BRODY DEIBY VELARDE HURTADO

```
import turtle
import time
import random
from abc import ABC, abstractmethod

# Configuración de la pantalla
screen = turtle.Screen()
screen.title("Snake Game - Abstract Factory Pattern")
screen.bgcolor("black")
screen.setup(width=600, height=600)
screen.tracer(0)  # Apaga las animaciones automáticas

# Interfaz abstracta para la comida
class Comida(ABC):
    @abstractmethod
    def obtener_color(self):
        pass

    @abstractmethod
    def aplicar_efecto(self, snake, score):
        pass

    @abstractmethod
    def obtener_descripcion(self):
        pass

# Implementaciones concretas de comida
class ComidaVenenosa(Comida):
    def obtener_color(self):
        return "purple"

    def aplicar_efecto(self, snake, score):
        # Reduce el tamaño de la serpiente y el puntaje
        if len(snake.segments) > 1:
            segment = snake.segments.pop()
            segment.goto(1000, 1000)  # Mueve el segmento fuera de la
pantalla
        return max(0, score - 1)  # Reduce el puntaje pero no menos de 0
```

```
    def obtener_descripcion(self):
        return "☠️ Comida Venenosa: -1 punto, -1 segmento"

class ComidaFit(Comida):
    def obtener_color(self):
        return "green"

    def aplicar_efecto(self, snake, score):
        # Aumenta el puntaje normalmente
        snake.add_segment()
        return score + 1

    def obtener_descripcion(self):
        return "✅ Comida Fit: +1 punto"

class ComidaAltoEnGrasas(Comida):
    def obtener_color(self):
        return "yellow"

    def aplicar_efecto(self, snake, score):
        # Aumenta el puntaje significativamente pero ralentiza la serpiente
        snake.add_segment()
        snake.add_segment()
        snake.add_segment()
        snake.speed = max(0.05, snake.speed * 1.5)  # Ralentiza la serpiente
        snake.speed_timer = time.time() + 5  # Efecto dura 5 segundos
        return score + 3

    def obtener_descripcion(self):
        return "🍔 Comida Alto en Grasas: +3 puntos, velocidad reducida por 5
segundos"

class ComidaParaReyes(Comida):
    def obtener_color(self):
        return "orange"

    def aplicar_efecto(self, snake, score):
        # Otorga un bonus máximo y aumenta la velocidad
        for _ in range(5):
            snake.add_segment()
        snake.speed = max(0.01, snake.speed * 0.7)  # Aumenta la velocidad
        return score + 5
```

```
    def obtener_descripcion(self):
        return "🎂 Comida para Reyes: +5 puntos, velocidad aumentada"

# Interfaz de la fábrica abstracta
class FabricaComida(ABC):
    @abstractmethod
    def crear_comida(self):
        pass

# Implementaciones concretas de fábricas
class FabricaComidaVenenosa(FabricaComida):
    def crear_comida(self):
        return ComidaVenenosa()

class FabricaComidaFit(FabricaComida):
    def crear_comida(self):
        return ComidaFit()

class FabricaComidaAltoEnGrasas(FabricaComida):
    def crear_comida(self):
        return ComidaAltoEnGrasas()

class FabricaComidaParaReyes(FabricaComida):
    def crear_comida(self):
        return ComidaParaReyes()

# Clase para la serpiente
class Snake:
    def __init__(self):
        self.segments = []
        self.create_snake()
        self.head = self.segments[0]
        self.head.shape("square")  # Cabeza cuadrada
        self.head.color("green")  # Color verde
        self.speed = 0.1  # Velocidad inicial
        self.speed_timer = 0  # Temporizador para efectos temporales
        self.direction = "right"

        # Personalización de la cabeza
        self.head.color("lime green")  # Verde brillante para la cabeza
        self.head.shapesize(1.2, 1.2)  # Hacer la cabeza un poco más grande
```

```
        self.speed = 0.1
        self.speed_timer = 0
        self.direction = "right"
        self._rotate_head()

    def create_snake(self):
        for i in range(3):
            self.add_segment((-20 * i, 0))

    def add_segment(self, position=None):
        new_segment = turtle.Turtle()
        new_segment.speed(0)
        new_segment.shape("circle")  # Forma circular para el cuerpo

        # Si es el primer segmento (cabeza)
        if len(self.segments) == 0:
            new_segment.color("lime green")
        else:
            # Patrón de colores alternados para el cuerpo
            if len(self.segments) % 2 == 0:
                new_segment.color("dark green")
            else:
                new_segment.color("forest green")

        new_segment.penup()

        if position:
            new_segment.goto(position)
        elif len(self.segments) > 0:
            last_segment = self.segments[-1]
            new_segment.goto(last_segment.xcor(), last_segment.ycor())

        self.segments.append(new_segment)

    def _rotate_head(self):
        # Rota la cabeza según la dirección
        if self.direction == "up":
            self.head.setheading(90)
        elif self.direction == "down":
            self.head.setheading(270)
        elif self.direction == "left":
```

```
            self.head.setheading(180)
        elif self.direction == "right":
            self.head.setheading(0)

    def go_up(self):
        if self.direction != "down":
            self.direction = "up"
            self._rotate_head()

    def go_down(self):
        if self.direction != "up":
            self.direction = "down"
            self._rotate_head()

    def go_left(self):
        if self.direction != "right":
            self.direction = "left"
            self._rotate_head()

    def go_right(self):
        if self.direction != "left":
            self.direction = "right"
            self._rotate_head()

    def move(self):
        # Restaura la velocidad normal si el temporizador ha expirado
        if self.speed_timer > 0 and time.time() > self.speed_timer:
            self.speed = 0.1  # Velocidad normal
            self.speed_timer = 0

        # Mueve la serpiente
        for i in range(len(self.segments) - 1, 0, -1):
            x = self.segments[i - 1].xcor()
            y = self.segments[i - 1].ycor()
            self.segments[i].goto(x, y)

        # Mueve la cabeza
        if self.direction == "up":
            self.head.sety(self.head.ycor() + 20)
        elif self.direction == "down":
            self.head.sety(self.head.ycor() - 20)
        elif self.direction == "left":
```

```
            self.head.setx(self.head.xcor() - 20)
        elif self.direction == "right":
            self.head.setx(self.head.xcor() + 20)

    def go_up(self):
        if self.direction != "down":
            self.direction = "up"

    def go_down(self):
        if self.direction != "up":
            self.direction = "down"

    def go_left(self):
        if self.direction != "right":
            self.direction = "left"

    def go_right(self):
        if self.direction != "left":
            self.direction = "right"

    def check_collision(self):
        # Verifica colisión con los bordes
        if (self.head.xcor() > 290 or self.head.xcor() < -290 or
                self.head.ycor() > 290 or self.head.ycor() < -290):
            return True

        # Verifica colisión con el cuerpo
        for segment in self.segments[1:]:
            if self.head.distance(segment) < 20:
                return True

        return False

# Clase para la comida
class FoodManager:
    def __init__(self):
        # Lista de fábricas disponibles
        self.fabricas = [
            FabricaComidaVenenosa(),
            FabricaComidaFit(),
            FabricaComidaAltoEnGrasas(),
            FabricaComidaParaReyes()
```

```
        ]

        # Lista para almacenar múltiples comidas
        self.foods = []
        self.comidas_actuales = []

        # Número de comidas que aparecerán simultáneamente
        self.num_comidas = 3

        # Generar comidas iniciales
        for _ in range(self.num_comidas):
            self.generar_nueva_comida()

    def generar_nueva_comida(self):
        # Crea un nuevo objeto turtle para la comida
        food = turtle.Turtle()
        food.speed(0)
        food.shape("circle")
        food.penup()

        # Selecciona una fábrica aleatoria
        fabrica = random.choice(self.fabricas)
        comida_actual = fabrica.crear_comida()

        # Configura la comida con el color correspondiente
        food.color(comida_actual.obtener_color())

        # Posiciona la comida en un lugar aleatorio
        x = random.randint(-270, 270)
        y = random.randint(-270, 270)
        food.goto(x, y)

        # Añade la comida y su tipo a las listas
        self.foods.append(food)
        self.comidas_actuales.append(comida_actual)

    def check_collision(self, snake, score):
        # Verifica si la serpiente ha comido alguna de las comidas
        for i, food in enumerate(self.foods[:]):
            if snake.head.distance(food) < 20:
                # Aplica el efecto de la comida actual
```

```
                new_score = self.comidas_actuales[i].aplicar_efecto(snake,
score)

                # Elimina la comida comida
                food.goto(1000, 1000)  # Mueve fuera de la pantalla
                food.hideturtle()
                self.foods.remove(food)
                self.comidas_actuales.pop(i)

                # Genera una nueva comida para reemplazar la que se comió
                self.generar_nueva_comida()

                return True, new_score

        return False, score

    def obtener_descripciones(self):
        # Devuelve una lista con las descripciones de todas las comidas
actuales
        return [comida.obtener_descripcion() for comida in
self.comidas_actuales]

# Configuración del marcador
def setup_score_display():
    score_display = turtle.Turtle()
    score_display.speed(0)
    score_display.color("white")
    score_display.penup()
    score_display.hideturtle()
    score_display.goto(0, 260)
    return score_display

def setup_food_info_display():
    food_info = turtle.Turtle()
    food_info.speed(0)
    food_info.color("white")
    food_info.penup()
    food_info.hideturtle()
    food_info.goto(-280, -260)  # Cambiado para mejor legibilidad
    return food_info

# Función principal del juego
```

```
def reset_game():
    # Limpia todos los elementos existentes
    screen.clear()
    screen.bgcolor("black")

    # Reinicia el juego
    return iniciar_juego()

def iniciar_juego():
    # Asegura que las animaciones estén desactivadas durante la
inicialización
    screen.tracer(0)

    # Inicializa componentes
    snake = Snake()
    food_manager = FoodManager()
    score_display = setup_score_display()
    food_info = setup_food_info_display()

    # Configuración de controles
    screen.listen()
    screen.onkeypress(snake.go_up, "Up")
    screen.onkeypress(snake.go_down, "Down")
    screen.onkeypress(snake.go_left, "Left")
    screen.onkeypress(snake.go_right, "Right")
    screen.onkeypress(exit_game, "e")

    # Variables del juego
    score = 0
    game_over = False

    # Asegura que todo esté dibujado antes de continuar
    screen.update()

    # Bucle principal del juego
    while not game_over:
        screen.update()

        # Actualiza el marcador
        score_display.clear()
        score_display.write(f"Puntuación: {score}", align="center",
font=("Courier", 24, "normal"))
```

```
        # Muestra información de todas las comidas disponibles
        food_info.clear()
        descripciones = food_manager.obtener_descripciones()
        for i, descripcion in enumerate(descripciones):
            food_info.goto(-280, -260 + i * 20)
            food_info.write(descripcion, align="left", font=("Courier", 12,
"bold"))

        # Mueve la serpiente
        snake.move()

        # Verifica colisión con la comida
        food_eaten, score = food_manager.check_collision(snake, score)

        # Verifica colisión con los bordes o con el cuerpo
        if snake.check_collision():
            game_over = True
            # Muestra mensaje de fin de juego
            score_display.clear()
            score_display.write(f"¡Juego terminado!\nPuntuación final:
{score}",
                                align="center", font=("Courier", 24, "bold"))
            screen.update()
            time.sleep(1)  # Reducido a 1 segundo para una respuesta más
rápida
            return reset_game()

        # Pausa según la velocidad de la serpiente
        time.sleep(snake.speed)

def exit_game():
    screen.bye()

def main():
    iniciar_juego()

# Ejecuta el juego
if __name__ == "__main__":
    main()
    screen.exitonclick()
```

```python
# Ejecuta el juego
if __name__ == "__main__":
    main()

import turtle
import time
import random
from abc import ABC, abstractmethod

# Configuración de la pantalla
screen = turtle.Screen()
screen.title("Snake Game - Abstract Factory Pattern")
screen.bgcolor("black")
screen.setup(width=600, height=600)
screen.tracer(0)  # Apaga las animaciones automáticas

# Interfaz abstracta para la comida
class Comida(ABC):
    @abstractmethod
    def obtener_color(self):
        pass

    @abstractmethod
    def aplicar_efecto(self, snake, score):
        pass

    @abstractmethod
    def obtener_descripcion(self):
        pass

# Implementaciones concretas de comida
class ComidaVenenosa(Comida):
    def obtener_color(self):
        return "purple"

    def aplicar_efecto(self, snake, score):
        # Reduce el tamaño de la serpiente y el puntaje
        if len(snake.segments) > 1:
            segment = snake.segments.pop()
            segment.goto(1000, 1000)  # Mueve el segmento fuera de la
pantalla
        return max(0, score - 1)  # Reduce el puntaje pero no menos de 0
```

```
    def obtener_descripcion(self):
        return "☠ Comida Venenosa: -1 punto, -1 segmento"

class ComidaFit(Comida):
    def obtener_color(self):
        return "green"

    def aplicar_efecto(self, snake, score):
        # Aumenta el puntaje normalmente
        snake.add_segment()
        return score + 1

    def obtener_descripcion(self):
        return "☠ Comida Fit: +1 punto"

class ComidaAltoEnGrasas(Comida):
    def obtener_color(self):
        return "yellow"

    def aplicar_efecto(self, snake, score):
        # Aumenta el puntaje significativamente pero ralentiza la serpiente
        snake.add_segment()
        snake.add_segment()
        snake.add_segment()
        snake.speed = max(0.05, snake.speed * 1.5)  # Ralentiza la serpiente
        snake.speed_timer = time.time() + 5  # Efecto dura 5 segundos
        return score + 3

    def obtener_descripcion(self):
        return "☠ Comida Alto en Grasas: +3 puntos, velocidad reducida por 5
segundos"

class ComidaParaReyes(Comida):
    def obtener_color(self):
        return "orange"

    def aplicar_efecto(self, snake, score):
        # Otorga un bonus máximo y aumenta la velocidad
        for _ in range(5):
            snake.add_segment()
        snake.speed = max(0.01, snake.speed * 0.7)  # Aumenta la velocidad
```

```
        return score + 5

    def obtener_descripcion(self):
        return "🎁 Comida para Reyes: +5 puntos, velocidad aumentada"

# Interfaz de la fábrica abstracta
class FabricaComida(ABC):
    @abstractmethod
    def crear_comida(self):
        pass

# Implementaciones concretas de fábricas
class FabricaComidaVenenosa(FabricaComida):
    def crear_comida(self):
        return ComidaVenenosa()

class FabricaComidaFit(FabricaComida):
    def crear_comida(self):
        return ComidaFit()

class FabricaComidaAltoEnGrasas(FabricaComida):
    def crear_comida(self):
        return ComidaAltoEnGrasas()

class FabricaComidaParaReyes(FabricaComida):
    def crear_comida(self):
        return ComidaParaReyes()

# Clase para la serpiente
class Snake:
    def __init__(self):
        self.segments = []
        self.create_snake()
        self.head = self.segments[0]
        self.head.shape("square")  # Cabeza cuadrada
        self.head.color("green")  # Color verde
        self.speed = 0.1  # Velocidad inicial
        self.speed_timer = 0  # Temporizador para efectos temporales
        self.direction = "right"

        # Personalización de la cabeza
        self.head.color("lime green")  # Verde brillante para la cabeza
```

```
        self.head.shapesize(1.2, 1.2)  # Hacer la cabeza un poco más grande

        self.speed = 0.1
        self.speed_timer = 0
        self.direction = "right"
        self._rotate_head()

    def create_snake(self):
        for i in range(3):
            self.add_segment((-20 * i, 0))

    def add_segment(self, position=None):
        new_segment = turtle.Turtle()
        new_segment.speed(0)
        new_segment.shape("circle")  # Forma circular para el cuerpo

        # Si es el primer segmento (cabeza)
        if len(self.segments) == 0:
            new_segment.color("lime green")
        else:
            # Patrón de colores alternados para el cuerpo
            if len(self.segments) % 2 == 0:
                new_segment.color("dark green")
            else:
                new_segment.color("forest green")

        new_segment.penup()

        if position:
            new_segment.goto(position)
        elif len(self.segments) > 0:
            last_segment = self.segments[-1]
            new_segment.goto(last_segment.xcor(), last_segment.ycor())

        self.segments.append(new_segment)

    def _rotate_head(self):
        # Rota la cabeza según la dirección
        if self.direction == "up":
            self.head.setheading(90)
        elif self.direction == "down":
            self.head.setheading(270)
```

```
        elif self.direction == "left":
            self.head.setheading(180)
        elif self.direction == "right":
            self.head.setheading(0)

    def go_up(self):
        if self.direction != "down":
            self.direction = "up"
            self._rotate_head()

    def go_down(self):
        if self.direction != "up":
            self.direction = "down"
            self._rotate_head()

    def go_left(self):
        if self.direction != "right":
            self.direction = "left"
            self._rotate_head()

    def go_right(self):
        if self.direction != "left":
            self.direction = "right"
            self._rotate_head()

    def move(self):
        # Restaura la velocidad normal si el temporizador ha expirado
        if self.speed_timer > 0 and time.time() > self.speed_timer:
            self.speed = 0.1  # Velocidad normal
            self.speed_timer = 0

        # Mueve la serpiente
        for i in range(len(self.segments) - 1, 0, -1):
            x = self.segments[i - 1].xcor()
            y = self.segments[i - 1].ycor()
            self.segments[i].goto(x, y)

        # Mueve la cabeza
        if self.direction == "up":
            self.head.sety(self.head.ycor() + 20)
        elif self.direction == "down":
            self.head.sety(self.head.ycor() - 20)
```

```
        elif self.direction == "left":
            self.head.setx(self.head.xcor() - 20)
        elif self.direction == "right":
            self.head.setx(self.head.xcor() + 20)

    def go_up(self):
        if self.direction != "down":
            self.direction = "up"

    def go_down(self):
        if self.direction != "up":
            self.direction = "down"

    def go_left(self):
        if self.direction != "right":
            self.direction = "left"

    def go_right(self):
        if self.direction != "left":
            self.direction = "right"

    def check_collision(self):
        # Verifica colisión con los bordes
        if (self.head.xcor() > 290 or self.head.xcor() < -290 or
                self.head.ycor() > 290 or self.head.ycor() < -290):
            return True

        # Verifica colisión con el cuerpo
        for segment in self.segments[1:]:
            if self.head.distance(segment) < 20:
                return True

        return False

# Clase para la comida
class FoodManager:
    def __init__(self):
        # Lista de fábricas disponibles
        self.fabricas = [
            FabricaComidaVenenosa(),
            FabricaComidaFit(),
            FabricaComidaAltoEnGrasas(),
```

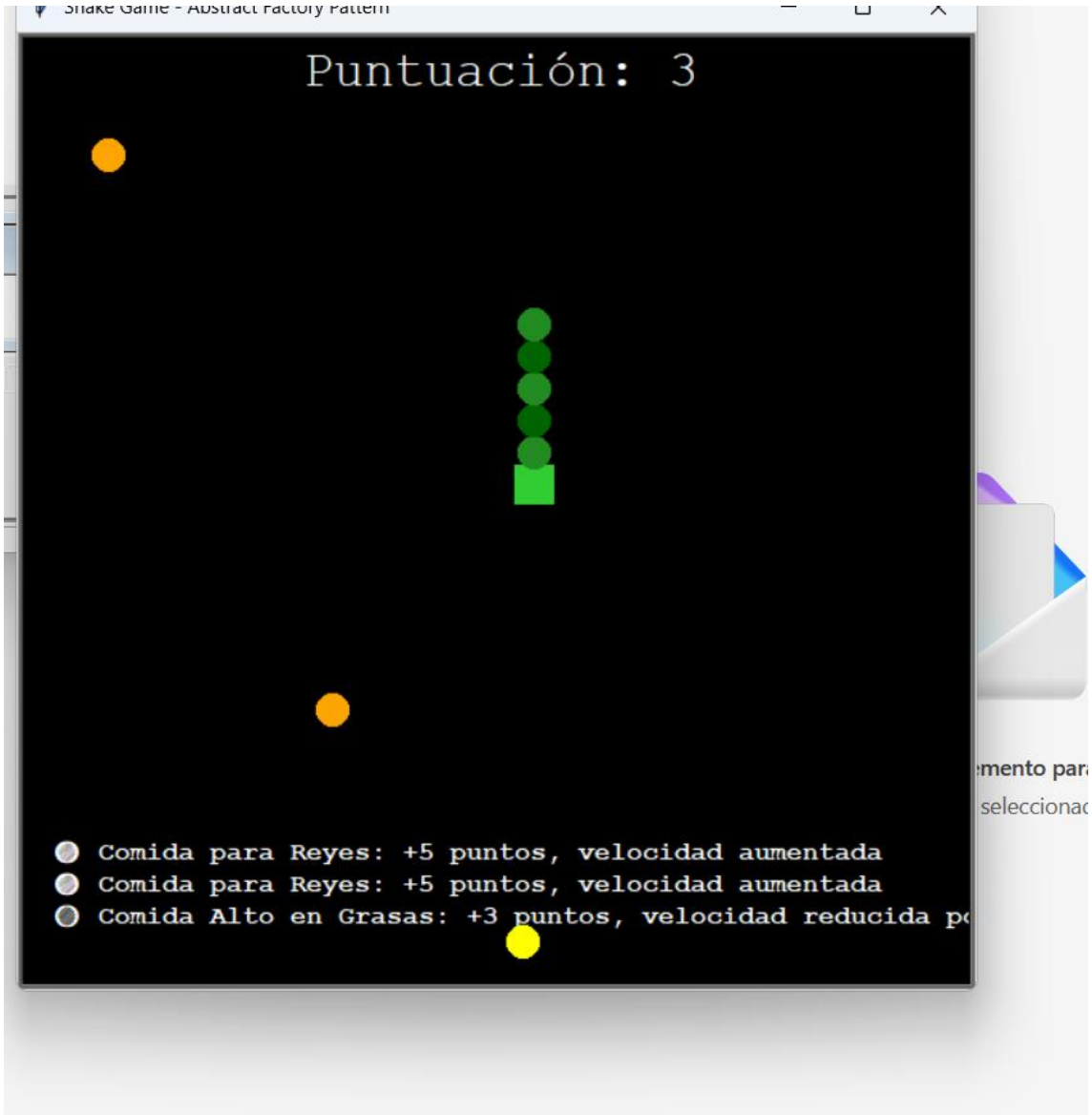
```
            FabricaComidaParaReyes()
        ]

        # Lista para almacenar múltiples comidas
        self.foods = []
        self.comidas_actuales = []

        # Número de comidas que aparecerán simultáneamente
        self.num_comidas = 3

        # Generar comidas iniciales
        for _ in range(self.num_comidas):
            self.generar_nueva_comida()

    def generar_nueva_comida(self):
        # Crea un nuevo objeto turtle para la comida
        food = turtle.Turtle()
        food.speed(0)
        food.shape("circle")
        food.penup()

        # Selecciona una fábrica aleatoria
        fabrica = random.choice(self.fabricas)
        comida_actual = fabrica.crear_comida()

        # Configura la comida con el color correspondiente
        food.color(comida_actual.obtener_color())

        # Posiciona la comida en un lugar aleatorio
        x = random.randint(-270, 270)
        y = random.randint(-270, 270)
        food.goto(x, y)

        # Añade la comida y su tipo a las listas
        self.foods.append(food)
        self.comidas_actuales.append(comida_actual)

    def check_collision(self, snake, score):
        # Verifica si la serpiente ha comido alguna de las comidas
        for i, food in enumerate(self.foods[:]):
            if snake.head.distance(food) < 20:
                # Aplica el efecto de la comida actual
```

```
                new_score = self.comidas_actuales[i].aplicar_efecto(snake,
score)

                # Elimina la comida comida
                food.goto(1000, 1000)  # Mueve fuera de la pantalla
                food.hideturtle()
                self.foods.remove(food)
                self.comidas_actuales.pop(i)

                # Genera una nueva comida para reemplazar la que se comió
                self.generar_nueva_comida()

                return True, new_score

        return False, score

    def obtener_descripciones(self):
        # Devuelve una lista con las descripciones de todas las comidas
actuales
        return [comida.obtener_descripcion() for comida in
self.comidas_actuales]

# Configuración del marcador
def setup_score_display():
    score_display = turtle.Turtle()
    score_display.speed(0)
    score_display.color("white")
    score_display.penup()
    score_display.hideturtle()
    score_display.goto(0, 260)
    return score_display

def setup_food_info_display():
    food_info = turtle.Turtle()
    food_info.speed(0)
    food_info.color("white")
    food_info.penup()
    food_info.hideturtle()
    food_info.goto(-280, -260)  # Cambiado para mejor legibilidad
    return food_info

# Función principal del juego
```

```
def reset_game():
    # Limpia todos los elementos existentes
    screen.clear()
    screen.bgcolor("black")

    # Reinicia el juego
    return iniciar_juego()

def iniciar_juego():
    # Asegura que las animaciones estén desactivadas durante la
inicialización
    screen.tracer(0)

    # Inicializa componentes
    snake = Snake()
    food_manager = FoodManager()
    score_display = setup_score_display()
    food_info = setup_food_info_display()

    # Configuración de controles
    screen.listen()
    screen.onkeypress(snake.go_up, "Up")
    screen.onkeypress(snake.go_down, "Down")
    screen.onkeypress(snake.go_left, "Left")
    screen.onkeypress(snake.go_right, "Right")
    screen.onkeypress(exit_game, "e")

    # Variables del juego
    score = 0
    game_over = False

    # Asegura que todo esté dibujado antes de continuar
    screen.update()

    # Bucle principal del juego
    while not game_over:
        screen.update()

        # Actualiza el marcador
        score_display.clear()
        score_display.write(f"Puntuación: {score}", align="center",
font=("Courier", 24, "normal"))
```

```
        # Muestra información de todas las comidas disponibles
        food_info.clear()
        descripciones = food_manager.obtener_descripciones()
        for i, descripcion in enumerate(descripciones):
            food_info.goto(-280, -260 + i * 20)
            food_info.write(descripcion, align="left", font=("Courier", 12,
"bold"))

        # Mueve la serpiente
        snake.move()

        # Verifica colisión con la comida
        food_eaten, score = food_manager.check_collision(snake, score)

        # Verifica colisión con los bordes o con el cuerpo
        if snake.check_collision():
            game_over = True
            # Muestra mensaje de fin de juego y opciones
            score_display.clear()
            score_display.write(f"¡Juego terminado!\nPuntuación final:
{score}\n\nPresiona 'r' para reiniciar\nPresiona 'e' para salir",
                                align="center", font=("Courier", 24, "bold"))
            screen.update()

            # Configura las teclas para reiniciar o salir
            screen.onkey(reset_game, "r")
            screen.onkey(exit_game, "e")
            screen.listen()

            # Mantén el juego en espera hasta que el jugador elija
            while True:
                screen.update()
                time.sleep(0.1)
            time.sleep(1)  # Reducido a 1 segundo para una respuesta más
rápida
            return reset_game()

        # Pausa según la velocidad de la serpiente
        time.sleep(snake.speed)

def exit_game():
```

```
    screen.bye()

def main():
    iniciar_juego()

# Ejecuta el juego
if __name__ == "__main__":
    main()
    screen.exitonclick()

# Ejecuta el juego
if __name__ == "__main__":
    main()
```




Snake Game - Abstract Factory Pattern
Presiona 'e' para salir
Comida Fit: +1 punto
Comida Alto en Grasas: +3 puntos, velocidad reducida p
Comida Venenosa: -1 punto, -1 segmento