Crystal patterns

 $\label{lem:decompletely} \textbf{Design patterns completely implemented in Crystal language.}$

The goal is to have a quick set of examples of GOF patterns for Crystal users.

Behavioural patterns

Behavioral patterns define manners of communication between classes and objects.

Command

The command pattern is a design pattern that enables all of the information for a request to be contained within a single object. The command can then be invoked as required, often as part of a batch of queued commands with rollback capabilities.

```
abstract class Command
  abstract def execute
  abstract def undo
class MoveLeft < Command</pre>
  def execute
   puts "One step left"
  def undo
   puts "Undo step left"
  end
end
class MoveRight < Command</pre>
  def execute
    puts "One step right"
  def undo
    puts "Undo step right"
  end
end
class Hit < Command
  def execute</pre>
    puts "Do one hit"
  end
  def undo
    puts "Undo one hit"
  end
end
class CommandSequence < Command
  def initialize</pre>
    @commands = [] of Command
  end
  def <<(command)</pre>
    @commands << command
  end
  def execute
    @commands.each &.execute
  end
  def undo
    @commands.reverse.each &.undo
  end
end
class CommandSequencePlayer
  def initialize(@sequence : CommandSequence)
  def forward
    @sequence.execute
  end
  def backward
    @sequence.undo
  end
end
sequence = CommandSequence.new.tap do |r|
  r << MoveLeft.new
  r << MoveLeft.new
  r << MoveLeft.new
  r << Hit.new
    << MoveRight.new
player = CommandSequencePlayer.new sequence
player.forward
  One step left
# One step left
# One step left
# Do one hit
# One step right
player.backward
# Undo step right
```

Undo one hit
Undo step left
Undo step left
Undo step left

Iterator

The iterator pattern is a design pattern that provides a means for the elements of an aggregate object to be accessed sequentially without knowledge of its structure. This allows traversing of lists, trees and other structures in a standard manner.

```
class Fighter
   getter name, weight
   def initialize(@name, @weight)
  end
end
class Tournament
   include Enumerable(Fighter)
   def initialize
      @fighters = [] of Fighter
   end
   def << (fighter)
@fighters << fighter</pre>
   end
   def each
     @fighters.each { |fighter| yield fighter }
   end
end
 # Sample
tournament = Tournament.new.tap do |t|
  t << Fighter.new "Jax", 150
t << Fighter.new "Liu Kang", 84
t << Fighter.new "Scorpion", 95
t << Fighter.new "Sub-Zero", 95
t << Fighter.new "Smoke", 252
end
tournament.select { |fighter| fighter.weight > 100 }
.map {|fighter| fighter.name}
# => ["Jax", "Smoke"]
```

Observer

Defines a link between objects so that when one object's state changes, all dependent objects are update automatically. Allows communication between objects in a loosely coupled manner.

```
module Observable(T)
  getter observers
  def add observer(observer)
    @observers ||= [] of T
@observers.not_nil! << observer</pre>
  end
  def delete_observer(observer)
    @observers.try &.delete(observer)
  end
 def notify_observers
@observers.try &.each &.update self
  end
end
class Fighter
  include Observable (Observer)
  getter name, health
  def initialize(@name)
    @health = 100
  end
  def damage(rate)
    if @health > rate
  @health -= rate
    else
      @health = 0
    end
    notify_observers
  end
  def is_dead?
    @health <= 0</pre>
  end
end
abstract class Observer
  abstract def update(fighter)
end
class Stats < Observer</pre>
  def update(fighter)
   puts "Updating stats: #{fighter.name}'s health is #{fighter.health}"
  end
end
class DieAction < Observer</pre>
  def update(fighter)
   puts "#{fighter.name} is dead. Fight is over!" if fighter.is_dead?
  end
end
# Sample
fighter = Fighter.new("Scorpion")
fighter.add_observer(Stats.new)
fighter.add_observer(DieAction.new)
fighter.damage(10)
# Updating stats: Scorpion's health is 90
fighter.damage(30)
# Updating stats: Scorpion's health is 60
fighter.damage(75)
# Updating stats: Scorpion's health is 0
# Scorpion is dead. Fight is over!
```

Iterator

Allows a set of similar algorithms to be defined and encaplusated in their own classes. The algorithm to be used for a particular purpose may then be selected at run-time.

```
class Fighter
  getter health, name setter health
  def initialize(@name, @fight strategy)
    @health = 100
  end
  def attack(opponent)
    @fight_strategy.attack self, opponent
puts "#{opponent.name} is dead" if opponent.is_dead?
  def is dead?
   health <= 0
  end
  def damage(rate)
  if @health > rate
       @health -= rate
    else
       @health = 0
    end
  end
end
abstract class FightStrategy
  HITS = {:punch => 40, :kick => 12}
  abstract def attack(fighter, opponent)
end
class Puncher < FightStrategy</pre>
  def attack(ft, op)
  puts "#{ft.name} attacks #{op.name} with 1 punch."
    op.damage(HITS[:punch])
  end
end
class Combo < FightStrategy
  def attack(ft, op)</pre>
    puts "#{ft.name} attacks #{op.name} with 2 kicks and 1 punch."
    op.damage(HITS[:kick])
    op.damage(HITS[:kick]
    op.damage(HITS[:punch])
  end
end
# Sample
scor = Fighter.new("Scorpion", Puncher.new)
noob = Fighter.new("Noob", Combo.new)
noob.attack scor
# Noob attacks Scorpion with 2 kicks and 1 punch.
scor.attack noob
# Scorpion attacks Noob with 1 punch.
noob.attack scor
  Noob attacks Scorpion with 2 kicks and 1 punch.
# Scorpion is dead
```

Creational Patterns

Creational patterns provide ways to instantiate single objects or groups of related objects.

Structural Patterns

Structural patterns provide a manner to define relationships between classes or objects.

Composite

The composite pattern is a design pattern that is used when creating hierarchical object models. The pattern defines a manner in which to design recursive tree structures of objects, where individual objects and groups can be accessed in the same manner

```
abstract class Strike
  abstract def damage
abstract def attack
class Punch < Strike</pre>
  def attack
   puts "Hitting with punch"
  def damage
  end
end
class Kick < Strike</pre>
  def attack
  puts "Hitting with kick"
  def damage
    8
  end
end
class Combo < Strike
  def initialize</pre>
   @sub_strikes = [] of Strike
  def << (strike)</pre>
   @sub_strikes << strike
  end
  def damage
   @sub_strikes.inject(0) { |acc, x| acc + x.damage }
  end
  def attack
    @sub_strikes.each &.attack
  end
end
# Sample
super_strike = Combo.new.tap do |s|
s << Kick.new</pre>
  s << Kick.new
 s << Punch.new
end
super strike.attack
# Hitting with kick
# Hitting with kick
# Hitting with punch
```