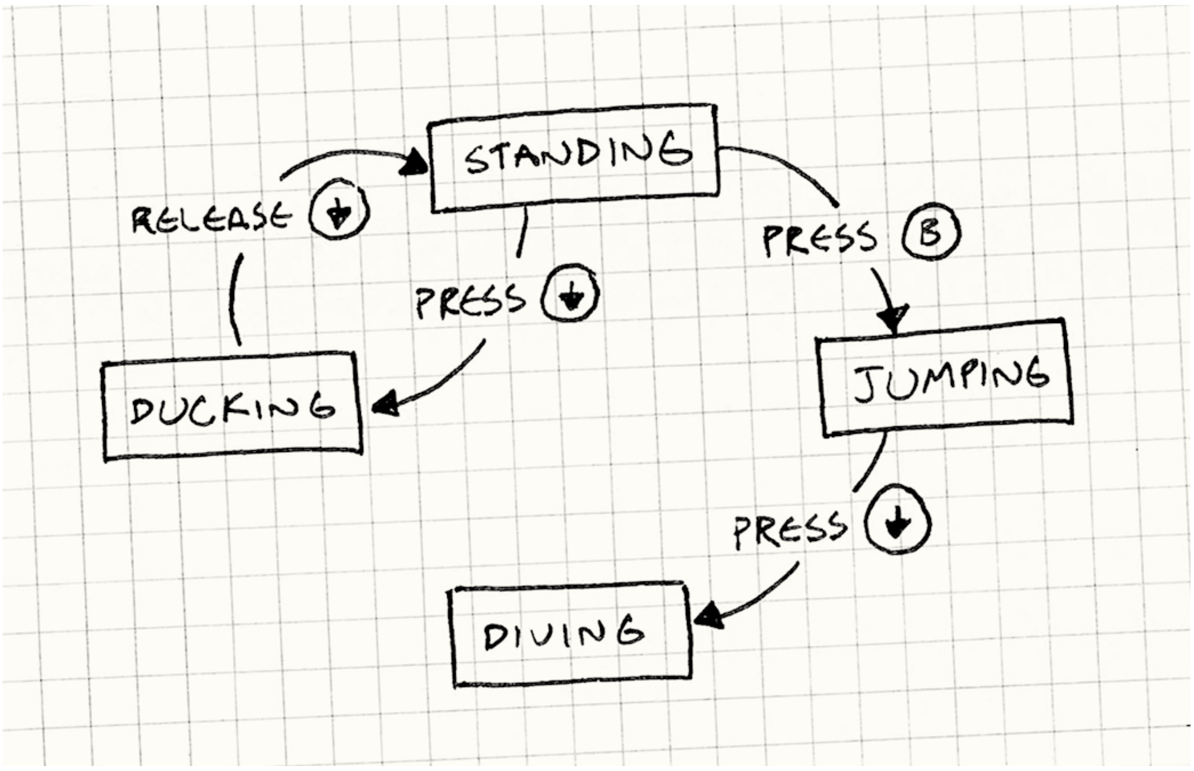
```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_)
    {
      isJumping_ = true;
      // Jump...
    }
  }
}
```

The implementation can be much more complex if we add one more action for the Heroine like ducking

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

As you can see from the code, things are already starting to get a bit messy. And if we were to add more actions for the heroine (such as *walking*, for example), things could quickly become even more complicated and difficult to manage.

To simplify the code and make it more manageable, we can **start by leveraging a Finite State Machine.** FSMs can help us to more clearly define and organize the possible states and transitions of our heroine object. This will enable us to create more elegant and maintainable code, while also improving the overall performance and user experience of our game.



The next step is to determine how we can implement this FSM. One powerful way to achieve this is by using the State Pattern. This involves creating an interface that defines the various states that our game heroine can be in, along with virtual methods for each behavior that varies based on her current state.

```
class HeroineState
{
public:
  virtual ~HeroineState() {}
  virtual void handleInput(Heroine& heroine, Input input) {}
  virtual void update(Heroine& heroine) {}
};
```

Then each state in the FSM should mplement the interface. For example:

```
class DuckingState : public HeroineState
{
public:
  DuckingState()
  : chargeTime_(0)
  {}

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN)
    {
      // Change to standing state...
      heroine.setGraphics(IMAGE_STAND);
    }
  }

  virtual void update(Heroine& heroine) {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      heroine.superBomb();
    }
  }

private:
  int chargeTime_;
};
```

Then we can implement the Heroine class as:

```
class Heroine
{
public:
  virtual void handleInput(Input input)
  {
    state_->handleInput(*this, input);
  }

  virtual void update()
  {
    state_->update(*this);
  }

  // Other methods...
private:
  HeroineState* state_;
};
```

Then, we need to figure out how to store all the possible states of our Heroine. The easist way of doing this is to make them static instances(Even if you have a bunch of FSMs all going at the same time in that same state, they can all point to the same instance since it has nothing machine-specific about it):

```
class HeroineState
{
public:
  static StandingState standing;
  static DuckingState ducking;
  static JumpingState jumping;
  static DivingState diving;

  // Other code...
};
```

To change state, you can set it inside the state class. For example, to make the heroine jump, the standing state would do something like:

```
if (input == PRESS_B)
{
  heroine.state_ = &HeroineState::jumping;
  heroine.setGraphics(IMAGE_JUMP);
}
```

However, the state has field specific to the Heroine instance(for example the ducking state). In that case, we have to create a state object when we transition to it. We'll allow `handleInput()` in `HeroineState` to optionally return a new state. The transition of state can be implemented as:

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL)
  {
    delete state_;
    state_ = state;
  }
}
```

Now, the standing state can transition to ducking by creating a new instance:

```
HeroineState* StandingState::handleInput(Heroine& heroine,
                                          Input input)
{
  if (input == PRESS_DOWN)
  {
    // Other code...
    return new DuckingState();
  }

  // Stay in this state.
  return NULL;
}
```

Finally, **some times we want to do something when the heroine transit from one state to another**(for example change the graphics of the heroine). To do so, we can implement `enter` method for each state and call it when transiting:

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL)
  {
    delete state_;
    state_ = state;

    // Call the enter action on the new state.
    state_->enter(*this);
  }
}
```

We can, of course, also extend this to support an *exit action*. This is just a method we call on the state we're *leaving* right before we switch to the new state.

Here is a summary of advantages of this pattern:

1. Flexibility: The pattern offers greater flexibility compared to other design patterns, making it easier to extend and modify the behavior of an object without affecting its context. This is

particularly useful when dealing with complex behaviors.

2. Encapsulation: The State design pattern promotes encapsulation by defining the behavior associated with a specific state within a separate class.
3. Improved maintainability: The State design pattern makes it easier to maintain and update the behavior of an object. Any changes made to one state only affect that particular state, allowing for updates to be made to the object's behavior without affecting other states.
4. Simplified code: The pattern simplifies code by removing large and complex conditional statements, replacing them with a set of state classes.
5. Improved reusability: The State design pattern promotes code reusability by allowing the state classes to be reused in different contexts.
6. Improved testing: The State design pattern promotes easier testing by allowing the behavior of an object to be tested in isolation, without affecting other parts of the program.