

Programação Orientada por Objetos

Classes Abstratas e Interfaces - Exemplo

Prof. José Cordeiro,

Prof. Cédric Grueau,

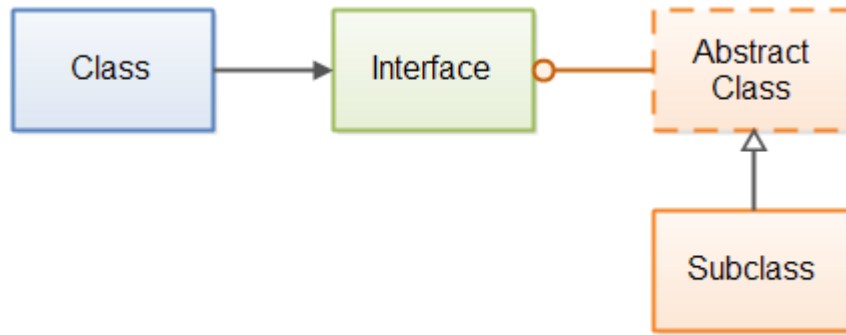
Prof. Laercio Júnior

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2019/2020

- ❑ Sessão 5: Exemplo Raposas e Coelhos – Melhorias no código
- ❑ Sessão 6: Exemplo Raposas e Coelhos – Nova funcionalidade
- ❑ Sessão 7: Desenho Casa – Melhorias e Nova Funcionalidade
- ❑ Sessão 8: Desenho Casa – Nova Solução



Módulo 3 – Classes Abstratas e Interfaces

SESSÃO 5 – EXEMPLO RAPOSAS E COELHOS – MELHORIAS NO CÓDIGO

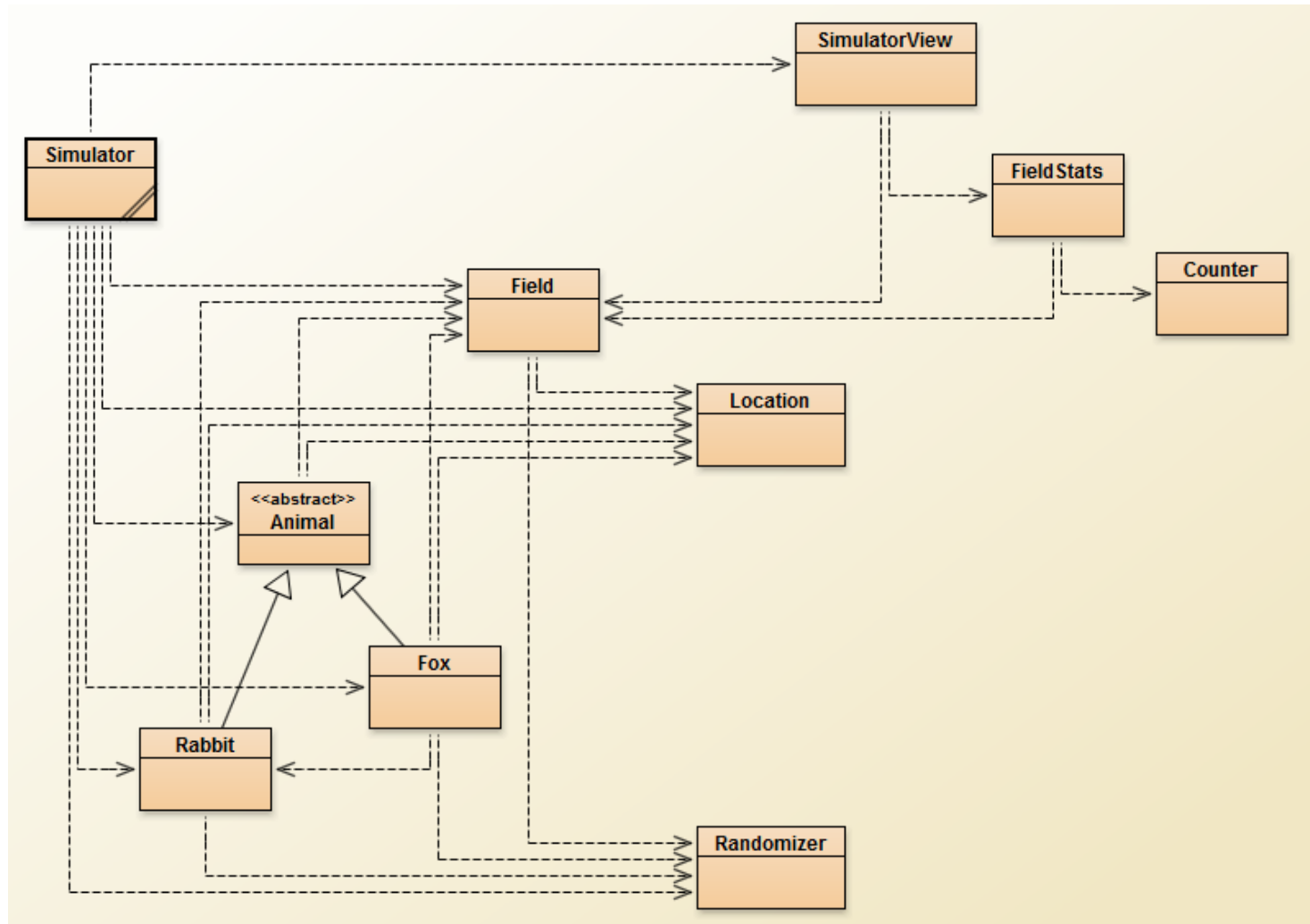
Exemplo — Foxes and Rabbits

- Requisitos da nova aplicação:
 - Melhorar o código.
 - Acrescentar um novo ator — o caçador.



Exemplo — Foxes and Rabbits

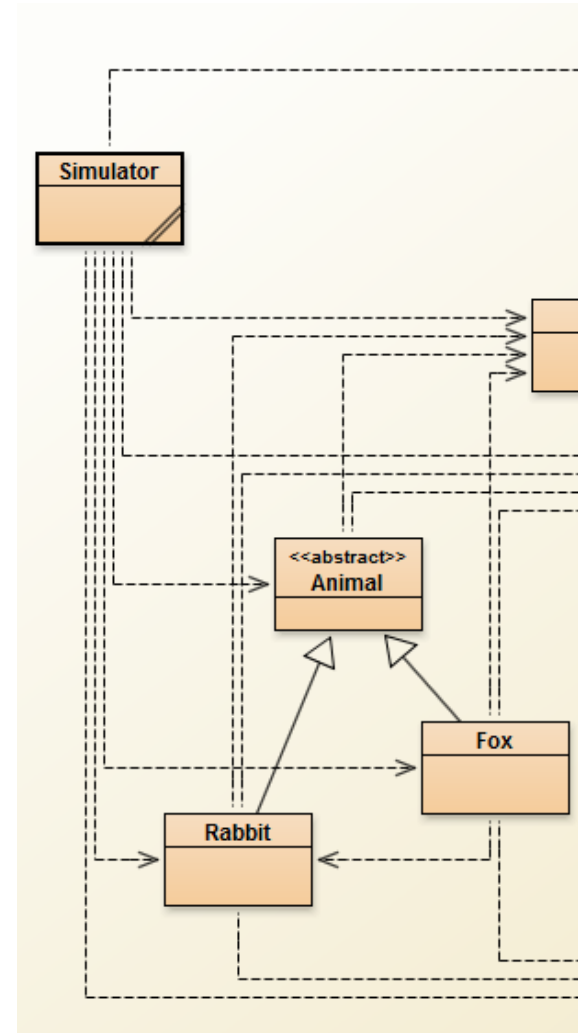
- Diagrama de classes da aplicação **Foxes and Rabbits**:



Exemplo — Foxes and Rabbits

- Diagrama de classes da aplicação **Foxes and Rabbits**:

Pelo diagrama de classes vê-se que existe uma dependência entre a classe **Simulator** e as classes **Rabbit** e **Fox**



Exemplo — Foxes and Rabbits

□ Classe **Simulator**

```
public class Simulator {  
    private static final int DEFAULT_WIDTH = 120;  
    private static final int DEFAULT_DEPTH = 80;  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    private List<Animal> animals;  
    private Field field;  
    private int step;  
    private SimulatorView view;  
  
    public void simulateOneStep() {  
        step++;  
  
        List<Animal> newAnimals = new ArrayList<Animal>();  
        for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {  
            Animal animal = it.next();  
            animal.act(newAnimals);  
            if(! animal.isAlive()) {  
                it.remove();  
            }  
        }  
        animals.addAll(newAnimals);  
  
        view.showStatus(step, field);  
    }  
    // Continua...
```

Estas constantes estão relacionadas com as classes Rabbit e Fox

Este método apenas refere a classe Animal não tendo ligações a Rabbit e/ou a Fox

Exemplo — Foxes and Rabbits

□ Classe **Simulator**

```
// Continuação da classe Simulator
```

```
public void reset() {  
    step = 0;  
    animals.clear();  
    populate();  
  
    view.showStatus(step, field);  
}
```

```
private void populate() {  
    Random rand = Randomizer.getRandom();  
    field.clear();  
    for(int row = 0; row < field.getDepth(); row++) {  
        for(int col = 0; col < field.getWidth(); col++) {  
            if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {  
                Location location = new Location(row, col);  
                Fox fox = new Fox(true, field, location);  
                animals.add(fox);  
            }  
            else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {  
                Location location = new Location(row, col);  
                Rabbit rabbit = new Rabbit(true, field, location);  
                animals.add(rabbit);  
            }  
            // else leave the location empty.  
        }  
    }  
}
```

Para (re)iniciar a simulação. Não tem as dependências encontradas

Para criar e distribuir a população inicial de raposas e coelhos

Aqui estão as dependências às classes **Rabbit e **Fox****

Exemplo — Foxes and Rabbits

□ Classe **Simulator** (Dependências)

```
public class Simulator {  
    // Código omitido  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    private List<Animal> animals;  
    private Field field;  
    private int step;  
    private SimulatorView view;  
  
    private void populate() {  
        Random rand = Randomizer.getRandom();  
        field.clear();  
        for(int row = 0; row < field.getDepth(); row++) {  
            for(int col = 0; col < field.getWidth(); col++) {  
                if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Fox fox = new Fox(true, field, location);  
                    animals.add(fox);  
                }  
                else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Rabbit rabbit = new Rabbit(true, field, location);  
                    animals.add(rabbit);  
                }  
            }  
        }  
    }  
}
```

**1. As constantes
estão relacionadas
com as classes Fox e
Rabbit**

**2. São criadas
instâncias das
classes Fox e
Rabbit**

Exemplo — Foxes and Rabbits

□ Classe **Simulator** (Dependências)

```
public class Simulator {  
    // Código omitido  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    private List<Animal> animals;  
    private Field field;  
    private int step;  
    private SimulatorView view;  
  
    private void populate() {  
        Random rand = Randomizer.getRandom();  
        field.clear();  
        for(int row = 0; row < field.getDepth(); row++) {  
            for(int col = 0; col < field.getWidth(); col++) {  
                if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Fox fox = new Fox(true, field, location);  
                    animals.add(fox);  
                }  
                else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Rabbit rabbit = new Rabbit(true, field, location);  
                    animals.add(rabbit);  
                }  
            }  
        }  
    }  
}
```

**Problema: Como
eliminar esta
dependência?**

**Solução: Criar
uma classe
separada para a
inicialização**

Exemplo — Foxes and Rabbits

□ Nova Classe **PopulationGenerator**

```
public class PopulationGenerator {  
    private static final double FOX_CREATION_PROBABILITY = 0.02;  
    private static final double RABBIT_CREATION_PROBABILITY = 0.08;  
  
    public PopulationGenerator(SimulatorView view) {  
        view.setColor(Rabbit.class, Color.orange);  
        view.setColor(Fox.class, Color.blue);  
    }  
  
    public void populate(Field field, List<Animal>animals) {  
        Random rand = Randomizer.getRandom();  
        for(int row = 0; row < field.getDepth(); row++) {  
            for(int col = 0; col < field.getWidth(); col++) {  
                if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Fox fox = new Fox(true, field, location);  
                    animals.add(fox);  
                }  
                else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {  
                    Location location = new Location(row, col);  
                    Rabbit rabbit = new Rabbit(true, field, location);  
                    animals.add(rabbit);  
                }  
            }  
        }  
    }  
}
```

**As constantes
também são
transferidas**

**A classe criada
recebe o método
de inicialização**

Exemplo — Foxes and Rabbits

□ Alterada a classe **Simulator**

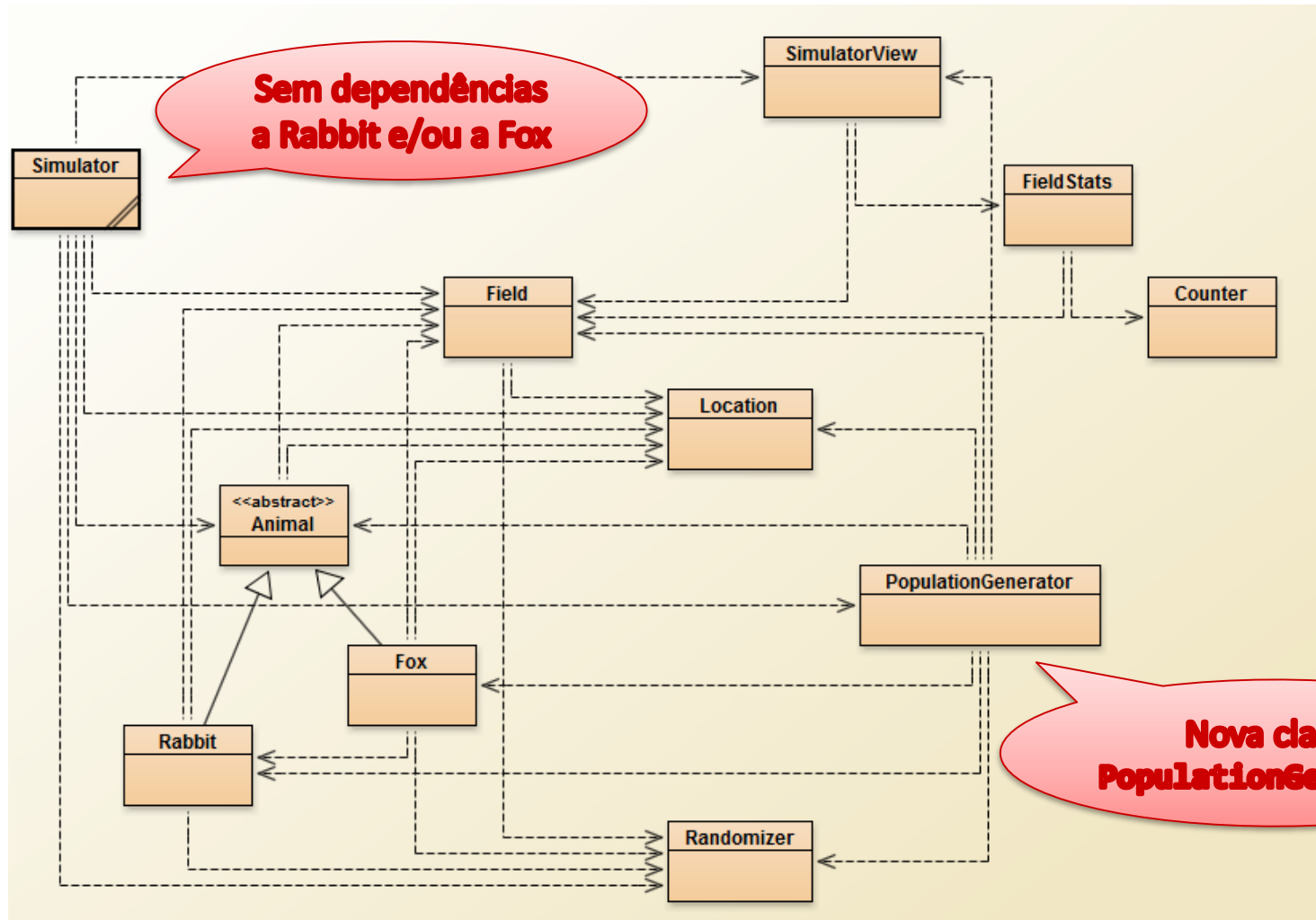
```
public class Simulator {  
    private static final int DEFAULT_WIDTH = 120;  
    private static final int DEFAULT_DEPTH = 80;  
  
    private List<Animal> animals;  
    private Field field;  
    private int step;  
    private SimulatorView view;  
    private PopulationGenerator populador;  
  
    public void reset() {  
        step = 0;  
        animals.clear();  
        field.clear();  
        populador.populate(field, animals);  
  
        view.showStatus(step, field);  
    }  
}
```

Novo atributo **populador**

Método populate passa a inicializar a distribuição das raposas e coelhos

Exemplo — Foxes and Rabbits

- Diagrama de classes da aplicação **Foxes and Rabbits**:



Exemplo — Foxes and Rabbits

□ Continuação da análise às Classes **Fox** e **Rabbit**

```
public class Fox {  
  
    // código omitido  
  
    private void giveBirth(List<Animal> newFoxes) {  
  
        Field field = getField();  
        List<Location> free =  
            field.getFreeAdjacentLocations(getLocation());  
        int births = breed();  
        for(int b=0; b<births && free.size() > 0; b++) {  
            Location loc = free.remove(0);  
            Fox young = new Fox(false, field, loc);  
            newFoxes.add(young);  
        }  
    }  
  
    // continua...
```



```
public class Rabbit {  
  
    // código omitido  
  
    private void giveBirth(List<Animal> newRabbits) {  
  
        Field field = getField();  
        List<Location> free =  
            field.getFreeAdjacentLocations(getLocation());  
        int births = breed();  
        for(int b=0; b<births && free.size() > 0; b++) {  
            Location loc = free.remove(0);  
            Rabbit young = new Rabbit(false, field, loc);  
            newRabbits.add(young);  
        }  
    }  
  
    // continua...
```

- Embora o código dos métodos seja diferente é possível passa-los para a classe **Animal** tirando partido do polimorfismo.

Exemplo — Foxes and Rabbits

□ Nova classe **Animal**

```
public abstract class Animal {
```

```
    abstract public int getBreedingAge();  
    abstract public double getBreedingProbability();  
    abstract public int getMaxLitterSize();
```

```
    abstract protected Animal createAnimal(boolean randomAge,  
                                           Field field, Location location);  
    protected void giveBirth(List<Animal> newborn) {  
        Field field = getField();  
        List<Location> free = field.getFreeAdjacentLocations(getLocation());  
        int births = breed();  
        for(int b = 0; b < births && free.size() > 0; b++) {  
            Location loc = free.remove(0);  
            newborn.add(createAnimal(false, field, loc));  
        }  
    }
```

```
    protected int breed() {  
        int births = 0;  
        if(canBreed() && rand.nextDouble() <= getBreedingProbability()) {  
            births = rand.nextInt(getMaxLitterSize()) + 1;  
        }  
        return births;  
    }  
}
```

Novos métodos abstratos

createAnimal é abstrato
e retorna um **Animal**

Usa o método **createAnimal**
que será implementado de forma
diferente em **Fox** e **Rabbit**

Usado em **giveBirth**

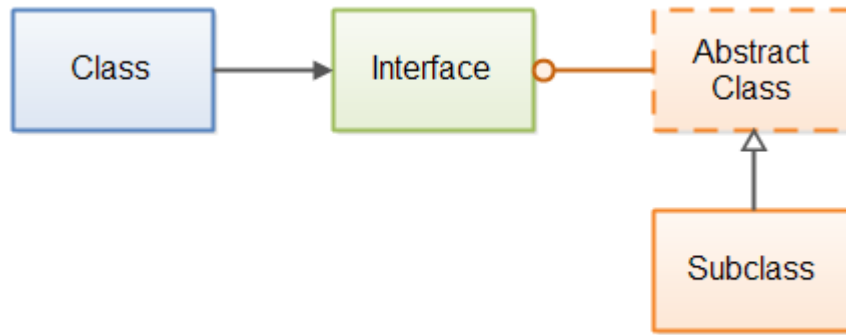
Exemplo — Foxes and Rabbits

□ Nova classe **Rabbit**

```
public class Rabbit extends Animal {  
    // Restante código omitido  
  
    public int getBreedingAge() {  
        return BREEDING_AGE;  
    }  
  
    public double getBreedingProbability() {  
        return BREEDING_PROBABILITY;  
    }  
  
    public int getMaxLitterSize() {  
        return MAX_LITTER_SIZE;  
    }  
  
    protected Animal createAnimal(boolean randomAge, Field field, Location location) {  
        return new Rabbit(randomAge, field, location);  
    }  
}
```

**Implementação dos
novos métodos
abstratos**

O método **createAnimal** retorna um objeto **Rabbit** embora o tipo de retorno seja **Animal**. Isto é possível pelo princípio da substituição



Módulo 3 – Classes Abstratas e Interfaces

SESSÃO 6 – EXEMPLO RAPOSAS E COELHOS – NOVA FUNCIONALIDADE

Exemplo — Foxes and Rabbits

- Acrescentar a nova classe **Hunter**
- Neste caso vai introduzir-se a interface **Actor** como foi sugerido anteriormente

```
public interface Actor {  
    void act(List<Actor> newActors);  
    boolean isActive();  
}
```

- A classe **Animal** passa a implementar a interface **Actor**
- A classe **Simulator** passa a ter uma lista de **Actor** em vez de **Animal**

Exemplo — Foxes and Rabbits

□ Nova classe **Animal**

```
public abstract class Animal implements Actor {
```

Implementa a interface **Actor**

```
    abstract public void act(List<Actor> newAnimals);
```

```
    protected void giveBirth(List<Actor> newborn) {
```

A lista agora é de **Actor**

```
        Field field = getField();
```

```
        List<Location> free = field.getFreeAdjacentLocations(getLocation());
```

```
        int births = breed();
```

```
        for(int b = 0; b < births && free.size() > 0; b++) {
```

```
            Location loc = free.remove(0);
```

```
            newborn.add(createAnimal(false, field, loc));
```

```
        }
```

```
    }
```

Este método também passa a receber uma lista de **Actor**

```
    protected int breed() {
```

```
        int births = 0;
```

```
        if(canBreed() && rand.nextDouble() <= getBreedingProbability()) {
```

```
            births = rand.nextInt(getMaxLitterSize()) + 1;
```

```
        }
```

```
        return births;
```

```
    }
```

```
} // restante código omitido
```

Exemplo — Foxes and Rabbits

□ Nova classe **Simulator**

```
public class Simulator {
```

```
    private List<Actor> animals;  
    // restantes atributos e constants omitidos
```

```
    public void simulateOneStep() {  
        step++;  
        List<Actor> newAnimals = new ArrayList<Actor>();  
        for(Iterator<Actor> it = animals.iterator(); it.hasNext(); ) {  
            Actor animal = it.next();  
            animal.act(newAnimals);  
            if(! animal.isActive()) {  
                it.remove();  
            }  
        }  
        animals.addAll(newAnimals);  
        view.showStatus(step, field);  
    }
```

```
    public void reset() {  
        step = 0;  
        animals.clear();  
        field.clear();  
        populater.populate(field, animals);  
        view.showStatus(step, field);  
    }  
} // restante código omitido
```

A lista agora é de **Actor**

Itera os objetos que implementam a interface **Actor**

Chama apenas os métodos **act** e **isActive** definidos na interface **Actor**

Exemplo — Foxes and Rabbits

□ Nova classe **Hunter**

```
public class Hunter implements Actor {
    private static final int MAX_KILLS = 4;
    private Field field;
    private Location location;

    public Hunter(Field field, Location location) {
        this.field = field;
        setLocation(location);
    }

    public boolean isActive() {
        return true;
    }

    public void setLocation(Location newLocation) {
        if(location != null) {
            field.clear(location);
        }
        location = newLocation;
        field.place(this, newLocation);
    }
    // continua...
```

Exemplo — Foxes and Rabbits

□ Nova classe **Hunter**

// continuação da classe Hunter...

```
public void act(List<Actor> newActors)    {
    int kills = 0;
    List<Location> adjacent = field.adjacentLocations(location);
    Iterator<Location> it = adjacent.iterator();

    while(it.hasNext() && kills < MAX_KILLS) {
        Location where = it.next();
        Object actor = field.getObjectAt(where);
        if(actor instanceof Animal) {
            Animal animal = (Animal) actor;
            animal.setDead();
            kills++;
        }
    }
    Location newLocation = field.freeAdjacentLocation(location);
    if(newLocation != null) {
        setLocation(newLocation);
    }
}
```

Tem de fazer *cast* para *Animal* para poder chamar o método **setDead**

Exemplo — Foxes and Rabbits

□ Nova classe **PopulationGenerator**

```
public class PopulationGenerator {
    private static final double HUNTER_CREATION_PROBABILITY = 0.01;
    // outras constantes omitidas

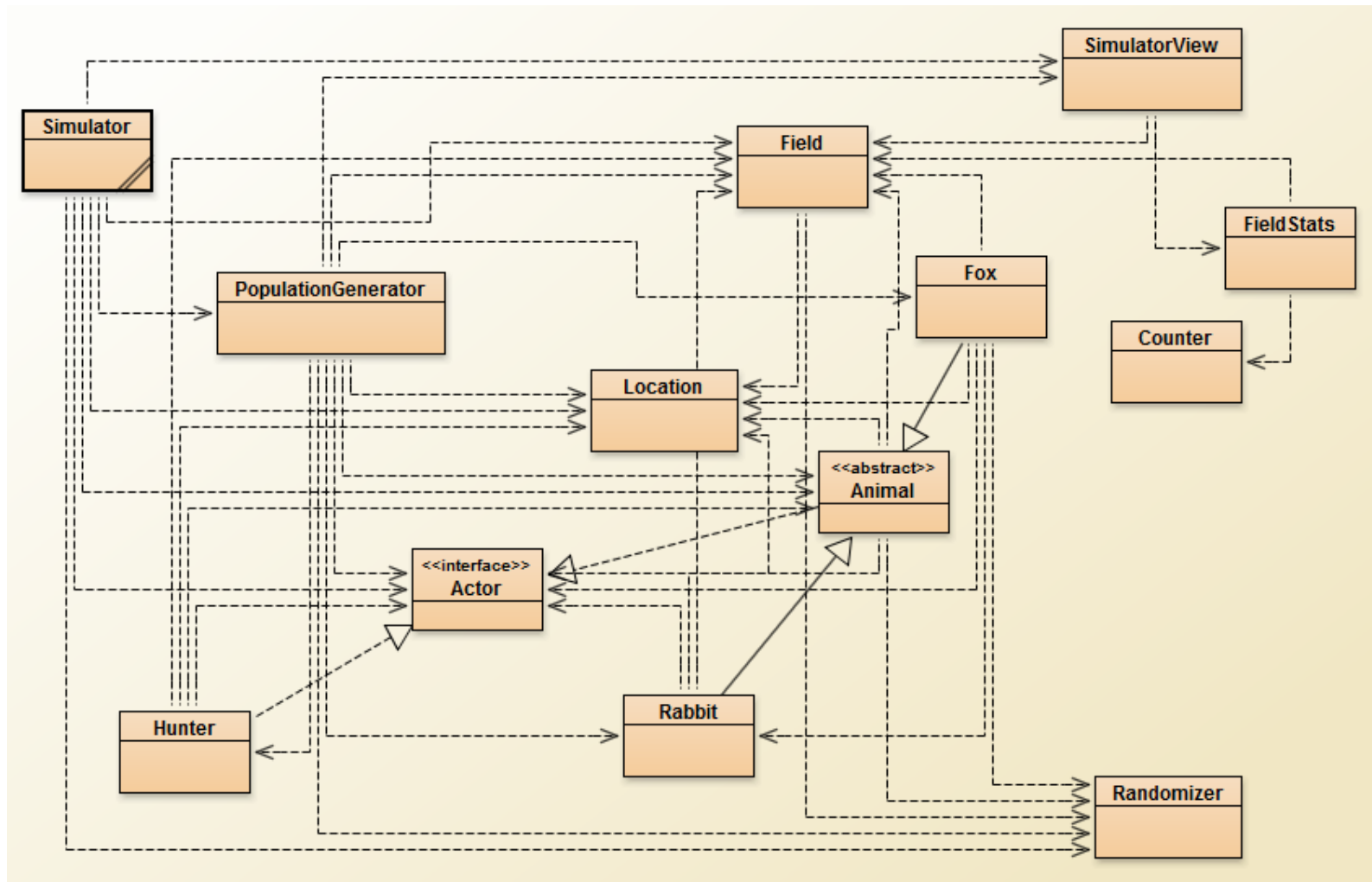
    public PopulationGenerator(SimulatorView view) {
        view.setColor(Hunter.class, Color.red);
        // código omitido
    }

    public void populate(Field field, List<Actor> actors) {
        Random rand = Randomizer.getRandom();
        for(int row = 0; row < field.getDepth(); row++) {
            for(int col = 0; col < field.getWidth(); col++) {
                if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                    Location location = new Location(row, col);
                    Animal fox = new Fox(true, field, location);
                    actors.add(fox);
                }
                else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {
                    Location location = new Location(row, col);
                    Animal rabbit = new Rabbit(true, field, location);
                    actors.add(rabbit);
                }
                else if(rand.nextDouble() <= HUNTER_CREATION_PROBABILITY) {
                    Location location = new Location(row, col);
                    Hunter hunter = new Hunter(field, location);
                    actors.add(hunter);
                }
            }
        }
    }
}
```

**Os caçadores ficam
com a cor vermelha
na nova simulação**

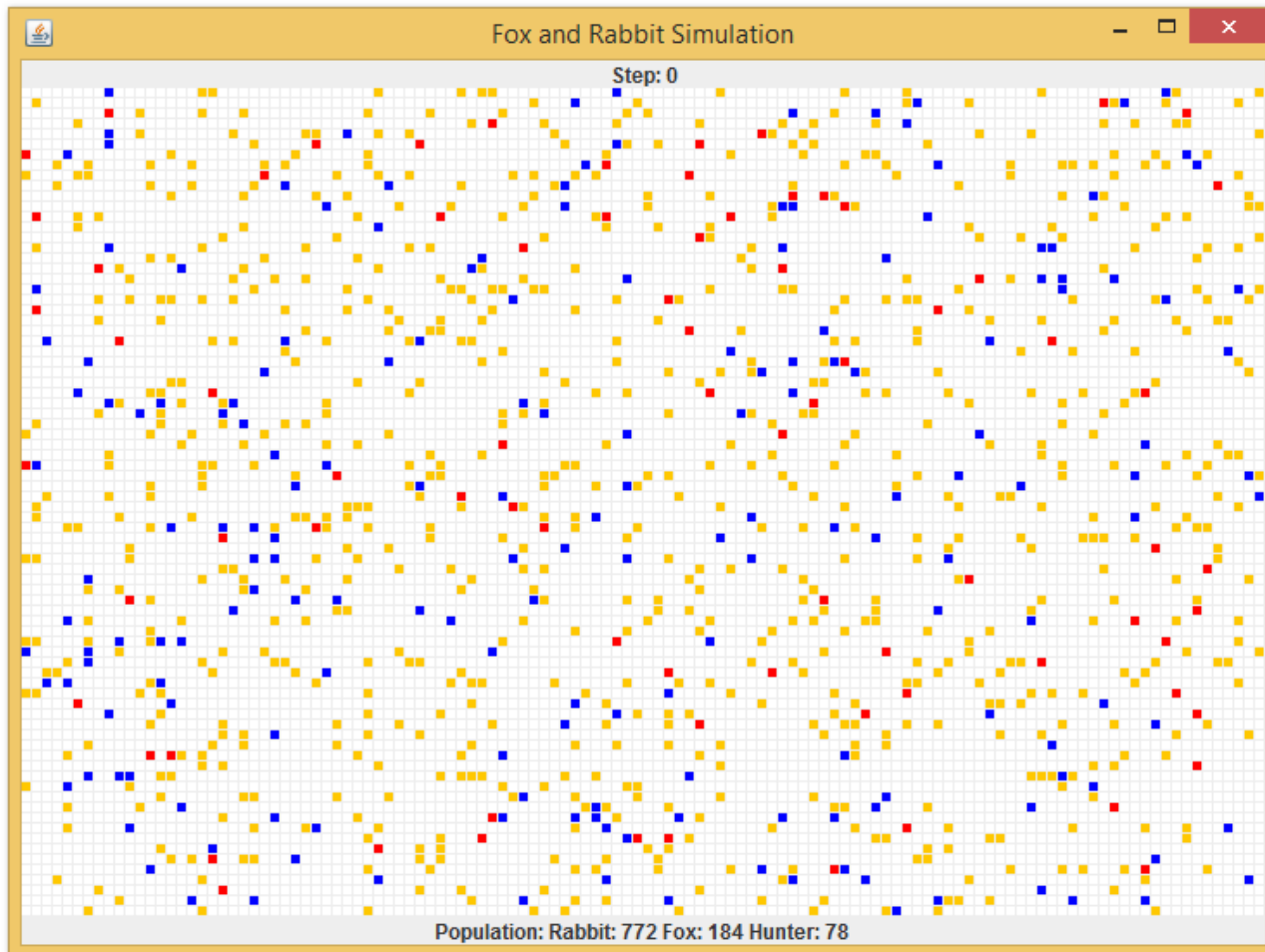
Exemplo — Foxes and Rabbits

- Diagrama de classes da **nova versão da aplicação**



Exemplo — Foxes and Rabbits

- Visualização da **nova versão da aplicação**





Módulo 3 – Classes Abstratas e Interfaces

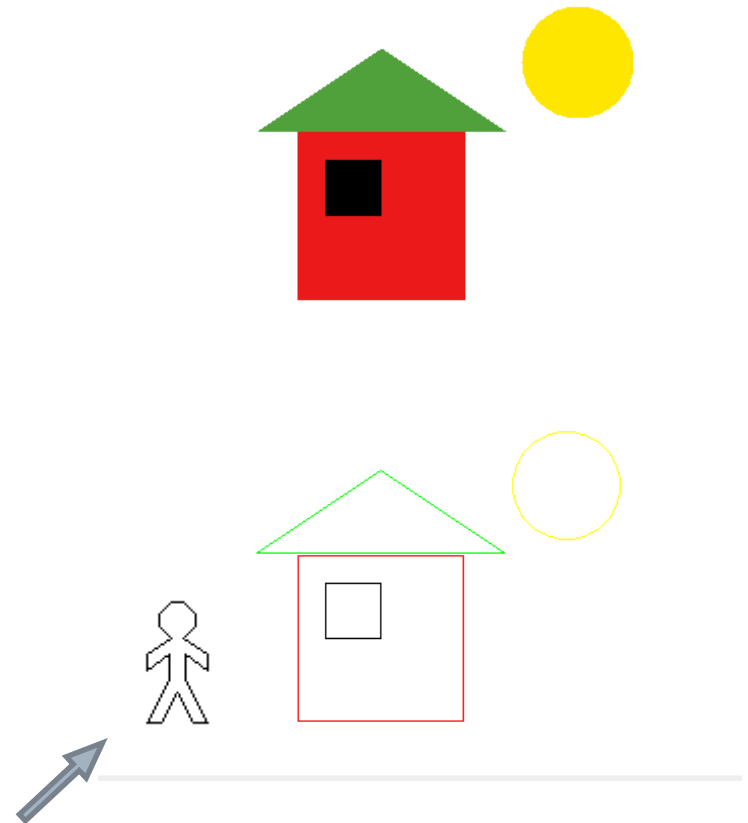
SESSÃO 7 – DESENHO CASA – MELHORIAS E NOVA FUNCIONALIDADE

Exemplo — Desenho Casa

□ Requisitos do desenho

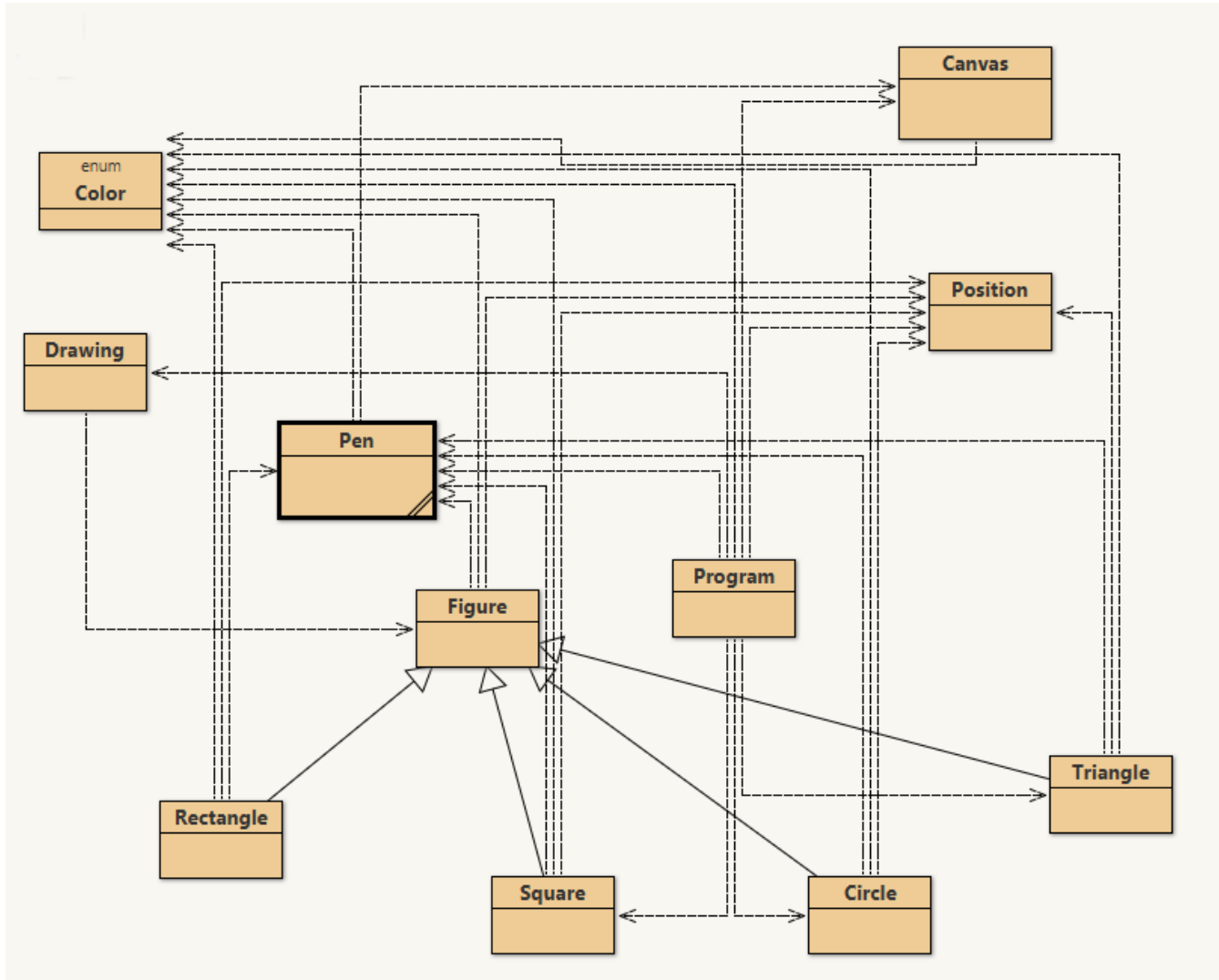
(Revisitado):

- O programa existente permite desenhar figuras geométricas.
- Pretende-se agora adicionar a representação de pessoas como um novo tipo de figura



Exemplo — Desenho casa

■ Diagrama de classes da solução existente



Exemplo — Desenho Casa

- Classes relacionadas com **figuras geométricas** da solução existente:

```
public class Figure {  
  
    private Pen pen;  
    private Position position;  
    private Color color;  
  
    // restante código  
}
```

```
public class Rectangle extends Figure {  
  
    private int width;  
    private int height;  
  
    // restante código  
}
```

```
public class Square extends Figure {  
  
    private int side;  
  
    // restante código  
}
```

```
public class Circle extends Figure {  
  
    private int radius;  
  
    // restante código  
}
```

```
public class Triangle extends Figure {  
  
    private int width;  
    private int height;  
  
    // restante código  
}
```

Exemplo — Desenho casa

- Solução existente — classe **Drawing**

```
public class Drawing {  
  
    private ArrayList<Figure> figures;  
  
    public Drawing() {  
        figures = new ArrayList<>();  
    }  
  
    public void addFigure(Figure figure) {  
        if (figure != null) {  
            figures.add(figure);  
        }  
    }  
  
    public void draw() {  
        for (Figure figure : figures) {  
            figure.draw();  
        }  
    }  
}
```

Usa o polimorfismo do método draw no desenho das formas geométricas guardadas na lista

Exemplo — Desenho Casa

■ Solução existente — classe base **Figure**

```
public class Figure {  
  
    private Pen pen;  
    private Position position;  
    private Color color;  
  
    public Figure() {  
        this(new Position(), new Pen(), Color.BLACK);  
    }  
  
    public Figure(Pen pen, Color color) {  
        this(new Position(), pen, color);  
    }  
  
    public Figure(Position position, Pen pen, Color color) {  
        this.pen = (pen != null) ? pen : new Pen();  
        this.position = (position != null) ? new Position(position.getX(),  
                                                             position.getY()) : new Position();  
        this.color = (color != null) ? color : Color.BLACK;  
    }  
  
    public void draw() {  
    }  
    // restante código  
}
```

Tendo em conta que se pretende que o método **draw** seja **exportado para a hierarquia** e que não se querem criar objetos da classe **Figure** pode **tornar-se este método abstrato**

Exemplo — Desenho Casa

■ Solução existente — classe base **Figure**

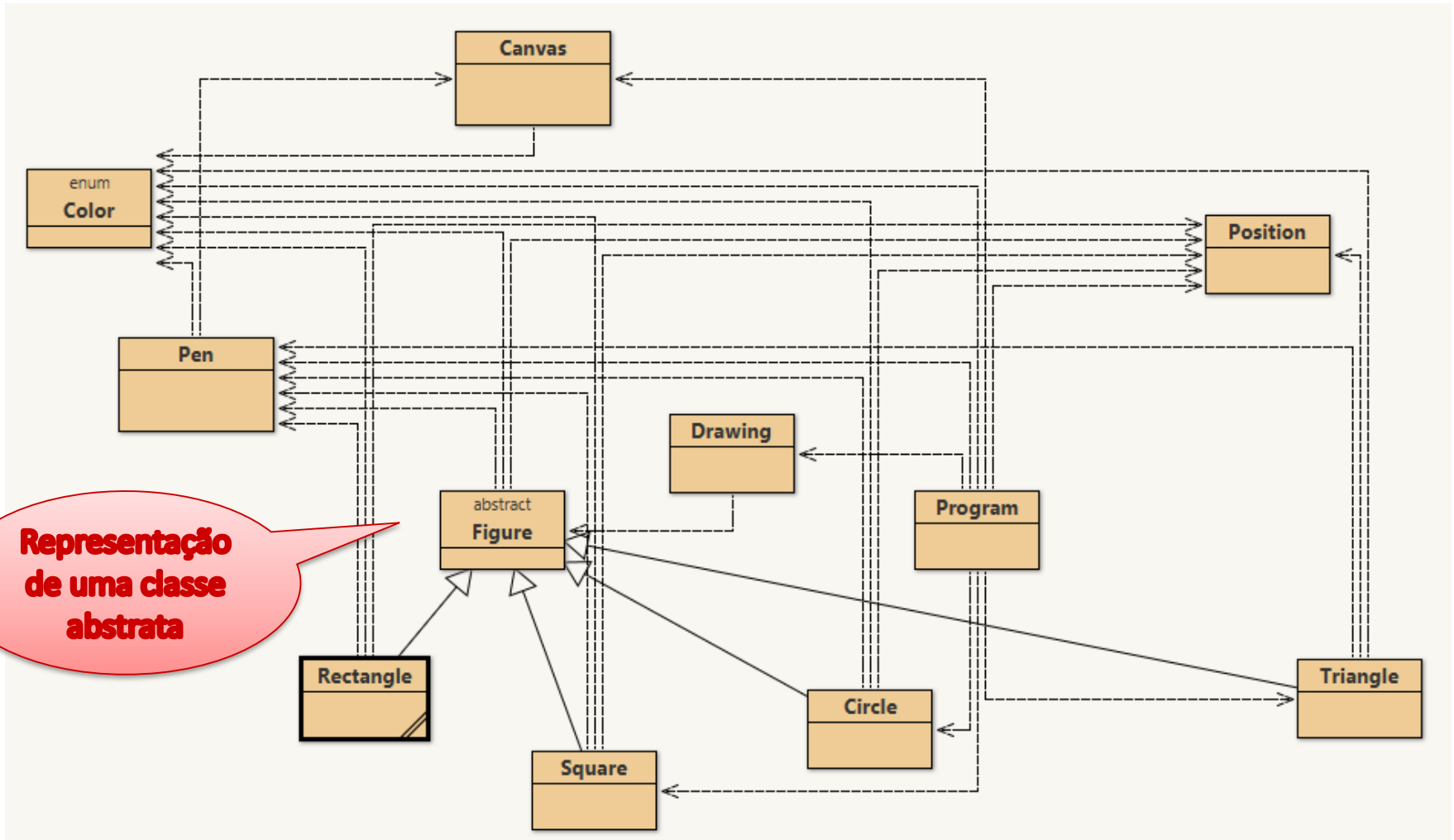
```
public abstract class Figure {  
  
    private Pen pen;  
    private Position position;  
    private Color color;  
  
    public Figure() {  
        this(new Position(), new Pen(), Color.BLACK);  
    }  
  
    public Figure(Pen pen, Color color) {  
        this(new Position(), pen, color);  
    }  
  
    public Figure(Position position, Pen pen, Color color) {  
        this.pen = (pen != null) ? pen : new Pen();  
        this.position = (position != null) ? new Position(position.getX(),  
                                                             position.getY()) : new Position();  
        this.color = (color != null) ? color : Color.BLACK;  
    }  
  
    public abstract void draw();  
  
    // restante código  
}
```

**A classe passa a ser
obrigatoriamente
abstrata**

**Fica sem código
e a terminar em
ponto e vírgula**

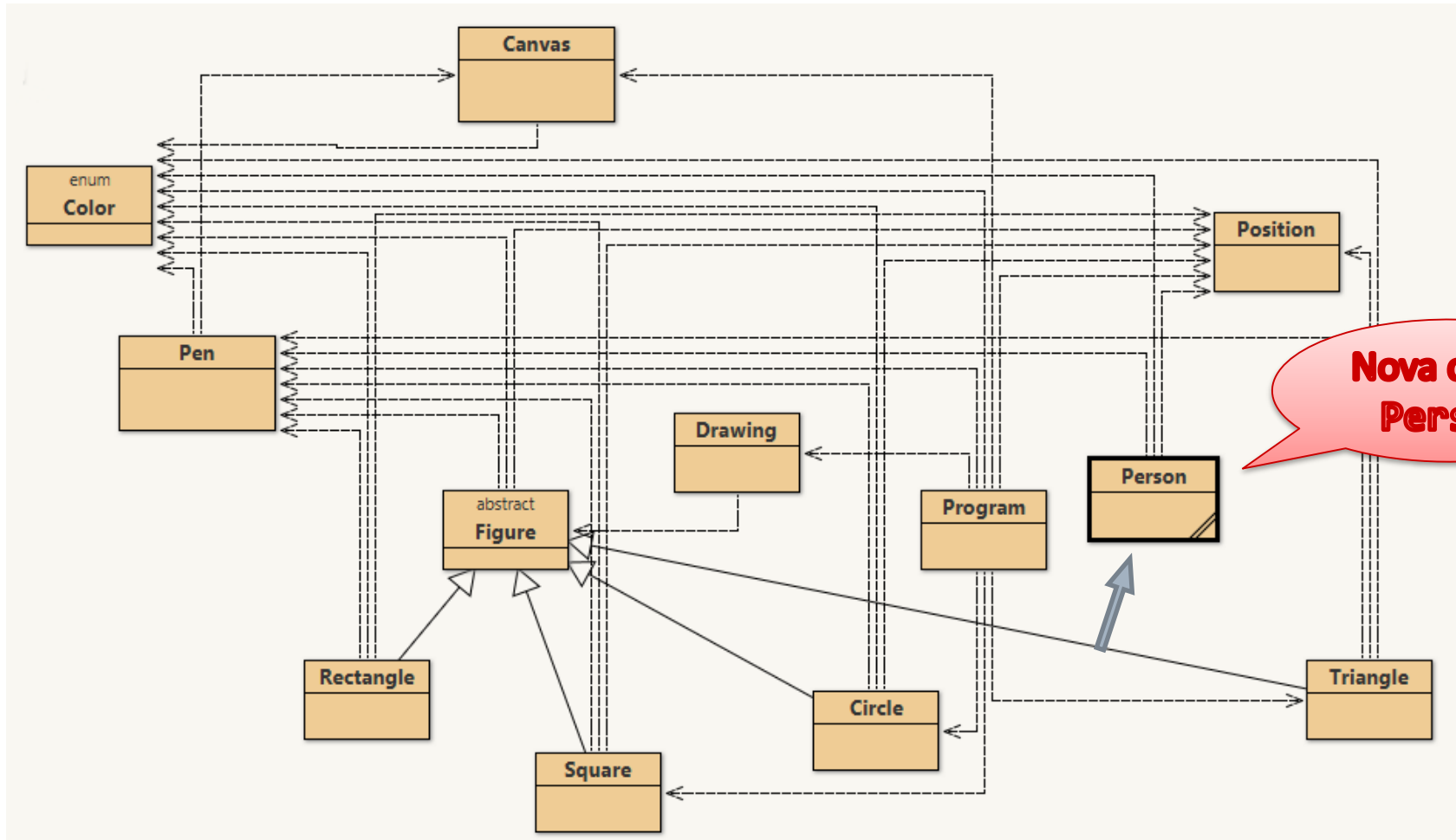
Exemplo — Desenho casa

■ Diagrama de classes da solução



Exemplo — Desenho casa

■ Diagrama de classes da nova solução



Exemplo — Desenho Casa

■ Classe **Person**

```
public class Person {  
  
    private int height;  
    private int width;  
    private Position position;  
    private Color color;  
    private Pen pen;  
  
    public Person() {  
        this(new Pen());  
    }  
  
    public Person(Pen pen) {  
        height = 60;  
        width = 30;  
        position = new Position(280, 190);  
        color = Color.BLACK;  
        this.pen = (pen != null) ? pen : new Pen();  
    }  
  
    // restante código  
}
```



**Utiliza a pen
para o desenho**

Exemplo — Desenho Casa

■ Classe Person - método **draw**

```
public void draw() {  
    int bh = (int) (height * 0.7);  
    int hh = (height - bh) / 2;  
    int hw = width / 2;  
    int x = position.getX();  
    int y = position.getY();  
  
    int[] xpoints = {x - 3, x - hw, x - hw, x - (int) (hw * 0.2) - 1, x - (int) (hw * 0.2) - 1, x - hw,  
        x - hw + (int) (hw * 0.4) + 1, x, x + hw - (int) (hw * 0.4) - 1, x + hw, x + (int) (hw * 0.2) + 1,  
        x + (int) (hw * 0.2) + 1, x + hw, x + hw, x + 3, x + (int) (hw * 0.6),  
        x + (int) (hw * 0.6), x + 3, x - 3, x - (int) (hw * 0.6), x - (int) (hw * 0.6)};  
    int[] ypoints = {y, y + (int) (bh * 0.2), y + (int) (bh * 0.4), y + (int) (bh * 0.2),  
        y + (int) (bh * 0.5), y + bh, y + bh, y + (int) (bh * 0.65), y + bh, y + bh,  
        y + (int) (bh * 0.5), y + (int) (bh * 0.2), y + (int) (bh * 0.4), y + (int) (bh * 0.2),  
        y, y - hh + 3, y - hh - 3, y - hh - hh, y - hh - hh, y - hh - 3, y - hh + 3};  
  
    pen.penUp();  
    pen.moveTo(xpoints[0], ypoints[0]);  
    pen.penDown();  
  
    for (int i = 1; i < xpoints.length; i++) {  
        pen.moveTo(xpoints[i], ypoints[i]);  
    }  
    pen.moveTo(xpoints[0], ypoints[0]);  
}
```

**Pontos de um
polígono com o
contorno da pessoa**

Exemplo — Desenho Casa

■ Classe Person – métodos **seletores e modificadores**

```
public void setSize(int height, int width) {
    this.height = height;
    this.width = width;
}

public void setColor(Color color) {
    if (color != null) { this.color = color; }
}

public Color getColor() { return color; }

public int getX() { return position.getX(); }

public int getY() { return position.getY(); }

public void setX(int x) { position.setX(x); }

public void setY(int y) { position.setY(y); }

public Position getPosition() {
    return new Position(position.getX(), position.getY());
}

public void setPosition(Position position) {
    if (position != null) {
        this.position = new Position(position.getX(), position.getY());
    }
}
```

Exemplo — Desenho Casa

❑ Problema

■ Solução existente

- ❑ O desenho é representado por uma lista de figuras geométricas
- ❑ O desenho tira partido do polimorfismo do método **draw** para o desenho das várias figuras geométricas

■ Nova Solução

- ❑ É necessário incluir a nova classe **Person** nos desenhos
- ❑ Uma pessoa não é uma figura geométrica pelo que não deveria ser incluída na lista de figuras geométricas que compõem um desenho



Módulo 3 – Classes Abstratas e Interfaces

SESSÃO 8 – DESENHO CASA – NOVA SOLUÇÃO

Exemplo — Desenho Casa

□ Solução

■ Solução 1:

- Criar uma classe base **GraphicElement** e fazer as classes **Figure** e **Person** herdarem desta classe. O desenho passaria a ser uma lista de elementos gráficos.
 - Poderia ser uma boa solução se quiséssemos apenas utilizar a classe **Person** apenas para os desenhos.

■ Solução 2

- Criar uma interface **Drawable** com o método **draw** e implementar esta interface nas classes **Figure** e **Person**.
 - Vamos optar por esta solução.

Exemplo — Novo Desenho Casa

- Interface **Drawable**

```
public interface Drawable {  
    public void draw();  
}
```

- Basta incluir o método **draw**
 - é a funcionalidade que queremos ter nos objetos que irão aparecer nos desenhos
 - Todas as classes que implementarem o método desenhar terão esta nova funcionalidade

Exemplo — Novo Desenho Casa

■ Classe **Person**

```
public class Person implements Drawable {  
  
    private int height;  
    private int width;  
    private Position position;  
    private Color color;  
    private Pen pen;  
  
    @Override  
    public void draw() {  
        // código omitido  
    }  
  
    // restante código  
}
```

**Implementa a interface
Drawable**

**Tem que implementar o
método draw da
interface Drawable**

Exemplo — Novo Desenho Casa

■ Classe base **Figure**

```
public abstract class Figure implements Drawable {  
  
    private Pen pen;  
    private Position position;  
    private Color color;  
  
    // public void draw() {  
    // }  
  
    // restante código  
}
```

**Implementa a interface
Drawable**

**Neste caso não é
necessário colocar o
método draw da interface**

- Como o método **draw** não é implementado a classe fica com o método da interface que é abstrato, torna-se abstrata porque não o implementa e deixa para as classes derivadas a sua implementação.

Exemplo — Novo Desenho Casa

■ Nova classe **Drawing**

```
public class Drawing {  
  
    private ArrayList<Drawable> figures;  
  
    public Drawing() {  
        figures = new ArrayList<>();  
    }  
  
    public void addFigure(Drawable figure) {  
        if (figure != null) {  
            figures.add(figure);  
        }  
    }  
  
    public void draw() {  
        for (Drawable figure : figures) {  
            figure.draw();  
        }  
    }  
}
```


**Passa a definir uma
lista de Drawable**

**Usa o polimorfismo com
objetos que pertencem a
classes que implementam
a interface Drawable**


Exemplo — Novo Desenho Casa

■ método **main**

```
public static void main() {  
  
    Canvas canvas = new Canvas("Casa", 500, 300, Color.WHITE);  
    Pen pen = new Pen(0, 0, canvas);  
    Drawing drawing = new Drawing();  
  
    Square wall = new Square(120, new Position(170, 140), pen, Color.RED);  
    Square window = new Square(40, new Position(190, 160), pen, Color.BLACK);  
    Triangle roof = new Triangle(180, 80, new Position(140, 138), pen, Color.GREEN);  
    Circle sun = new Circle(40, new Position(360, 40), pen, Color.YELLOW);  
  
    Person person = new Person(pen);  
    person.setPosition(new Position(340, 210));  
  
    drawing.addFigure(wall);  
    drawing.addFigure(window);  
    drawing.addFigure(roof);  
    drawing.addFigure(sun);  
    drawing.addFigure(person);  
  
    drawing.draw();  
}
```



Criada a person

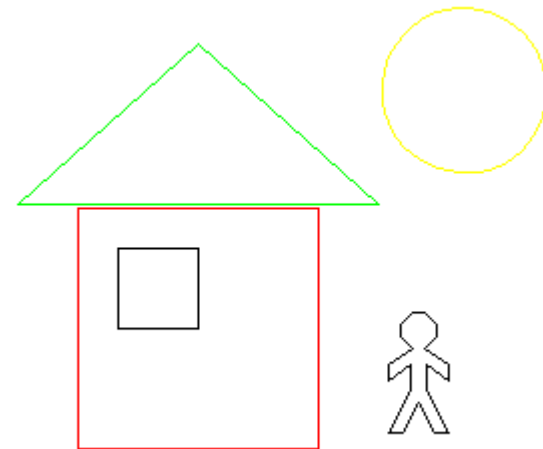


**Adicionada ao
desenho**

Exemplo — Novo Desenho Casa

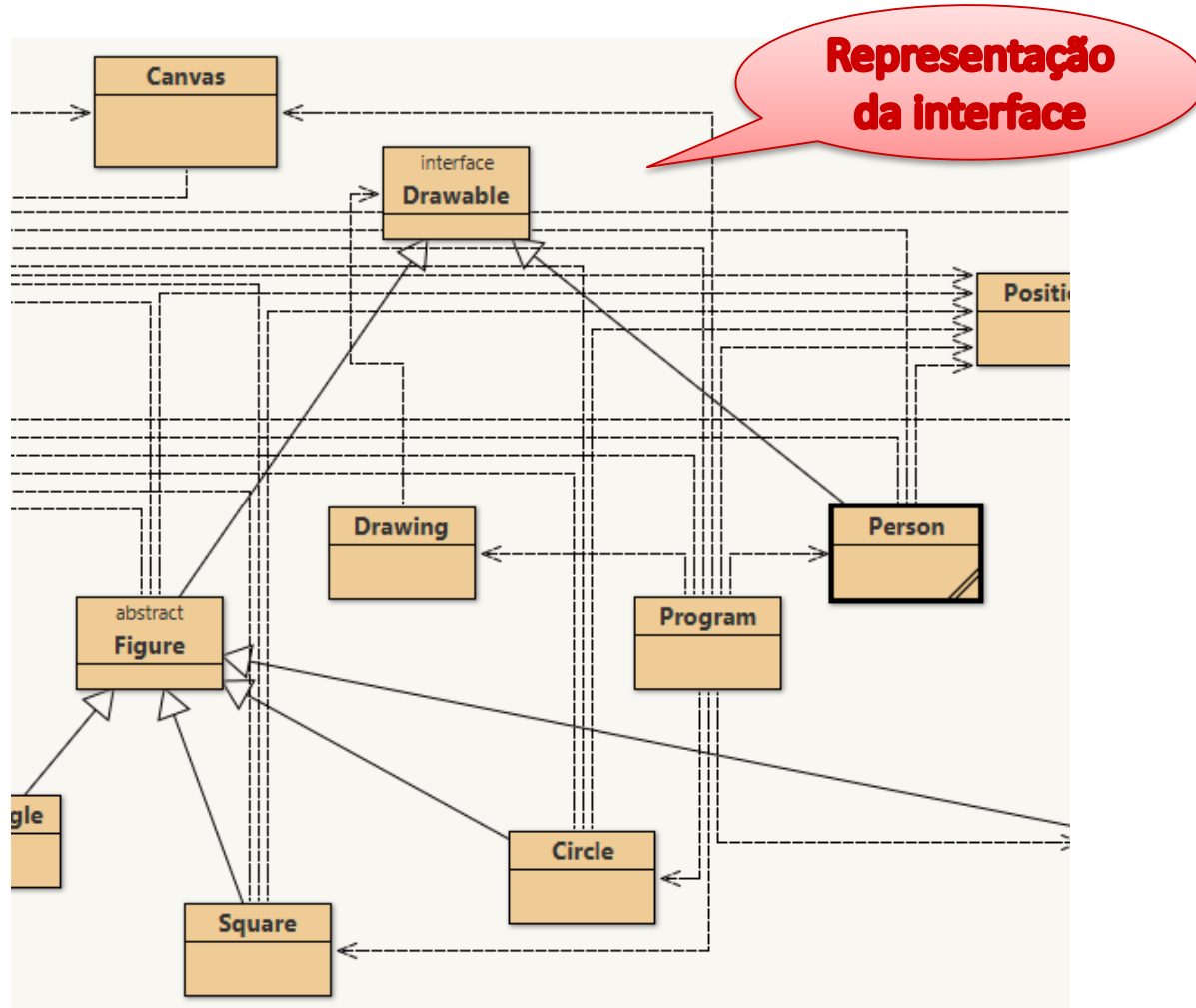
■ método **main**

```
public static void main() {  
  
    Canvas canvas = new Canvas("Casa", 500, 300, Color.WHITE);  
    Pen pen = new Pen(0, 0, canvas);  
    Drawing drawing = new Drawing();  
  
    Square wall = new Square(120, new Position(170, 140), pen, Color.RED);  
    Square window = new Square(40, new Position(190, 160), pen, Color.BLACK);  
    Triangle roof = new Triangle(180, 80, new Position(140, 138), pen, Color.GREEN);  
    Circle sun = new Circle(40, new Position(360, 40), pen, Color.YELLOW);  
  
    Person person = new Person(pen);  
    person.setPosition(new Position(340,210));  
  
    drawing.addFigure(wall);  
    drawing.addFigure(window);  
    drawing.addFigure(roof);  
    drawing.addFigure(sun);  
    drawing.addFigure(person);  
  
    drawing.draw();  
}
```



Exemplo — Novo Desenho Casa

- Diagrama de classes do novo desenho casa



Bibliografia

- Objects First with Java (6th Edition), David Barnes & Michael Kölling, Pearson Education Limited, 2016
 - Capítulo 12

