

Interface

*Adaptado dos materiais de Phyllipe Lima (UNIFEI) por Paulo Meirelles (IME-USP), paulormm@ime.usp.br

Dark Souls

Dark Souls (ダークソウル *Dāku Sōru*[?]) é um **jogo eletrônico** de **RPG de ação** desenvolvido pela **FromSoftware** e publicado pela **Namco Bandai Games**. Lançado originalmente em setembro de 2011 para **PlayStation 3** e **Xbox 360**, é um **sucessor espiritual** de *Demon's Souls* e a segundo título da série *Souls*. *Dark Souls* se passa no reino fictício de Lordran, onde os jogadores assumem o papel de um personagem morto-vivo amaldiçoado que inicia uma peregrinação para descobrir o destino de sua espécie. Um relançamento para **Microsoft Windows** foi realizado em agosto de 2012, com conteúdos adicionais não presentes em suas versões originais. Em outubro de 2012, um novo **conteúdo para download** foi disponibilizado para a versão de consoles, sob o subtítulo *Artorias of the Abyss*.

Dark Souls recebeu aclamação da crítica, com muitos citando-o como um dos **maiores jogos de todos os tempos**. Os críticos elogiaram a profundidade de seu combate e *level design*. No entanto, a dificuldade do jogo recebeu críticas mistas, com alguns criticando-o por ser implacável demais. A versão original do jogo para Windows foi menos bem recebida, com críticas direcionadas a vários problemas técnicos. Em abril de 2013, o jogo havia vendido mais de dois milhões de cópias em todo o mundo. Duas sequências, *Dark Souls II* e *Dark Souls III*, foram lançadas em meados da década de 2010, enquanto uma versão remasterizada, *Dark Souls: Remastered*, foi lançada em 2018.

Índice [esconder]

- 1 Jogabilidade
- 2 Enredo
- 3 Recepção da crítica
 - 3.1 Legado
- 4 Referências

Dark Souls	
	
Desenvolvedora(s)	FromSoftware
Publicadora(s)	Namco Bandai Games ^{JP} FromSoftware
Diretor(es)	Hidetaka Miyazaki
Produtor(es)	Hidetaka Miyazaki, Naotoshi Zin, Yuya

Dark Souls

- Criamos uma superclasse ***abstrata*** chamada Inimigo e outra três subclasses, herdando dela
 - ZumbiLerdo
 - CavaleiroNegro
 - CavaleiroPrata
- Fizemos a classe Inimigo ***abstrata*** pois não fazia sentido uma ***instância*** Inimigo. Porém, fazia todo sentido utilizar o tipo Inimigo como referência, permitindo assim o polimorfismo

Dark Souls

- Depois, fizemos uma classe ***abstrata*** Jogador para representar
 - Mago
 - Guerreiro
 - Arqueiro
- Agora, iremos criar um novo tipo de jogador, Sacerdote

Jogadores que Curam

- Conforme o projeto do jogo avança, novos requisitos surgem
- O projetista do jogo decidiu que alguns tipos de jogadores (isto é, classes que herdam de **Jogador**) podem recuperar vida.
- Ficou definido que apenas **Mago** e **Sacerdote** podem fazer isso
- Como poderíamos implementar esse requisito?

Jogadores que Curam

- Pode parecer adequado inserir um método chamado *recuperarVida()* na classe **Jogador**
- Mas lembre-se que todas as classes que herdaram de **Jogador** também terão esse comportamento *recuperarVida()*
- E os requisitos deixam bem claro que apenas **Mago** e **Sacerdote** podem ter esse comportamento

```
public abstract class Jogador{

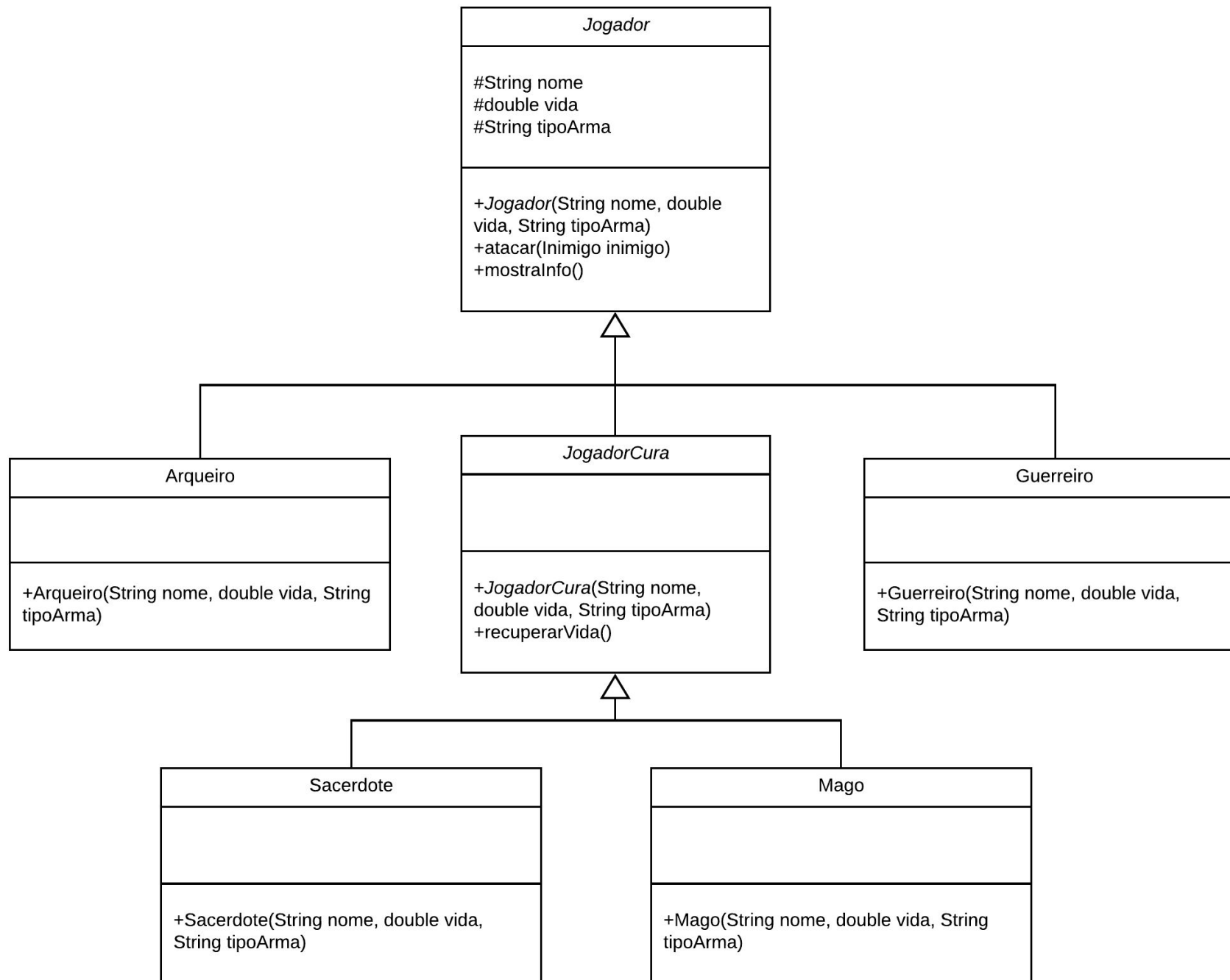
    protected String nome;
    protected double vida;
    protected String tipoArma;

    public Jogador(String nome, double vida, String tipoArma) {
        this.nome = nome;
        this.vida = vida;
        this.tipoArma = tipoArma;
    }

    //Não é boa ideia, pois Guerreiro e Arqueiro
    //Também conseguirão utilizar esse método
    public void recuperarVida() {
        this.vida += 50;
    }
}
```


Jogadores que Curam

- Outra ideia seria criar uma camada intermediária de herança
- Poderíamos criar uma nova classe chamada JogadorCura contendo o método recuperarVida() e apenas as classes Mago e Sacerdote *herdam* de JogadorCura. E JogadorCura, por sua vez, *herda* de Jogador
- Assim, Mago e Sacerdote podem ser referenciados como JogadorCura ou Jogador



Jogadores que Curam

- A princípio esse código continuaria funcionando de forma adequada apenas enquanto criamos novos jogadores
 - Exemplo, imagine que teremos agora um novo tipo chamado Bruxo, e ele também é capaz de recuperar sua vida. Ele pode herdar de JogadorCura e está tudo resolvido. O projeto continua coeso afinal, o Bruxo também **é um** Jogador
- O problema irá surgir se quisermos que algo que **não seja** um Jogador (por exemplo, um Inimigo) possa recuperar sua vida também

Inimigos também Curam

- Vamos fazer CavaleiroDePrata **herdar** de JogadorCura, uma vez que no geral Inimigos e Jogadores estão parecidos
 - Podemos dizer que CavaleiroDePrata **é um** Jogador? Não! Não faz sentido e deixaria nosso programa totalmente confuso
- Mesmo que no código isso **poderia** funcionar, mas, misturando classes que representam “coisas” diferentes, podemos criar um código impossível (ou muito difícil) de evoluir e dar manutenção

Inimigos não são Jogadores

- A classe Jogador possui um método *atacar()* que recebe um Inimigo (não faz sentido o CavaleiroDePrata herdar isso)
- É importante utilizar a herança apenas quando existe a relação “*é um*”
- Caso contrário, estamos ignorando a orientação a objeto e fazendo nossas classes serem apenas depósito de código

Inimigos não são Jogadores

- Ainda não resolvemos o problema de termos Inimigos e Jogadores que podem recuperar a vida
- O que precisamos é criar um “contrato” **do que** algumas classe podem fazer. E toda classe, que desejar, poderia **implementar** esse “contrato”

Criando um “Contrato”

- Estamos falando ***o que fazer*** e não ***como fazer***
- Todo método possui duas características:
 - O que ele faz, isto é, sua assinatura: modificador, tipo de retorno e parâmetros
 - Como ele faz, isto, o código implementado no seu corpo
- Saber o que uma classe ***faz*** nos permite mais generalização, pois é isso que deixamos exposto. ***Como ela faz*** é algo que fica encapsulado dentro do corpo dos métodos

Criando uma Interface

- Em muitas linguagens orientada a objetos podemos criar uma **interface** que define **o que** uma classe deve fazer, **mas não como fazer**
- Vamos criar uma interface chamada “Curavel”. E todo Jogador ou Inimigo que deseje também **ser um** Curavel deve **implementar** essa interface e **obrigatoriamente** dizer **como irá fazer**
- A interface Curavel só diz **o que fazer**. Cada classe que a **implementar** irá decidir **como fazer**

Interface “Curavel”

```
public interface Curavel {  
    public void recuperarVida();  
}
```

- Interface é um ***tipo*** de dado
- Todos os seus métodos são ***públicos*** e ***abstratos*** por padrão
- Não possuem nenhuma implementação, isto é, dizem apenas ***o que fazer*** e não ***como fazer***

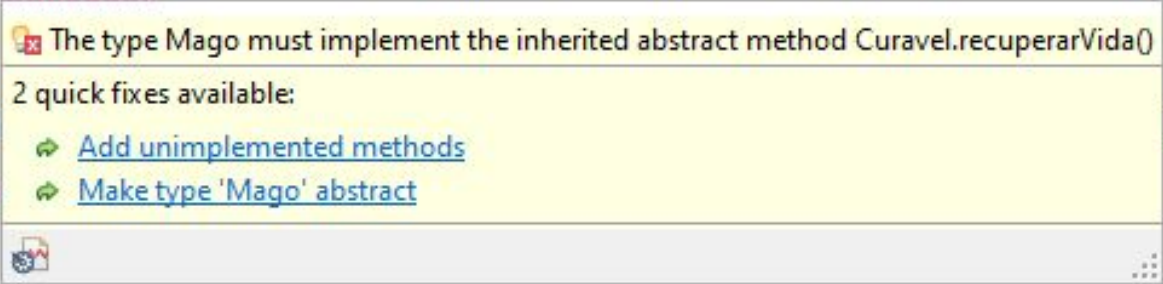
Interface “Curavel”

- Com a nossa interface pronta, podemos fazer qualquer outra classe ***implementá-la*** através da palavra chave ***implements***
- Considere o exemplo para o Mago:

```
public class Mago extends Jogador implements Curavel{  
    public Mago(String nome, double vida, String tipoArma) {  
        super(nome, vida, tipoArma);  
    }  
}
```

Interface “Curavel”

```
public class Mago extends Jogador implements Curavel{  
    public Ma  
    super  
}  
}
```



The type Mago must implement the inherited abstract method Curavel.recuperarVida()
2 quick fixes available:
➔ [Add unimplemented methods](#)
➔ [Make type 'Mago' abstract](#)

- As classes que implementam Curavel, **obrigatoriamente**, precisam implementar todos os métodos. Afinal, por padrão eles são abstratos

Interface “Curavel”

```
public class Mago extends Jogador implements Curavel{

    public Mago(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }

    @Override
    public void recuperarVida() {
        this.vida += 10;
    }
}
```

- Cada classe poderá implementar de acordo com suas especificações

```
public class CavaleiroPrata extends Inimigo implements Curavel {  
  
    public CavaleiroPrata(String nome, double vida, String tipoArma) {  
        super(nome, vida, tipoArma);  
    }  
  
    public void ataqueForte() {  
        System.out.println("Ataque Forte!");  
    }  
  
    @Override  
    public void recuperarVida() {  
        this.vida += 40;  
    }  
}
```

- CavaleiroPrata também é Curavel
- Ele **é um** Inimigo e **é um** Curavel
- Então pode ser referenciado como Curavel e Inimigo?


```
public static void main(String[] args) {  
  
    ZumbiLerdo zumbi = new ZumbiLerdo("Zumbi Lerdo", 50, "Espada Curta");  
    CavaleiroNegro cavNegro =  
        new CavaleiroNegro("Cavaleiro Negro", 150, "Espada Longa");  
    CavaleiroPrata cavPrata =  
        new CavaleiroPrata("Cavaleiro Prata", 175, "Silver Sword");  
  
    //Não compila, pois ZumbiLerdo não é um Curavel  
    Curavel inimigoCuravel = zumbi;  
    //Compila! Pois CavaleiroPrata É UM Curavel  
    Curavel inimigoCuravel2 = cavPrata;  
    //E também É UM Inimigo. Ou seja, pode ser referenciado como ambos!  
    //E claro, como um CavaleiroPrata  
    Inimigo inimigo = cavPrata;  
}
```

Interfaces

- Classes que ***implementam*** Curavel também podem ***ser tratadas*** como Curavel
- Podem existir métodos do nosso programa que recebem variáveis do tipo Curavel, assim como temos métodos que recebem variáveis do tipo Inimigo
- Em outras palavras, com interfaces podemos utilizar o polimorfismo também
- Métodos que recebem variáveis do tipo Curavel podem receber instâncias de Mago, Sacerdote e CavaleiroPrata

Interfaces vs Classe Abstrata

- Ambas não podem ser instanciadas
- Ambas definem novos tipos
- Classes podem implementar mais de uma interface (***implements***), mas podem ***herdar*** apenas de uma superclasse (***extends***)
- Uma classe abstrata que possui apenas métodos abstratos públicos e nenhum atributo, na prática se tornou uma interface
- Como uma interface possui apenas métodos, a mensagem que queremos passar é que estamos criando um novo tipo que só possui comportamento. Por exemplo , Curavel

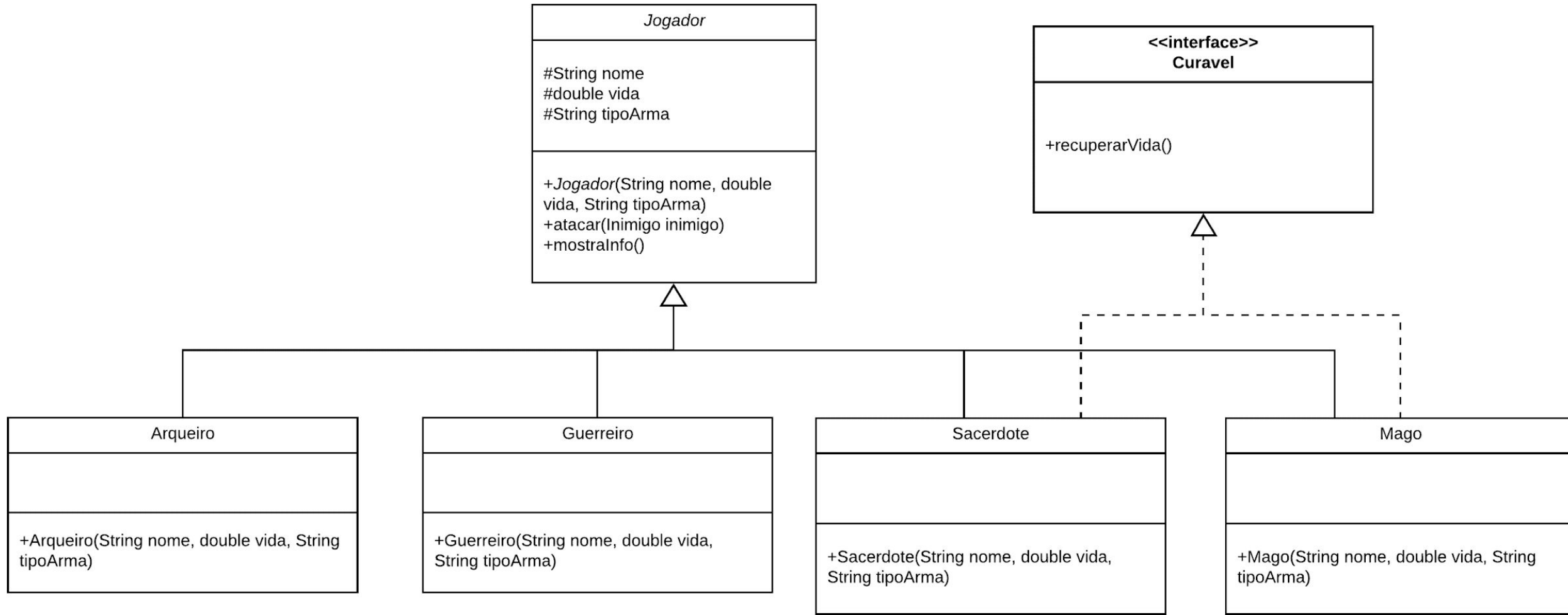
Interfaces vs Classe Abstrata

- A Classe Abstrata pode ter estado também, como Inimigo e Jogador
 - Interface: Estamos abstraindo comportamento
 - Curavel é uma ação que corresponde a recuperar a vida
 - Classe Abstrata: Estamos abstraindo estado e comportamento
 - Jogador: Possui estado (nome, vida) e comportamento (sabe atacar)

UML (Interface)

- No diagrama UML, para representar uma interface, usamos a palavra **<<interface>>** e logo abaixo colocamos o nome, por exemplo, Curavel
- Para indicar implementação usamos a seta branca, mas a **linha tracejada**. A ponta da seta fica na interface, semelhante a herança
- Observe no diagrama UML que a interface não possui área para atributos, reforçando que ela possui apenas comportamento

Exercício 3



Exercício 3

- Termine de implementar a nossa versão do Dark Souls (evoluindo o código dos exercícios 1 e 2)
- Considere que Mago, Sacerdote e CavaleiroPrata irão implementar Curavel
- Escreva os testes de unidade para as novas funcionalidades

Interface

*Adaptado dos materiais de Phyllipe Lima (UNIFEI) por Paulo Meirelles (IME-USP), paulormm@ime.usp.br