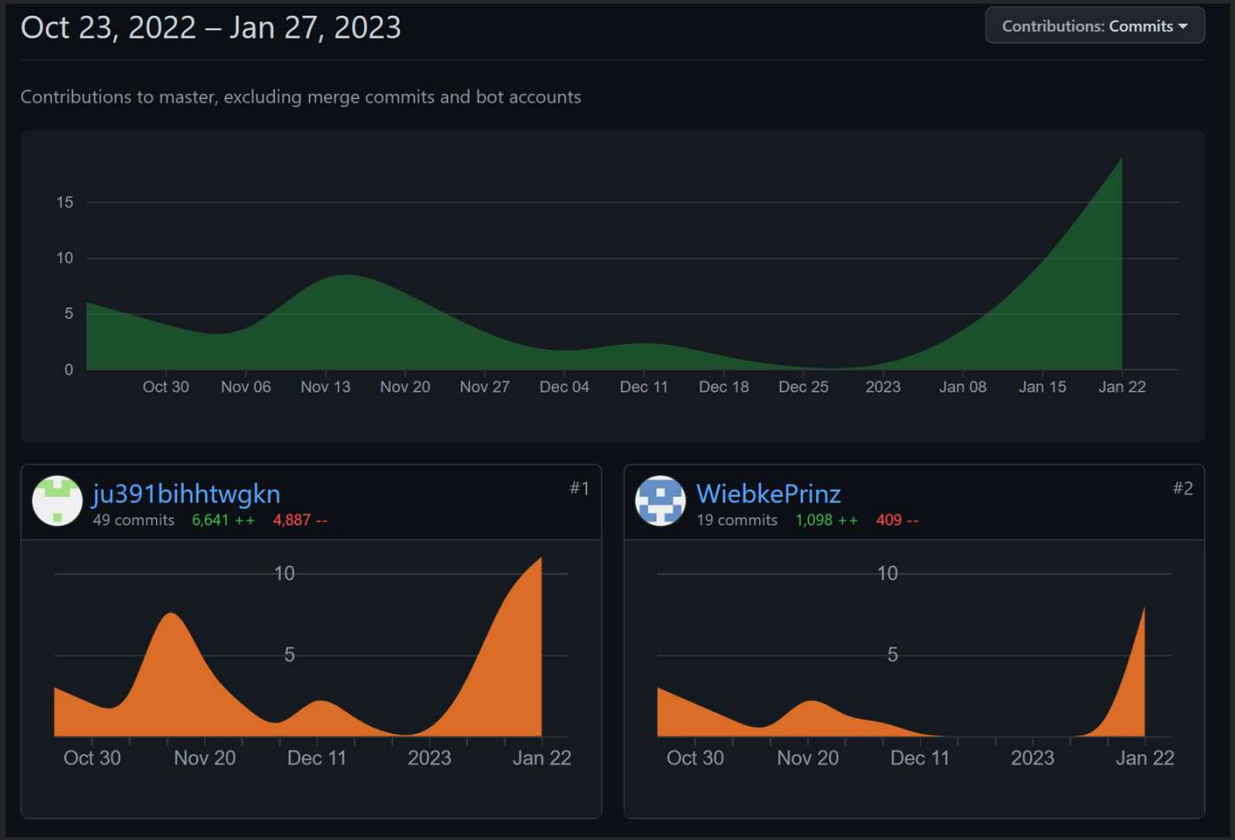
EXPLODING KITTENS

Whats the Game about?

- Whoever explodes loses
- Who doesn't explode, wins
- All other cards reduce the chance to explode by exploding kittens.

- Explode by pulling an exploding kittens
- Survive by luck and defuse cards
- Other Cards:
  - look into the future
  - suspend a turn
  - attack fellow players
  - draw from the bottom
  - steal cards from an opponent
  - reshuffle
  - ...

Git and Github



WP1

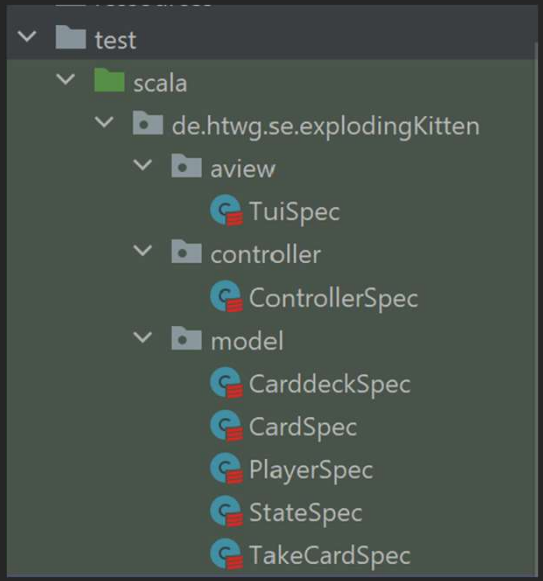
Coverage Reports

Exploding Kitten

This is a project for Software Engineering!

Scala CI failing

coverage 80%



Text User Interface

Take Card

```
def processInputLine(): Unit = {
  while (controller.flag == true) {
    println("Your turn " +
controller.gameState.players(controller.gameState.currentPlayer).name)

println(controller.gameState.players(controller.gameState.currentPlayer).handCards.toString()
)
    val input = readLine()
    input match {
      case "t" => {
        if(controller.gameState.deck.head.cardName == "Exploding Kitten") {
          println("You have drawn a Exploding Kitten")
          println("Choose a place to put it into the deck!")
          context.setStrategy(new TakeExploding(readLine().toInt))
          context.executeStrategy(controller)

          context.setStrategy(new NextPlayer())
          context.executeStrategy(controller)
        } else {
          // Take a Card
          context.setStrategy(new TakeCard())
          context.executeStrategy(controller)
          // Next Player
          context.setStrategy(new NextPlayer())
          context.executeStrategy(controller)
        }
      }
```

```
Wiebke   you have drawn this card: Defuse

Julian you have drawn this card: Exploding Kitten

Julian You have lost :(
```

Play Card

```
case "p" =>
  // Play a Card
  println("Which Card do you want to play ? Please enter the Number of the Card")
  val input = readLine().toInt
  val cardType =
controller.gameState.players(controller.gameState.currentPlayer).handCards(input).actionCode

  cardType match {
    case 1 =>
      context.setStrategy(new DrawFromTheBottom(input))
      context.executeStrategy(controller)
    case 2 =>
      context.setStrategy(new SeeTheFuture(input))
      context.executeStrategy(controller)
    case 3 =>
      context.setStrategy(new Skip(input))
      context.executeStrategy(controller)
    case 4 =>
      context.setStrategy(new Attack(input))
    case 5 =>
      println("Choose a Player to Attack")
      context.setStrategy(new TargetedAttack(input, readLine().toInt))
      context.executeStrategy(controller)
    case 14 =>
      context.setStrategy(new Shuffle(input))
      context.executeStrategy(controller)
  }
```

```
You played: See The Future
Vector(Defuse
Defuse the exploding kitten, Defuse
Defuse the exploding kitten, Feral Cat
Use this as any cat card)
```

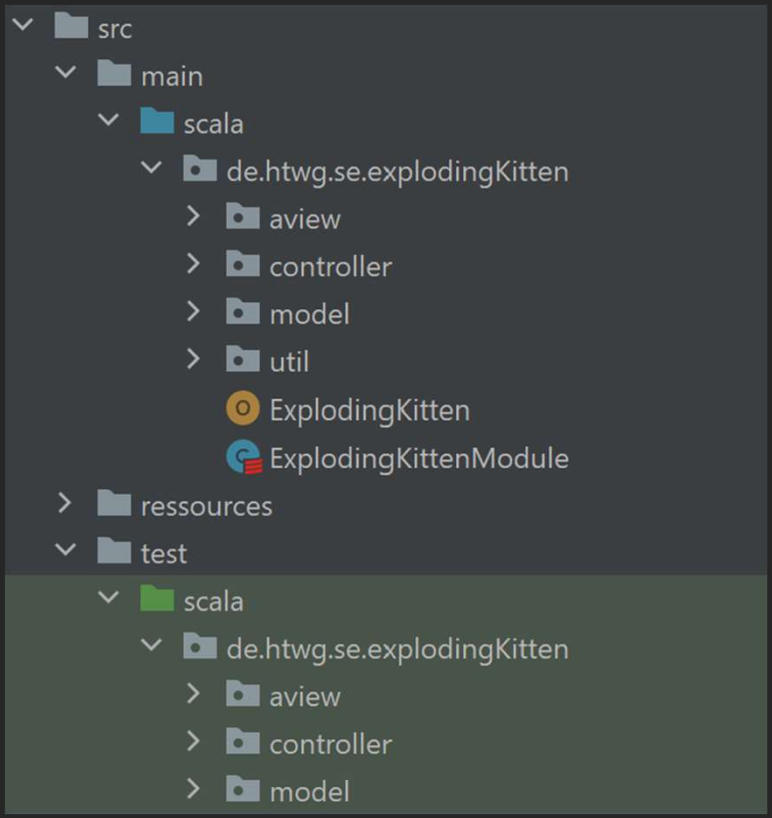
```
You played: Targeted Attack
```

```
You played: Draw From The Bottom
l you have drawn this card:
See The Future
Look at the top 3 cards
```

Undo/Redo and Save/Load

```
case "r" =>
      controller.redo()
case "u" =>
      controller.undo()
case "s" =>
  controller.save()
case "l" =>
  controller.load()
    }
  }
}
override def update: Unit = controller.gameState
```

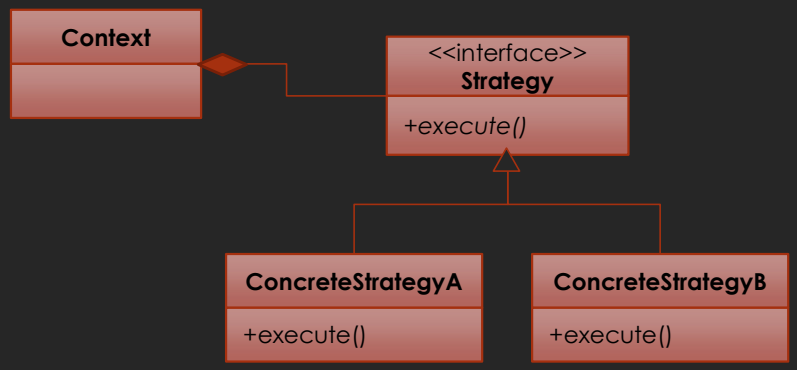
MVC Architecture



Design Pattern

Strategy – Java Style


```
trait Move {
  def makeMove(state: Gamestate): Gamestate
}
```

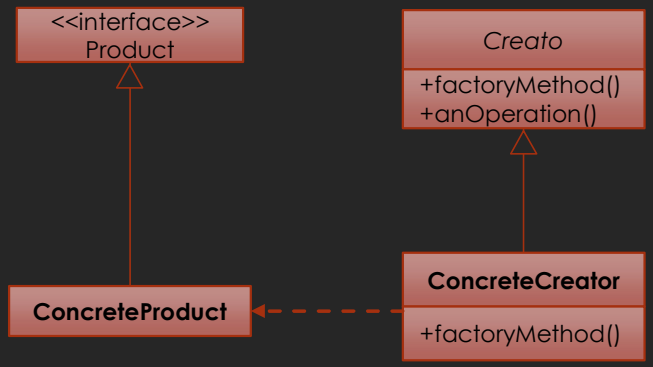



```
class NextPlayer extends Move {
  override def makeMove(state: Gamestate): Gamestate = {
    val nextPlayer = (state.currentPlayer + 1) % state.pl
```

```
class TakeCard extends Move {

  var flag = false

  override def makeMove(state: Gamestate): Gamestate = {
    // get top card
    val tempCard = state.deck.head
```

```
class SeeTheFuture(i: Int) extends Move {

 var flag = false

 override def makeMove(state: Gamestate): Gamestate = {

  val currentPlayer = state.currentPlayer
  val card = state.players(currentPlayer).handCards(i)
  println("You played: " + card.cardName)
  val newDiscardPile = state.discardPile.appended(card)
  val newHandCards = state.players(currentPlayer).playCard(card)
  val newPlayer = state.players.updated(currentPlayer, Player(state.players(currentPlayer).name, newHandCards))
  val newState = state.copy(players = newPlayer, discardPile = newDiscardPile)

  if (newState.deck.length > 2) {
   val topCards = newState.deck.take(3)
   println(topCards)
   newState
  } else {
   val topCards = newState.deck.take(newState.deck.length)
   println(topCards)
   newState
  }
 }
}
```

Factory

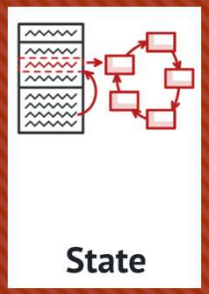



```
trait Card {
  def cardName: String
  def cardDescription: String
  def card: String
  def actionCode: Int
  def toString: String
}
```

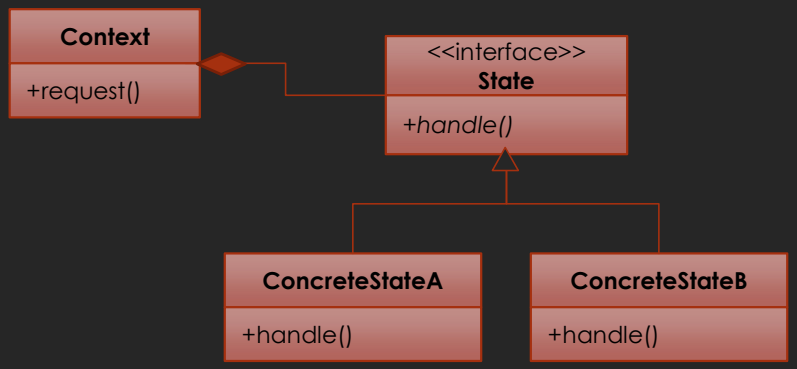
```
def apply(s: String): Card = {
  s match {
    case "DrawFromTheBottom" => new DrawFromTheBottom
    case "SeeTheFuture" => new SeeTheFuture
    case "Skip" => new Skip
    case "Attack" => new Attack
    case "TargetedAttack" => new TargetedAttack
    case "Defuse" => new Defuse
    case "ExplodingKitten" => new ExplodingKitten
    case "AlterTheFuture" => new AlterTheFuture
    case "FeralCat" => new FeralCat
    case "MelonCat" => new MelonCat
    case "BeardedCat" => new BeardedCat
    case "TacoCat" => new TacoCat
    case "HairyPotatoCat" => new HairyPotatoCat
    case "RainbowCat" => new RainbowCat
    case "Shuffle" => new Shuffle
  }
```

```
object Card {

  val eol = sys.props("line.separator")
  private class DrawFromTheBottom extends Card {
    override def cardName: String = "Draw from the Bottom"
    override def cardDescription: String = "Draw a card from the Bottom"
    override def card:String = cardName + eol + cardDescription
    override def actionCode: Int = 1
    override def toString: String = card
  }

  private class SeeTheFuture extends Card {
    override def cardName: String = "See the Future"
```

State

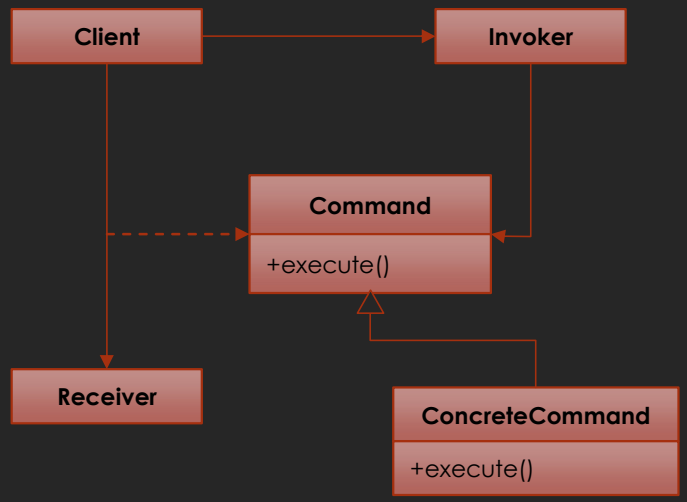



```scala
trait GameStateInterface {

  val currentPlayer: Int
  val players: Vector[Player]
  val deck: Vector[Card]
  val discardPile: Vector[Card]

  def handle(move: Move): GameStateInterface
}
```




```scala
case class Gamestate(
                      currentPlayer: Int,
                      players: Vector[Player],
                      deck: Vector[Card],
                      discardPile: Vector[Card],
                    ) extends GameStateInterface {

  def handle(move: Move): Gamestate = {
    move match {
      case nextPlayer => nextPlayer.makeMove( state = this)
      case takeCard => takeCard.makeMove( state = this)
      case playCard => playCard.makeMove( state = this)
    }
  }
}
```

Undo



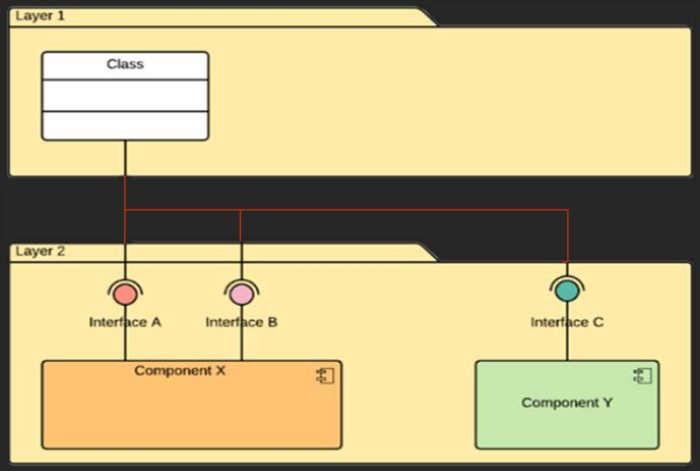

```
class UndoManager {
  private var undoStack: List[Command]= Nil
  private var redoStack: List[Command]= Nil
  def doStep(command: Command): Unit = {
    undoStack = command::undoStack
    command.doStep()
  }
```

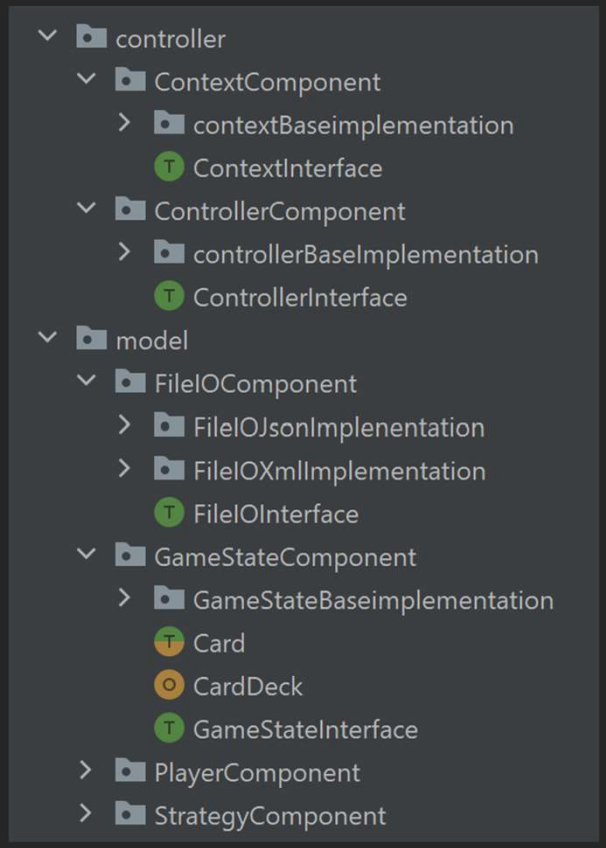
```
def undoStep: Unit= {
  undoStack match {
    case  Nil =>
    case head::stack => {
      head.undoStep
      undoStack=stack
      redoStack= head::redoStack
    }
  }
}
```

```
def redoStep: Unit = {
  redoStack match {
    case Nil =>
    case head::stack => {
      head.redoStep()
      redoStack=stack
      undoStack=head::undoStack
    }
  }
}
```

GUI

Components


Layer 1
Class
Layer 2
Interface A
Interface B
Interface C
Component X
Component Y


controller
ContextComponent
contextBaseimplementation
ContextInterface
ControllerComponent
controllerBaseImplementation
ControllerInterface
model
FileIOComponent
FileIOJsonImplenentation
FileIOXmlImplementation
FileIOInterface
GameStateComponent
GameStateBaseimplementation
Card
CardDeck
GameStateInterface
PlayerComponent
StrategyComponent

File I/O JSON

```
trait FileIOInterface {
  def save(gameState: GameStateInterface): Unit
  def load: GameStateInterface
}
```

Load()

```
  val currentPlayer = (json \ "gameState" \ "currentPlayer").get.as[String].toInt
  val p1Name = (json \ "gameState" \ "player 1" \ "name").get.as[String]
  val p1Cards = Cards(json, player = "player 1", length = "cardLength")
  val p2Name = (json \ "gameState" \ "player 2" \ "name").get.as[String]
  val p2Cards = Cards(json, player = "player 2", length = "cardLength")
  val deck = Cards(json, player = "deck", length = "cardLength")
  val discardPile = Cards(json, player = "discardPile", length = "cardLength")

  Gamestate(currentPlayer,Vector(Player(p1Name, p1Cards),Player(p2Name, p2Cards)), deck, discardPile)
}
```

Save()

```
"player 3" -> Json.obj(
  fields = "name" -> gameState.players(2).name,
  "cards" -> gameState.players(2).handCards.map(k => k.cardName),
  "cardLength" -> gameState.players(2).handCards.length.toString
),
"deck" -> Json.obj(
  fields = "cards" -> gameState.deck.map(k => k.cardName),
  "cardLength" -> gameState.deck.length.toString
),
"discardPile" -> Json.obj(
  fields = "cards" -> gameState.discardPile.map(k => k.cardName),
  "cardLength" -> gameState.discardPile.length.toString
)
```

File I/O XML

```
def playerToXml(player: Player) = {
 <player>
  <name>{player.name}
  </name>
  <cards>{player.handCards.map(k => k.cardName + ",")}
  </cards>
  <cardLength>{player.handCards.length.toString}
  </cardLength>
 </player>
}

def cardsToXml(cards: Vector[Card]) = {
 <cards>{cards.map(k => k.cardName + ",")}
 </cards>
}
```

```
def gameStateToXml(gameState: GameStateInterface): Elem = {
  if(gameState.players.length > 2) {
    <gameState>
      <currentPlayer>
        {gameState.currentPlayer.toString}
      </currentPlayer>
      <playerLength>
        {gameState.players.length.toString}
      </playerLength>
      <player1>
        {playerToXml(gameState.players(0))}
      </player1>
      <player2>
        {playerToXml(gameState.players(1))}
      </player2>
      <player3>
        {playerToXml(gameState.players(1))}
      </player3>
      <deck>
        {cardsToXml(gameState.deck)}
      </deck>
      <discardPile>
        {cardsToXml(gameState.discardPile)}
      </discardPile>
    </gameState>
```

Docker

Dockerfile

```
FROM hseeberger/scala-sbt:8u222_1.3.5_2.13.1
WORKDIR /exploding-k
ADD . /exploding-k
CMD sbt run
```

```
>docker build . -t explodingkitten
```

```
\Exploding-K>docker run --interactive -t explodingkitten
```

Thank you for your attention!