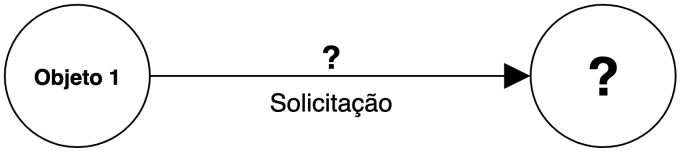
Command

Padrões Comportamentais

O padrão de projeto *Command* encapsula uma solicitação como um objeto, isso lhe permite parametrizar clientes com diferentes solicitações, enfileirar ou registrar *(log)* solicitações e suportar solicitações que podem ser desfeitas.

Motivação (Por que utilizar?)

Considere um cenário onde é necessário emitir diferentes solicitações para diferentes objetos sem conhecer detalhes da operação que está sendo solicitada ou o destinatário da solicitação. E que ainda seja possível que, ao longo do tempo, novas solicitações possam ser suportadas pelo sistema e por isso ele deve ser expansível.



Solicitação desconhecida a um objeto desconhecido

É difícil pensar em um cenário como esse, porém, é nesse cenário que o *Command* se aplica muito bem.

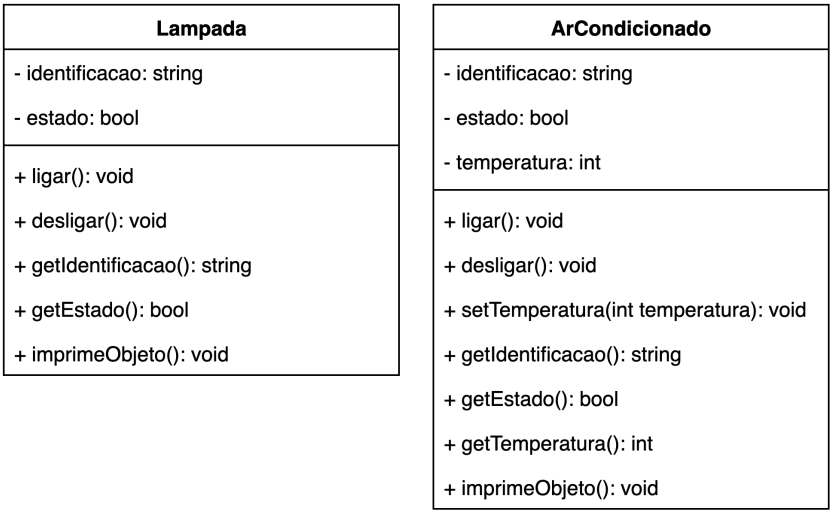
Vamos utilizar um exemplo para ilustrar um desses cenários. Imagine que precisamos implementar um aplicativo para controlar os dispositivos de uma casa pelo celular (*internet of things*).

A casa possui 3 dispositivos inteligentes:

- Lâmpada da Sala;
- Lâmpada do Quarto;
- Ar-condicionado;

Os proprietários da casa já avisaram que desejam adquirir novos produtos inteligentes no futuro, então nosso aplicativo deve ser facilmente expansível.

Cada um dos dispositivos inteligentes aceita um conjunto específico de solicitações, suas classes já foram escritas e fornecidas pelos fabricantes dos dispositivos.



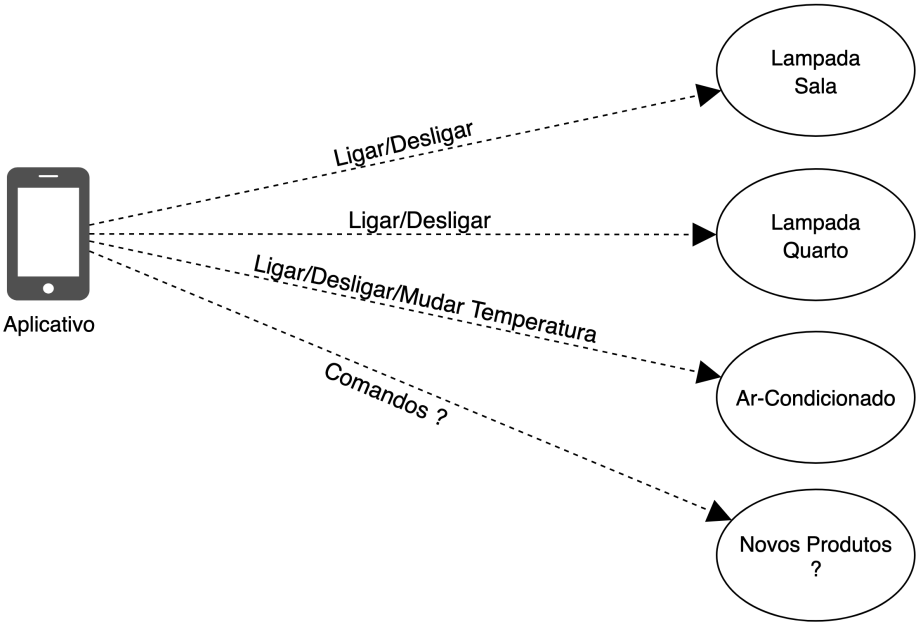
Classes fornecidas pelos fabricantes dos dispositivos

Vejamos os códigos das classes. Lembrando que tais classes são de propriedade dos fabricantes dos dispositivos, por este motivo não podemos modificá-las.

```php
class Lampada
{
    private string $identificacao;
    private bool $estado;

    public function __construct(string $identificacao, bool $estado)
    {
        $this->estado = $estado;
        $this->identificacao = $identificacao;
    }

    public function ligar(): void
    {
        $this->estado = true;
    }

    public function desligar(): void
    {
        $this->estado = false;
    }

    public function getIdentificacao(): string
    {
        return $this->identificacao;
    }

    public function getEstado(): bool
    {
        return $this->estado;
    }

    public function imprimeObjeto()
    {
        $identificao = 'O objeto ' . $this->identificacao . ' está ';
        $identificao .= $this->estado ? "Ligado." : "Desligado.";
        $identificao .= "<br>";
        echo $identificao;
    }
}
```

```php
class ArCondicionado
{
    private string $identificacao;
    private bool $estado;
    private int $temperatura;

    public function __construct(string $identificacao, bool $estado, int $temperatura)
    {
        $this->estado = $estado;
        $this->identificacao = $identificacao;
        $this->temperatura = $temperatura;
    }

    public function ligar()
    {
        $this->estado = true;
    }

    public function desligar()
    {
        $this->estado = false;
    }

    public function setTemperatura(int $temperatura): void
    {
        $this->temperatura = $temperatura;
    }

    public function getIdentificacao(): string
    {
        return $this->identificacao;
    }

    public function getEstado(): bool
    {
        return $this->estado;
    }

    public function getTemperatura(): int
    {
        return $this->temperatura;
    }

    public function imprimeObjeto(): void
    {
        $identificao = 'O objeto ' . $this->identificacao . ' está ';
        $identificao .= $this->estado ? "Ligado" : "Desligado";
        $identificao .= $this->estado ? ' a ' . $this->temperatura . ' graus celsius.' : ".";
        $identificao .= "<br>";
        echo $identificao;
    }
}
```

Uma **Lampada** pode ser ligada e desligada, Um **ArCondicionado** também pode ser ligado e desligado, porém, também pode ter sua temperatura alterada. A casa inteligente deverá ser controlada por um aplicativo para *smartphone* e os comandos podem ser enviados para os respectivos objetos conforme a imagem a seguir.



Possíveis solicitações para possíveis objetos

Nosso objetivo é centralizar o controle de todos os objetos em um único lugar, no caso o `Aplicativo`. Os comandos devem ser enviados para os respectivos objetos somente quando o usuário interagir com o aplicativo, clicando em um botão "Ligar Lâmpada do Quarto" ou "Desligar Ar-condicionado" por exemplo.

Como podemos deixar os comandos pré-configurados no aplicativo de modo que possam ser invocados pelos proprietários quando eles desejarem? E ainda como manter o aplicativo aberto para que receba novos comandos dos novos dispositivos que forem adquiridos pelos proprietários?

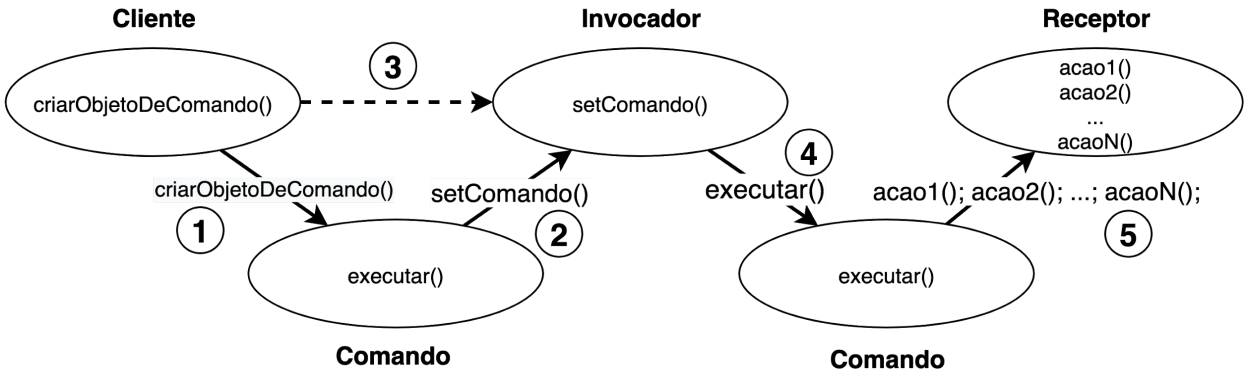
O padrão de projeto *Command* sugere a criação de **objetos de comando.** Tais objetos encapsulam uma solicitação, vinculando um conjunto de ações a serem executadas em um objeto receptor específico.

Um objeto de comando tem as seguintes características:

- Possui um conjunto de ações que devem ser executadas;
- Conhece o receptor que deve executar as ações;
- Expõe um método (`execute()` *ou* `executar()`) que quando chamado, executa as ações no objeto receptor.
- Se necessário, pode expor um método (`undo()`, `desfazer()`) que desfaz as ações do método *executor*.

Externamente nenhum objeto cliente sabe realmente quais ações são executadas em qual receptor. Eles sabem apenas que suas solicitações são atendidas quando o método executor é acionado.

O fluxo de processamento desde a criação de um comando até a execução ocorre da seguinte maneira:



Fluxo de solicitação e execução de um comando

Cliente: Responsável por criar o objeto de comando, que consiste em um conjunto de ações que serão executadas em um receptor.

Comando: Conhece as ações que serão executadas e em qual objeto receptor serão executadas. Fornece um método `executar()`, que encapsula as ações e pode ser chamado para invocar tais ações no objeto receptor.

Invocador: É o objeto intermediário entre um comando e seu receptor. O **Cliente** pede para o **invocador** chamar o método `executar()` do objeto de **Comando**. Possui o método `setComando()` que serve para o **Cliente** definir qual **Comando** o **invocador** deve executar.

Receptor: É o objeto que contém a implementação das ações, que por intermédio do **invocador**, serão chamadas pelo método `executar()` implementado no objeto de **Comando.**

1. O cliente cria um objeto de **Comando;**
2. O **Cliente** chama o método `setComando()` do invocador passando o objeto de **Comando** para ele. O objeto de **Comando** fica armazenado no **Invocador** esperando que seja acionado no futuro.
3. No futuro o **Cliente** pede ao **Invocador** que chame o método `executar()` do **Comando.**
4. O método `executar()` chama os métodos de `Acao()` do **Receptor**
5. O Objeto **Receptor** os executa.

Sabemos que todo comando precisa ter um método **executar()**. Também sabemos que um **Invocador** aceita apenas objetos do tipo **Comando**, então ele precisa ser capaz de verificar isso de alguma forma. No nosso exemplo também iremos implementar o método **desfazer()** de cada objeto de comando.

Vamos criar uma Interface (supertipo) para um comando.

```
interface Command
{
    public function executar(): void;

    public function desfazer(): void;
}
```

Agora vamos identificar quais comandos serão necessário, todos devem implementar a interface **Command**:

Ligar Lâmpada:

```
class LigarLampada implements Command
{
    private Lampada $lampada; //Precisamos saber qual lâmpada será ligada

    public function __construct(Lampada $lampada)
    {
        $this->lampada = $lampada;
    }

    public function executar(): void
    {
        //No método executar, chamar o método ligar() de um objeto Lampada
        $this->lampada->ligar();
        //chamar o método imprimeObjeto() de um objeto Lampada para dar feedback
        $this->lampada->imprimeObjeto();
    }

    public function desfazer(): void
    {
        //No método desfazer, chamar o método ligar() de um objeto Lampada
        $this->lampada->desligar();
        //chamar o método imprimeObjeto() de um objeto Lampada para dar feedback
        $this->lampada->imprimeObjeto();
    }
}
```

Desligar Lâmpada:

```
class DesligarLampada implements Command
{
    private Lampada $lampada; //Precisamos saber qual lâmpada será desligada

    public function __construct(Lampada $lampada)
    {
        $this->lampada = $lampada;
    }

    public function executar(): void
    {
        //No método executar, chamar o método desligar() de um objeto Lampada
        $this->lampada->desligar();
        //chamar o método imprimeObjeto() de um objeto Lampada para dar feedback
        $this->lampada->imprimeObjeto();
    }

    public function desfazer(): void
    {
        //No método desfazer, chamar o método ligar() de um objeto Lampada
        $this->lampada->ligar();
        //chamar o método imprimeObjeto() de um objeto Lampada para dar feedback
        $this->lampada->imprimeObjeto();
    }
}
```

Ligar Ar-condicionado:

```
class LigarArCondicionado implements Command
{
    private ArCondicionado $arCondicionado; //O ar-condicionado que será ligado.

    public function __construct(ArCondicionado $arCondicionado)
    {
        $this->arCondicionado = $arCondicionado;
    }

    public function executar(): void
    {
        //No método executar, chamar o método ligar() de um objeto ArCondicionado
        $this->arCondicionado->ligar();
        //No método executar, chamar o método ligar() de um objeto ArCondicionado
        $this->arCondicionado->imprimeObjeto();
    }

    public function desfazer(): void
    {
        //No método desfazer, chamar o método desligar() de um objeto ArCondicionado
        $this->arCondicionado->desligar();
        //chamar o método imprimeObjeto() de um objeto ArCondicionado para dar feedback
        $this->arCondicionado->imprimeObjeto();
    }
}
```

Desligar Ar-condicionado:

```
class DesligarArCondicionado implements Command
{
    private ArCondicionado $arCondicionado; //O ar-condicionado que será ligado.

    public function __construct(ArCondicionado $arCondicionado)
    {
        $this->arCondicionado = $arCondicionado;
    }

    public function executar(): void
    {
        //No método executar, chamar o método desligar() de um objeto ArCondicionado
        $this->arCondicionado->desligar();
        //No método executar, chamar o método ligar() de um objeto ArCondicionado
        $this->arCondicionado->imprimeObjeto();
    }

    public function desfazer(): void
    {
        //No método desfazer, chamar o método ligar() de um objeto ArCondicionado
        $this->arCondicionado->ligar();
        //chamar o método imprimeObjeto() de um objeto ArCondicionado para dar feedback
        $this->arCondicionado->imprimeObjeto();
    }
}
```

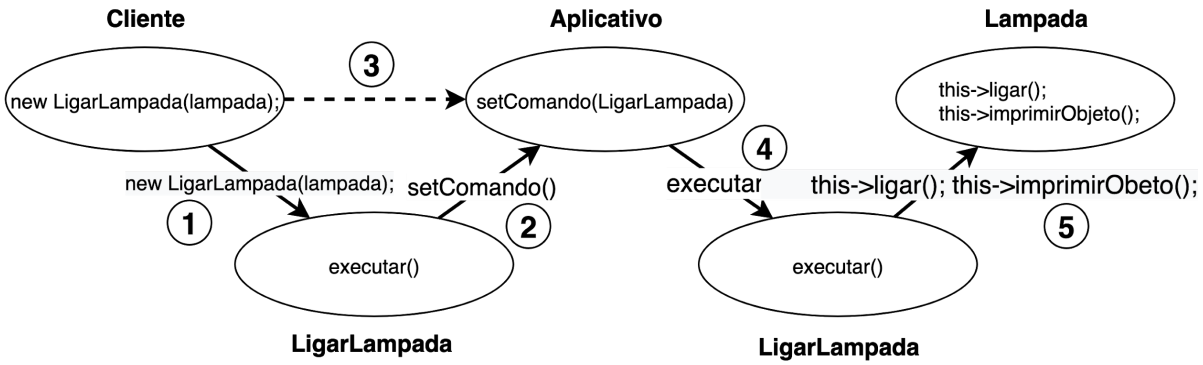
Mudar Temperatura do Ar-condicionado:

```
class MudarTemperaturaArCondicionado implements Command
{
    private ArCondicionado $arCondicionado; //O ar-condicionado que terá temperatura mudada.
    private int $temperatura;
    private int $temperaturaAnterior;

    public function __construct(ArCondicionado $arCondicionado)
    {
        $this->arCondicionado = $arCondicionado;
        $this->temperatura = $arCondicionado->getTemperatura();
    }

    //Define a temperatura do ArCondicionado terá quando o método executar for executado.
    public function setTemperatura(int $temperatura): void
    {
        //Salva a temperatura antiga
        $this->temperaturaAnterior = $this->temperatura;
        //Define a nova temperatura
        $this->temperatura = $temperatura;
    }

    public function executar(): void
    {
        //No método executar, chamar o método setTemperatura() de um objeto ArCondicionado
        $this->arCondicionado->setTemperatura($this->temperatura);
        //chamar o método imprimeObjeto() de um objeto ArCondicionado para dar feedback
        $this->arCondicionado->imprimeObjeto();
    }

    public function desfazer(): void
    {
        //No método executar, chamar o método setTemperatura() de um objeto ArCondicionado
        $this->arCondicionado->setTemperatura($this->temperaturaAnterior);
        //chamar o método imprimeObjeto() de um objeto ArCondicionado para dar feedback
        $this->arCondicionado->imprimeObjeto();
    }
}
```

Temos os comandos devidamente implementados, então vamos implementar nosso **Invocador**, que será o back-end do nosso aplicativo.

```
class Aplicativo
{
    private array $comandos = []; //Array que suporta quantos comandos forem necessários.

    /* Para um comando ser adicionado à lista de comandos do aplicativo ele precisa
    implementar a interface Command, assim sabemos que ele possui o método executar(). */
    public function setComando(Command $comando): int
    {
        $this->comandos[] = $comando; //Adiciona o comando na última posição de $comandos;

        //Retorna para o cliente a posição do array em que o comando foi inserido;
        return count($this->comandos) - 1;
    }

    /* Método que será chamado sempre um botão for pressionado na interface do aplicativo
    O botão precisa passar o indice do comando a ser executado. */
    public function aoPrecionarBotao(int $id): void
    {
        $this->comandos[$id]->executar();
    }

    /* Método que será chamado sempre um botão for pressionado
    duas vezes rapidamente na interface do aplicativo.
    O botão precisa passar o indice do comando a ser desfeito. */
    public function duploCliqueBotao(int $id): void
    {
        $this->comandos[$id]->desfazer();
    }

    // retorna um comando do aplicativo.
    public function getComando(int $id): Command
    {
        return $this->comandos[$id];
    }
}
```

Para exemplificar vamos analisar passo a passo o fluxo de um comando que liga uma lâmpada (para simplificar desconsidere o método **desfazer()**).



Fluxo de solicitação e execução do comando LigarLampada

1. O cliente cria um objeto de **Comando** no caso um objeto **LigarLampada** que recebe uma **Lampada** em seu construtor;

```php
//Primeiro precisamos de um objeto do tipo Lâmpada
$lampada = new Lampada('Lampada', false); //É inicializada desligada.

//Agora podemos criar o comando passando o objeto lâmpada em seu construtor;
$ligarLampada = new LigarLampada($lampada);
```

2. O **Cliente** chama o método **setComando()** do invocador passando o objeto de **Comando** para ele. O objeto de **Comando** fica armazenado no **Invocador** esperando que seja acionado no futuro.

```php
//Precisamos do invocador que é nosso aplicativo
$aplicativo = new Aplicativo();

//Agora podemos armazenamos o comando no aplicativo e guardamos o ID atribuído a ele.
$idLigarLampada = $aplicativo->setComando($ligarLampada);
```

3. No futuro o **Cliente** pede ao **Invocador** que chame o método **executar()** do **Comando**. Nosso **Invocador** é o **aplicativo**.

```php
//O proprietário clicou no botão que deve acender a lâmpada
//Nesse caso chamamos o método aoPrecionarBotao() do aplicativo passando o ID do comando.
$aplicativo->aoPrecionarBotao($idLigarLampada);
```

   O método **aoPrecionarBotao()** chamará o método **executar()** do objeto de comando referente ao ID que passamos por parâmetro, que é o objeto de comando **ligarLampada**;

4. O método **executar()** chama os métodos de **Acao()** do **Receptor**, que são os métodos **ligar()** e **imprimeObjeto()** do Objeto **lampada**.

5. Por fim, o objeto **Receptor lampada** os executa.

Agora que já entendemos como funciona para um comando vamos escrever o código para todos eles.

```
// ----- Vamos criar a Lâmpada da sala -----
$lampadaSala = new Lampada('Lâmpada da Sala', false);

// ----- Agora a Lâmpada do quarto -----
$lampadaQuarto = new Lampada('Lâmpada do Quarto', false);

// ----- O Ar-condidionado -----
$arCondicionado = new ArCondicionado('Ar-Condicionado', false, 26);

// ----- Vamos criar o Aplicativo -----
$aplicativo = new Aplicativo();

// ----- É a vez dos comando serem criados -----
//Comando para a lâmpada da sala
$ligarLampadaSala    = new LigarLampada($lampadaSala);
$desligarLampadaSala = new DesligarLampada($lampadaSala);

//Comando para a lâmpada do quarto
$ligarLampadaQuarto    = new LigarLampada($lampadaQuarto);
$desligarLampadaQuarto = new DesligarLampada($lampadaQuarto);

//Comandos para o Ar-condicionado
$ligarAr            = new LigarArcondicionado($arCondicionado);
$desligarAr         = new DesligarArcondicionado($arCondicionado);
$mudarTemperaturaAr = new MudarTemperaturaArCondicionado($arCondicionado);

// ----- Vamos armazenar os comandos no aplicativo -----

//Os $ids recebem o índice em que o comando foi adicionado ao aplicativo.
$idLigarLampadaSala      = $aplicativo->setComando($ligarLampadaSala);
$idDesligarLampadaSala   = $aplicativo->setComando($desligarLampadaSala);

$idLigarLampadaQuarto    = $aplicativo->setComando($ligarLampadaQuarto);
$idDesligarLampadaQuarto = $aplicativo->setComando($desligarLampadaQuarto);

$idLigarAr               = $aplicativo->setComando($ligarAr);
$idDesligarAr            = $aplicativo->setComando($desligarAr);
$idMudarTemperaturaAr    = $aplicativo->setComando($mudarTemperaturaAr);

// ----- Vamos executar os comandos. -----

$aplicativo->aoPrecionarBotao($idLigarLampadaSala); //Ligar lâmpada da sala
$aplicativo->aoPrecionarBotao($idDesligarLampadaSala); //Desligar lâmpada da sala

$aplicativo->aoPrecionarBotao($idLigarLampadaQuarto); //Ligar lâmpada do quarto
$aplicativo->aoPrecionarBotao($idDesligarLampadaQuarto); //Desligar lâmpada da quarto

$aplicativo->aoPrecionarBotao($idLigarAr); //Ligar Ar-condicionado
//Definir nova temperatura do Ar-condicionado
$aplicativo->getComando($idMudarTemperaturaAr)->setTemperatura(25);
//Mudar temperatura do Ar-condicionado
$aplicativo->aoPrecionarBotao($idMudarTemperaturaAr);

$aplicativo->aoPrecionarBotao($idDesligarAr); //Desligar Ar-condicionado

echo "<br>### Desfazer Comandos ###<br><br>";

$aplicativo->duploCliqueBotao($idDesligarAr); //Defazer o comando ligar ar
$aplicativo->duploCliqueBotao($idMudarTemperaturaAr); //Desfazer a mudança de temperatura
$aplicativo->duploCliqueBotao($idLigarAr); //Desfazer o ligamento do ar
$aplicativo->duploCliqueBotao($idDesligarLampadaQuarto); //Defazer Desligar Lâmpada Quarto
```

Saída

```
O objeto Lâmpada da Sala está Ligado.
O objeto Lâmpada da Sala está Desligado.
O objeto Lâmpada do Quarto está Ligado.
O objeto Lâmpada do Quarto está Desligado.
O objeto Ar-Condicionado está Ligado a 26 graus celsius.
O objeto Ar-Condicionado está Ligado a 25 graus celsius.
O objeto Ar-Condicionado está Desligado.

### Desfazer Comandos ###

O objeto Ar-Condicionado está Ligado a 25 graus celsius.
O objeto Ar-Condicionado está Ligado a 26 graus celsius.
O objeto Ar-Condicionado está Desligado.
O objeto Lâmpada do Quarto está Ligado.
```

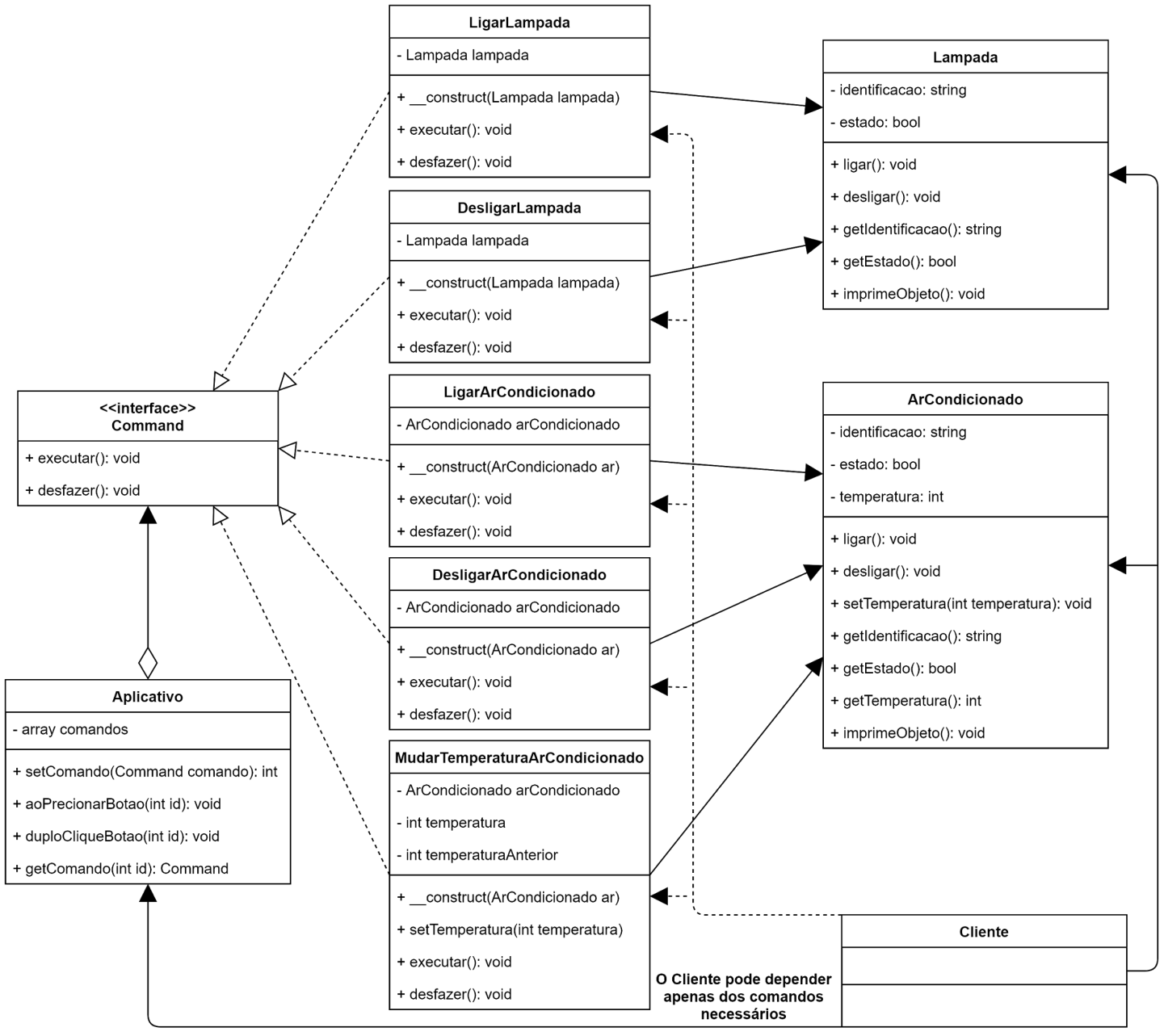


Diagrama de classes completo do exemplo

Aplicabilidade (Quando utilizar?)

- Quando é preciso parametrizar objetos como ação a ser executada:
    - Tal parametrização pode ser expressada numa linguagem procedural através de uma função callback, ou seja, uma função que é registrada em algum lugar para ser chamada em um momento mais adiante.
    - Os Comandos são uma substituição orientada a objetos para callbacks.
- Quando há necessidade de especificar, enfileirar e executar solicitações em tempos diferentes:
    - Um objeto Comando, sabe quais ações executar e conhece o objeto receptor capaz de executá-las. Ele pode executar as ações a qualquer momento, deste modo, pode ter um tempo de vida independente da solicitação original.
    - É possível enfileirar Comandos para serem executados de forma controlada.
- Para suportar desfazer operações:
    - Para desfazer seus efeitos, um comando pode armazenar os estados em que objeto receptor se encontrava antes da a execução das ações (chamada do método *execute()*).
    - A interface *Command* pode conter uma operação *undo()*, que reverte os efeitos de uma chamada anterior de *execute()*. De acordo com a complexidade da reversão ela pode ser baseada nos estados em que o receptor se encontrava anteriormente.
    - Os comandos executados podem ser armazenados em uma lista histórica.
    - O nível ilimitado de desfazer e refazer operações é obtido percorrendo esta lista para trás e para frente, chamando operações *undo*() e *execute*() respectivamente.

- Para tornar possível que mudanças no sistema sejam recuperadas por meio de um registro de mudanças (logging) em caso de queda:
    - Ao aumentar a interface de *Command* com as operações *load()* e *store()*, pode-se manter um registro (log) persistente das mudanças feitas no objeto receptor.
    - A recuperação de uma queda de sistema envolve a recarga dos comandos registrados a partir do disco e sua reexecução com a operação *execute()*.
- Se faz necessário estruturar um sistema em torno de operações de alto nível construídas sobre operações primitivas:
    - Tal estrutura é comum em sistemas de informação que suportam transações.
    - Uma transação encapsula um conjunto de mudanças nos dados.
    - O padrão *Command* fornece uma maneira de modelar transações, já que os comandos podem ser desfeitos para o caso algum deles falhar.
    - Os *Commands* têm uma interface comum, permitindo invocar todas as transações da mesma maneira.
    - O padrão também torna mais fácil estender o sistema com novas transações.

Componentes

- **Cliente:** É o responsável pela criação de um ComandoConcreto e pela definição de seu receptor.
- **Invocador:** Contém um comando e em algum momento pede ao comando para entender uma solicitação chamando o seu método *execute()*.
- **Command:** Declara uma interface para todos os comandos. Um comando é invocado através de seu método *execute()*, que pede a um receptor para executar uma ação. Essa interface também pode possuir um método *undo()*, que desfaz a última ação executada.
- **ComandoConcreto:** Implementa a interface *Command* e define um vínculo entre uma ação e um objeto Receptor. Implementa o método execute através da invocação da(s) correspondente(s) operação(ões) no Receptor.

- **Receptor:** O receptor sabe como executar as tarefas necessárias para atender a uma solicitação, qualquer classe pode atuar como um receptor.

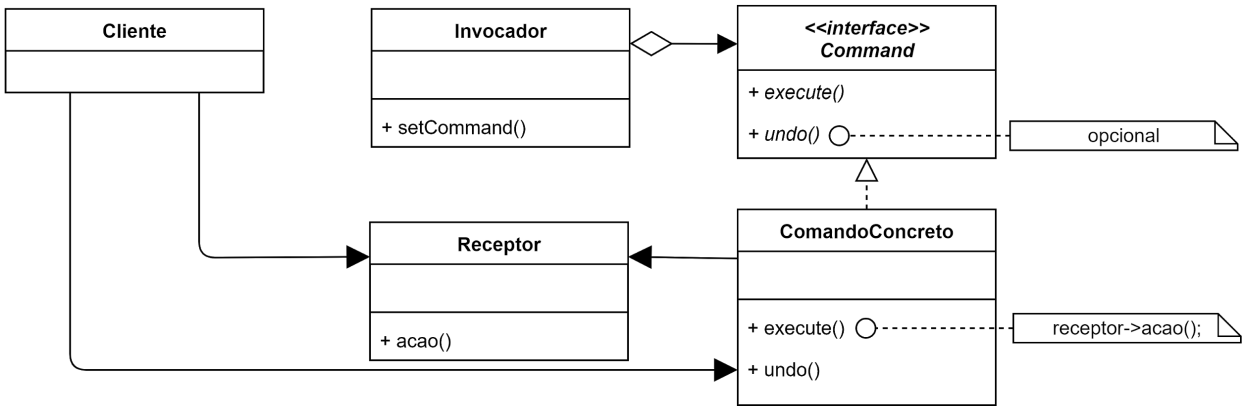


Diagrama de Classes

Consequências

- Comandos são objetos de primeira classe, ou seja, podem ser manipulados e estendidos como qualquer outro objeto.
- Um comando pode ser composto por outros comandos.
- É fácil acrescentar novos comandos porque não é preciso mudar as classes existentes.
- Implementar operações de desfazer fica mais fácil.
- O código pode se tornar mais complexo por existir camadas entre o cliente (solicitante) e o objeto recebedor.