Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

Observer

Padrões Comportamentais

O *Observer* é um padrão de projeto de software que define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente.

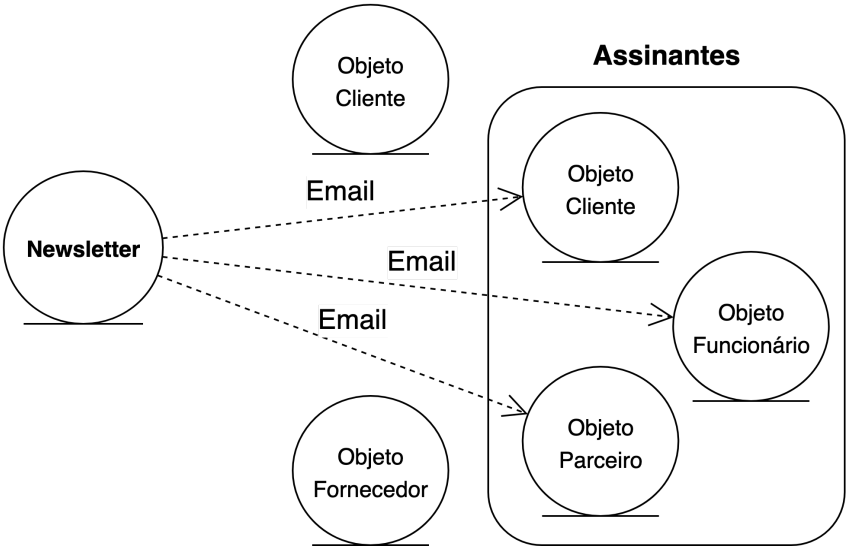
Motivação (Por que utilizar?)

Um sistema pode precisar manter a consistência entre objetos relacionados. Não é recomendado garantir tal consistência tendo como efeito colateral tornar as classes fortemente acopladas, pois isso reduz sua reutilização. Um objeto que se relaciona com outros objetos deve permitir que seus elementos sejam acessados sem que a sua estrutura interna seja exposta.

Para garantir que objetos dependentes entre si possam propagar suas mudanças de estado o padrão *observer* propõe que:

- Os observadores (*observers*) devem conhecer o objeto de interesse.
- O objeto de interesse (*subject*) deve notificar os observadores quando for atualizado.

Os objetos devem interligar-se entre si sem que se conheçam em tempo de compilação. Tomemos como exemplo a implementação de uma *Newsletter* para uma empresa (serviço de assinatura de emails) onde clientes, funcionários, parceiros e fornecedores podem se inscrever para receber emails de notícias sobre a empresa. A *Newsletter* é nosso objeto de interesse, portanto ela é nosso *subject* e os clientes, funcionários, parceiros e fornecedores são os observers.



Esquema de notificações do padrão *Observer*

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

A criação de um novo e-mail é uma mudança no estado de *newsletter*, tal e-mail deve ser enviado a todos os assinantes. No esquema acima temos uma instância de **Cliente** e outra de **Fornecedor** que ainda não são assinantes da *newsletter* e por isso não receberam e-mail, porém, tais objetos podem se tornar assinantes a qualquer momento. Da mesma forma qualquer assinante pode cancelar sua assinatura e deixar de receber emails.

A *newsletter* precisa saber como notificar todos os assinantes a respeito do novo email. Para que isso seja possível eles precisam implementar um método em comum. É possível garantir isso fazendo eles implementarem uma interface em comum.

```
interface Observer
{
    public function update(string $mensagem): void;

    public function getNome(): string;

    public function getEmail(): string;
}
```

Na interface acima apenas o método **update()** é parte do padrão *observer*, os métodos **getNome()** e **getEmail()** fazem parte do contexto do problema.

Para viabilizar nosso exemplo considere a classe Email, que simula o envio de emails aos objetos *Observers*.

```
class Email
{
    public static function enviarEmail(Observer $observer, string $mensagem)
    {
        echo '--------------------------------------------------------------------------------<br>';
        echo 'Email enviado para '.$observer->getNome().' - '.$observer->getEmail().'<br>';
        echo 'Mensagem: ' . $mensagem . '<br><br>';
    }
}
```

Ela recebe como parâmetro um objeto Observer e a mensagem que será enviada. Possui apenas uma método estático que imprime a seguinte saída:

```
--------------------------------------------------------------------------------
Email enviado para “NomeObserver” - “EmailObserver”
Mensagem: “Mensagem recebida por parâmentro”
```

Vamos agora a implementação dos *observers*.

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

```php
class Cliente implements Observer
{
    private string $nome;
    private string $email;
    private Subject $subject;

    //O subject é recebido por parâmetro para que o cliente tenha acesso a ele.
    public function __construct(string $nome, string $email, Subject $subject)
    {
        $this->nome = $nome;
        $this->email = $email;
        $this->subject = $subject;
        $this->subject->registerObserver($this);
    }

    public function update(string $mensagem): void //Faz o envio da mensagem para o email.
    {
        Email::enviarEmail($this, $mensagem);
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getEmail(): string
    {
        return $this->email;
    }
}
```

```php
class Funcionario implements Observer
{
    private string $nome;
    private string $email;
    private Subject $subject;

    public function __construct(string $nome, string $email, Subject $subject)
    {
        $this->nome = $nome;
        $this->email = $email;
        $this->subject = $subject;
        $this->subject->registerObserver($this);
        $this->subject->removeObserver($this);
    }

    public function update(string $mensagem): void
    {
        Email::enviarEmail($this, $mensagem);
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getEmail(): string
    {
        return $this->email;
    }
}
```

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

```php
class Parceiro implements Observer
{
    private string $nome;
    private string $email;
    private Subject $subject;

    public function __construct(string $nome, string $email, Subject $subject)
    {
        $this->nome = $nome;
        $this->email = $email;
        $this->subject = $subject;
        $this->subject->registerObserver($this);
    }

    public function update(string $mensagem): void
    {
        Email::enviarEmail($this,$mensagem);
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getEmail(): string
    {
        return $this->email;
    }
}
```

```php
class Fornecedor implements Observer
{
    private string $nome;
    private string $email;
    private Subject $subject;

    public function __construct(string $nome, string $email, Subject $subject)
    {
        $this->nome = $nome;
        $this->email = $email;
        $this->subject = $subject;
        $this->subject->registerObserver($this);
    }

    public function update(string $mensagem): void
    {
        Email::enviarEmail($this,$mensagem);
    }

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getEmail(): string
    {
        return $this->email;
    }
}
```

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

Em nosso exemplo as classes **Cliente**, **Funcionario**, **Parceiro** e **Fornecedor** são muito parecidas, poderiam inclusive ser subclasses de uma classe mais genérica para reduzir a duplicidade de código. Isso é proposital para simplificar o exemplo e focarmos no conceito do padrão *Observer*. Porém tais classes poderiam ser completamente distintas, a única coisa que precisam ter em comum é a implementação da interface **Observer**.

Agora a *newsletter* sabe que pode utilizar o método **update()** para notificar seus *observers*.

Da mesma forma que a *newsletter* precisa de garantias a respeito de seus observadores um observador precisa saber se um objeto é capaz de notificá-lo, ou seja, se o objeto é um *subject*.

Um *subject* deve ser capaz de:

- **Adicionar** *observers* a sua lista de objetos a serem notificados;
- **Remover** *observers* de sua lista de objetos a serem notificados;
- **Notificar** *observers* da sua lista de objetos a serem notificados.

É preciso criar um supertipo (interface) para os objetos observáveis para garantir aos observadores que suas necessidades serão supridas.

```
interface Subject
{
    public function registerObserver(Observer $observer): void;

    public function removeObserver(Observer $observer): void;

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

A *newsletter* é nosso objeto observável, portanto, ela implementa a interface **Subject** que será esperada pelos *Observers*.

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

```php
class Newsletter implements Subject
{
    private array $observers;
    private array $mensagens;

    /*Adiciona um objeto a lista de observers a serem notificados
      quando uma nova mensagem for inserida na newsletter. */
    public function registerObserver(Observer $observer): void
    {
        $this->observers[] = $observer;
    }

    /*Remove um objeto a lista de observers a serem notificados
      quando uma nova mensagem for inserida na newsletter. */
    public function removeObserver(Observer $observer): void
    {
        foreach ($this->observers as $key => $observerInArray) {
            if ($observerInArray === $observer) {
                unset($this->observers[$key]);
            }
        }
    }

    //Notifica todos os observer sobre a nova mensagem da newsletter.
    public function notifyObservers(): void
    {
        foreach ($this->observers as $o) {
            $o->update(end($this->mensagens));
        }
    }

    /*Adiciona uma nova mensagem a newsletter e solicita que todos
      os observers sejam notificados */
    public function addMensagen(string $mensagem): void
    {
        $this->mensagens[] = $mensagem;
        $this->notifyObservers();
    }
}
```

Deste modo os *observers* têm garantia de que irão se inscrever em um objeto capaz de notificá-los sempre que existir uma nova mensagem na **Newsletter**.

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

Vamos ao teste:

```php
//Criação da Newsletter (Subject);
$newsletter = new Newsletter();

//Criação de funcionários
$funcionario1 = new Funcionario('Funcionario 1', 'funcionario1@email.com', $newsletter);
$funcionario2 = new Funcionario('Funcionario 2', 'funcionario2@email.com', $newsletter);

//Criação de clientes
$cliente = new Cliente('Cliente 1', 'cliente1@email.com', $newsletter);

//Criação de parceiros
$parceiro = new Parceiro('Parceiro 1', 'parceiro1@email.com', $newsletter);

//Criação de fornecedor
$fornecedor = new Fornecedor('Fornecedor 1', 'fornecedor1@email.com', $newsletter);

//Envio da primeira mensagem
$newsletter->addMensagen('Primeira Mensagem');
echo '######<br><br>'; //Separador para facilitar a leitura da saída.

//Remoção do Funcionário 2 da lista de objetos (observers) a serem notificados.
$newsletter->removeObserver($funcionario2);

//Envio da segunda mensagem (Não irá para o Funcionário 2)
$newsletter->addMensagen('Segunda Mensagem');
echo '######<br><br>'; //Separador para facilitar a leitura da saída.

//O Funcionário 2 é reinserido na lista de observers.
$newsletter->registerObserver($funcionario2);

//Envio da terceira Mensagem
$newsletter->addMensagen('Terceira Mensagem');
```

Saída:

```
--------------------------------------------------------------------------------
Email enviado para Funcionário 1 - funcionario1@email.com
Mensagem: Primeira Mensagem

--------------------------------------------------------------------------------
Email enviado para Funcionário 2 - funcionario2@email.com
Mensagem: Primeira Mensagem

--------------------------------------------------------------------------------
Email enviado para Cliente 1 - cliente1@email.com
Mensagem: Primeira Mensagem

--------------------------------------------------------------------------------
Email enviado para Parceiro 1 - parceiro1@email.com
Mensagem: Primeira Mensagem

--------------------------------------------------------------------------------
Email enviado para Fornecedor 1 - fornecedor1@email.com
Mensagem: Primeira Mensagem

######
```

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

```
--------------------------------------------------------------------------------
Email enviado para Funcionário 1 - funcionario1@email.com
Mensagem: Segunda Mensagem

--------------------------------------------------------------------------------
Email enviado para Cliente 1 - cliente1@email.com
Mensagem: Segunda Mensagem

--------------------------------------------------------------------------------
Email enviado para Parceiro 1 - parceiro1@email.com
Mensagem: Segunda Mensagem

--------------------------------------------------------------------------------
Email enviado para Fornecedor 1 - fornecedor1@email.com
Mensagem: Segunda Mensagem

######

--------------------------------------------------------------------------------
Email enviado para Funcionário 1 - funcionario1@email.com
Mensagem: Terceira Mensagem

--------------------------------------------------------------------------------
Email enviado para Cliente 1 - cliente1@email.com
Mensagem: Terceira Mensagem

--------------------------------------------------------------------------------
Email enviado para Parceiro 1 - parceiro1@email.com
Mensagem: Terceira Mensagem

--------------------------------------------------------------------------------
Email enviado para Fornecedor 1 - fornecedor1@email.com
Mensagem: Terceira Mensagem

--------------------------------------------------------------------------------
Email enviado para Funcionário 2 - funcionario2@email.com
Mensagem: Terceira Mensagem
```

Devido a utilização do padrão Observer o código passa a obedecer alguns bons princípios de programação orientada a objetos:

1. **Programe para abstrações:** Newsletter *(subject)* e Cliente, Funcionario, Parceiro e Fornecedor *(Observers)* usam interfaces. O subject monitora os objetos que implementam a interface Observer enquanto os observadores registram e são notificados pela interface Subject.

2. **Mantenha objetos que se relacionam levemente ligados:**

    - A única coisa que o *subject* sabe sobre os *observers* é que eles implementam a interface Observer.

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

- *Subjects* e *Observers* podem ser reutilizados separadamente, um não depende do outro de forma concreta.

3. **Open-closed principle:**

   - Novos observadores podem ser adicionados a qualquer momento sem a necessidade de modificar o *subject.*

   - Alterações no *Subject* e *Observer* não afetarão um ao outro.

4. **De prioridade a composição em relação à herança:** O padrão *observer* utiliza a composição para compor, em tempo de execução, um *subject* com qualquer número de *observers.*

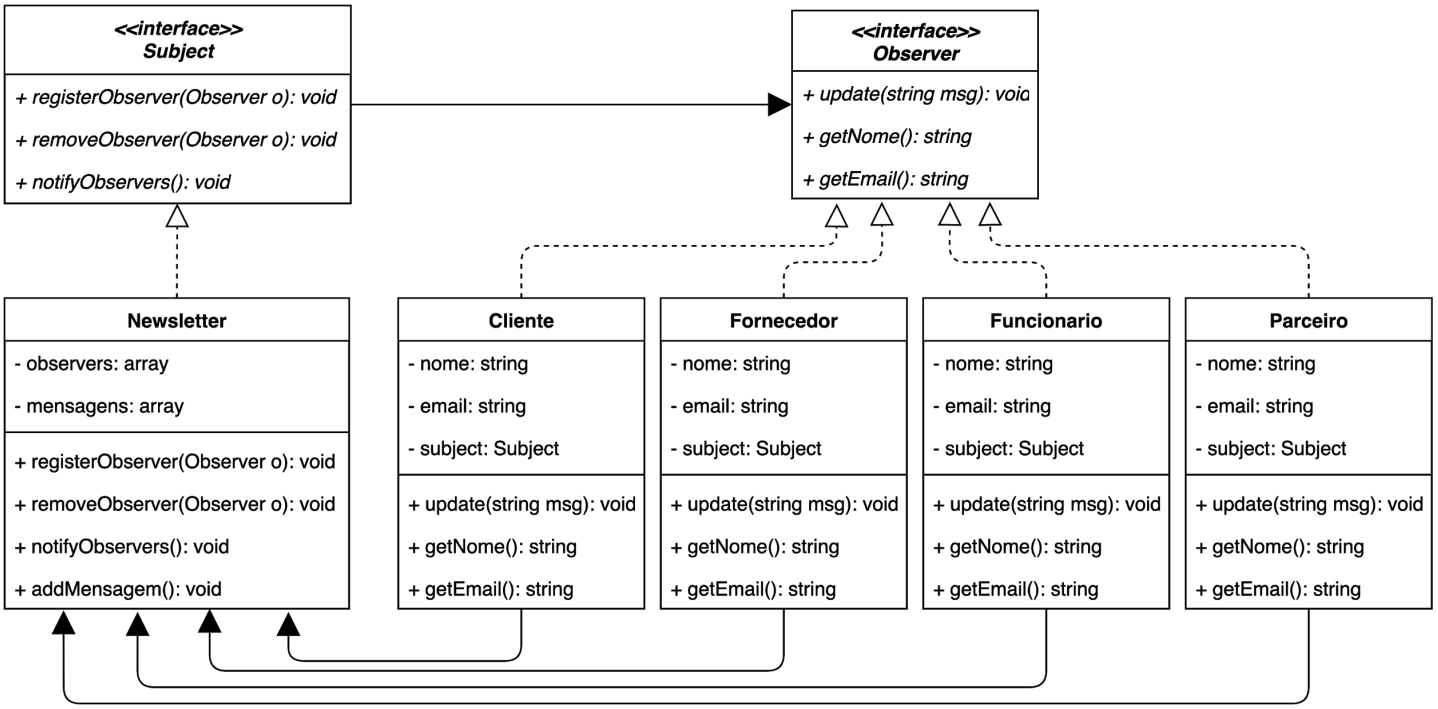


Diagrama de Classes do Exemplo

Aplicabilidade (Quando utilizar?)

- Quando uma abstração tem dois aspectos, um depende do outro, e é necessário que eles possam variar e serem reutilizados independentemente.

- Quando uma alteração em um objeto requer a alteração de outros, e não se conhece quantos objetos precisam ser alterados.

- Quando um objeto deve ser capaz de notificar outros objetos sem os conhecer, ou seja, tais objetos não podem ser fortemente acoplados.

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

Componentes

- **Subject:** Os objetos utilizam esta interface para se registrarem como observadores e para serem removidos.

- **Observer:** Define uma interface de atualização para objetos que devem ser notificados sobre alterações em um Subject.

- **AssuntoConcreto:** Sempre implementa a interface Subject além dos métodos para registrar e remover observers, o AssuntoConcreto implementa o método notifyObservers() que é utilizados para atualizar todos os observadores atuais sempre o que o estado do AssuntoConcreto é alterado. Também pode ter métodos para definir e obter seu estado.

- **ObservadoresConcretos:** Podem ser qualquer classe que implemente a interface Observer. Cada observador se registra a um AssuntoConcreto para receber atualizações. Mantém uma referência a um objeto AssuntoConcreto (que é observado por ele). Tal referência serve para saber de onde vem as notificações e para poder se registrar e se remover.

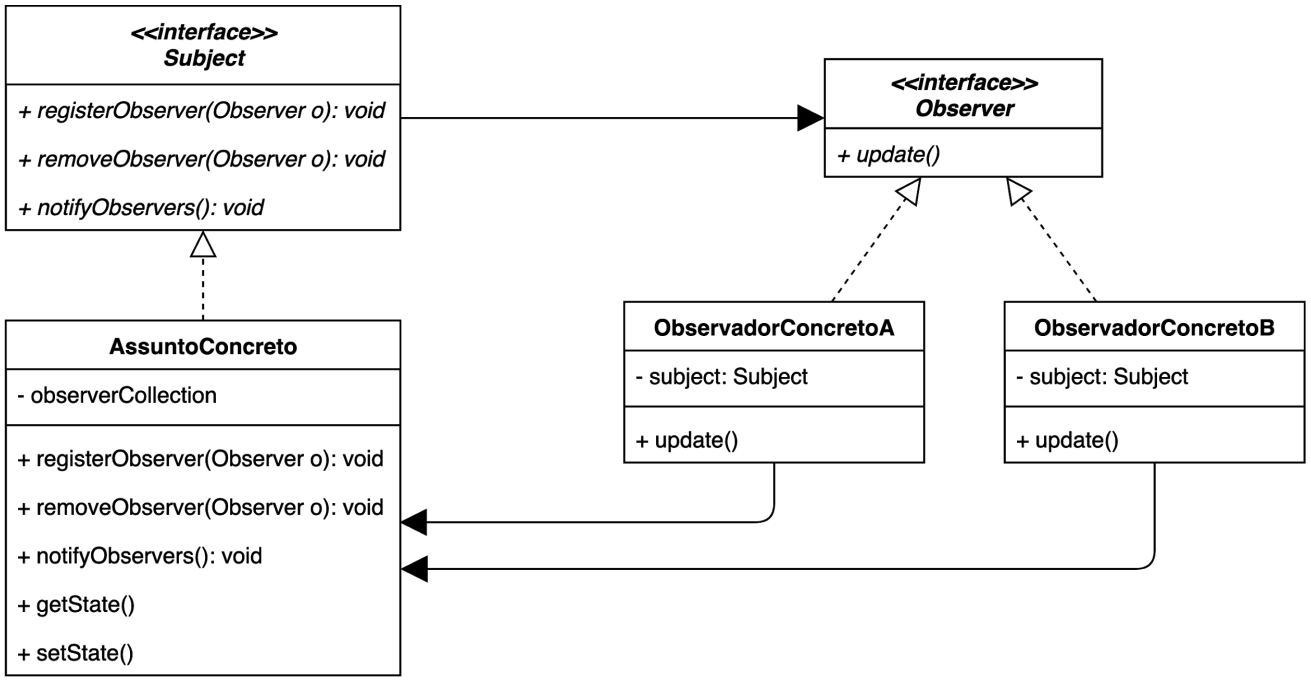


Diagrama de Classes

Licensed to Cleonildo Soares Guimarães Junior - cleu.junior@gmail.com

Consequências

- O padrão *Observer* permite variar assuntos (*subject*) e observadores (*observers*) de forma independente. É possível reutilizar assuntos sem reutilizar seus observadores e vice-versa. Também permite adicionar observadores sem modificar o assunto ou outros observadores.

- Acoplamento abstrato entre Assunto e Observador. Tudo que um assunto sabe é que ele possui uma lista de observadores, cada um em conformidade com a interface *Observer*. O assunto não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre assunto e seus observadores é abstrato e mínimo.

- Suporte para comunicação via *broadcast*. Ao contrário de uma solicitação comum, a notificação que um assunto envia não precisa especificar seu destinatário. A notificação é transmitida automaticamente para todos os objetos observadores que se inscreveram. O assunto não se importa com quantos objetos interessados existem, sua única responsabilidade é notificar seus observadores. Isso lhe dá a liberdade de adicionar e remover observadores a qualquer momento. Cabe ao observador manipular ou ignorar uma notificação.

- Pode causar atualizações inesperadas. Como os observadores não se conhecem, uma operação simples sobre o assunto pode causar uma cascata de atualizações em seus observadores e seus objetos dependentes. Além disso, critérios de dependência que não são bem gerenciados geralmente levam a atualizações desnecessárias, que podem ser difíceis de rastrear.