CES-28 Fundamentos de Eng. de Software

[Objetivos Instrucionais]

O aprendiz é capaz de elaborar diagramas de classe e de sequência.

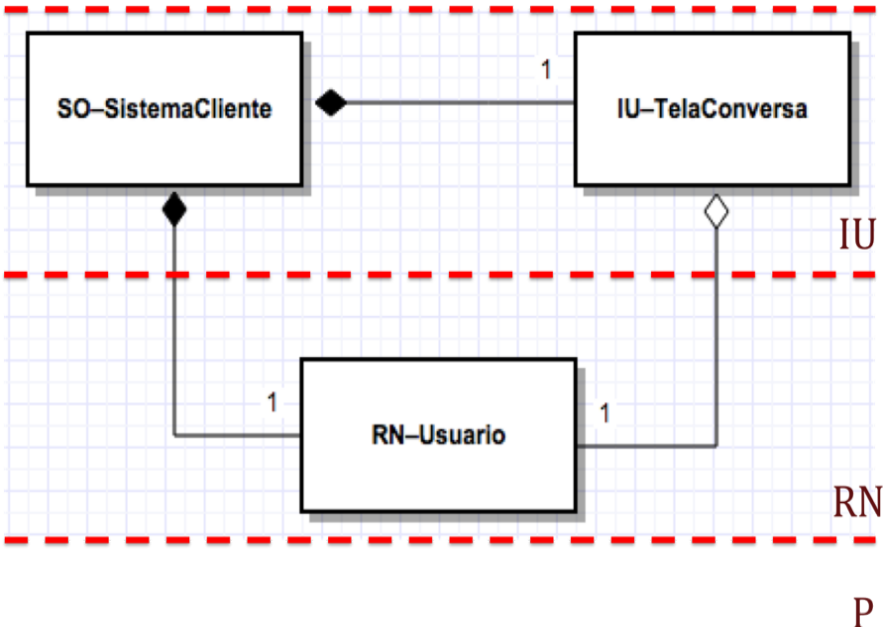
O aprendiz é capaz de aplicar mais um padrão de projeto no contexto abaixo.

[Recursos] Arquivos pdf e sites Internet.

1. **[5 minutos]** Dado o diagrama de classes abaixo e dado que estamos fazendo uso de arquitetura em camadas, considere que a classe IU-TelaConversa está na camada superior IU (Interface com o Usuário) e a classe RN-Usuario na camada imediatamente inferior RN (Regras de Negócio), conforme indicado pelos seus sufixos. Dado que numa arquitetura em camadas um objeto de classe de camada inferior não pode acessar diretamente um objeto de classe da camada superior (imediata ou não), e o objeto da classe RN-Usuario tem informação que precisa ser exibida na tela controlada por objeto da classe IU-TelaConversa, como proceder para cumprir a necessidade sem violar a arquitetura em camadas?

Criar uma classe intermediária para realizar essa conexão.

>>> Discuta sua solução com um colega



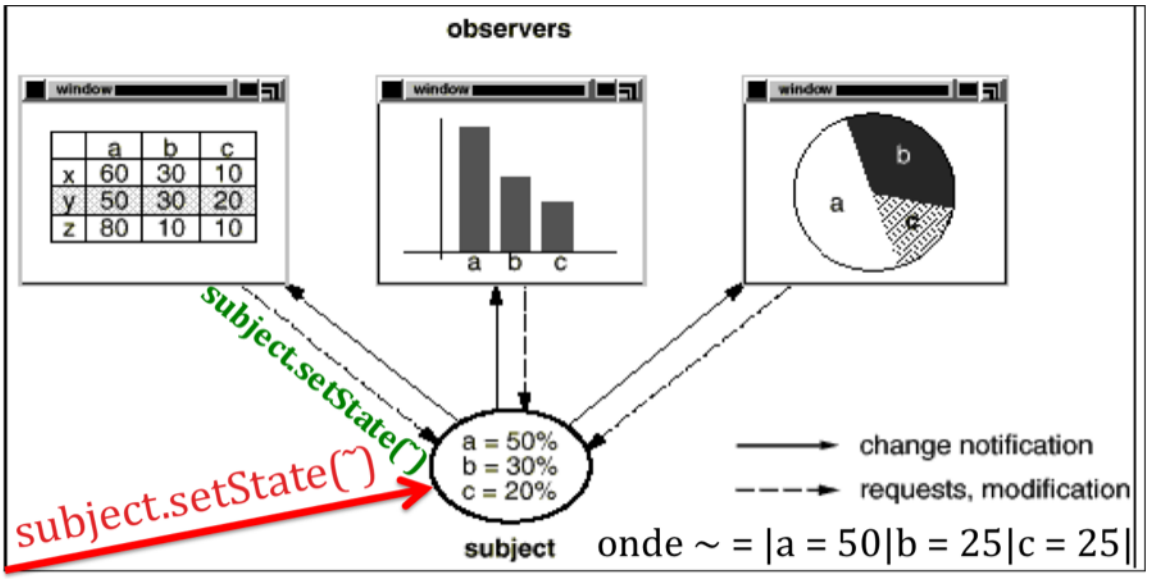
2. **[5 minutos]** Leia a descrição textual abaixo, apenas do padrão Observador. Não veja neste momento o diagrama de classes do padrão!
3. **[5 minutos]** [Papel Explicador] Explique para o seu colega o seu entendimento do padrão do seu jeito, talvez usando a figura abaixo! [Papel de Chato] Você não está entendendo nada da explicação do seu colega e você faz perguntas fruto desse desentendimento! [Vocês decidem quem vai fazer o papel de que na hora da conversa]

Padrões de Projeto

Os padrões de projeto visam a fornecer soluções corretas e elegantes para problemas recorrentes no desenvolvimento de software. Gamma et al. [1995] (GoF – Gang of Four) apresenta uma série de 23 padrões, categorizados como sendo de criação, estrutural e comportamental. Trabalhos subseqüentes apresentaram outros padrões e hoje é possível encontrar na literatura padrões para cada tipo ou situação de desenvolvimento de software.

O padrão *Observer* é um padrão comportamental que cria uma dependência um-para-muitos entre objetos, de modo que, quando o estado de um objeto é modificado, todos os objetos dependentes ou associados são notificados e

atualizados automaticamente. Vide a figura abaixo e o Diagrama de Classes desse padrão que se encontra mais abaixo no documento.



Na figura acima, quando a linha |y|50|30|20| muda para |y|50|25|25| , a janela correspondente informa ao objeto subject que houve tal mudança, fazendo uso do método setState(50, 25, 25). O objeto subject então notifica cada janela observadora da mudança e solicita que elas se atualizem à luz das alterações feitas em subject, inclusive a que tiver acabado de enviar a alteração de valores.

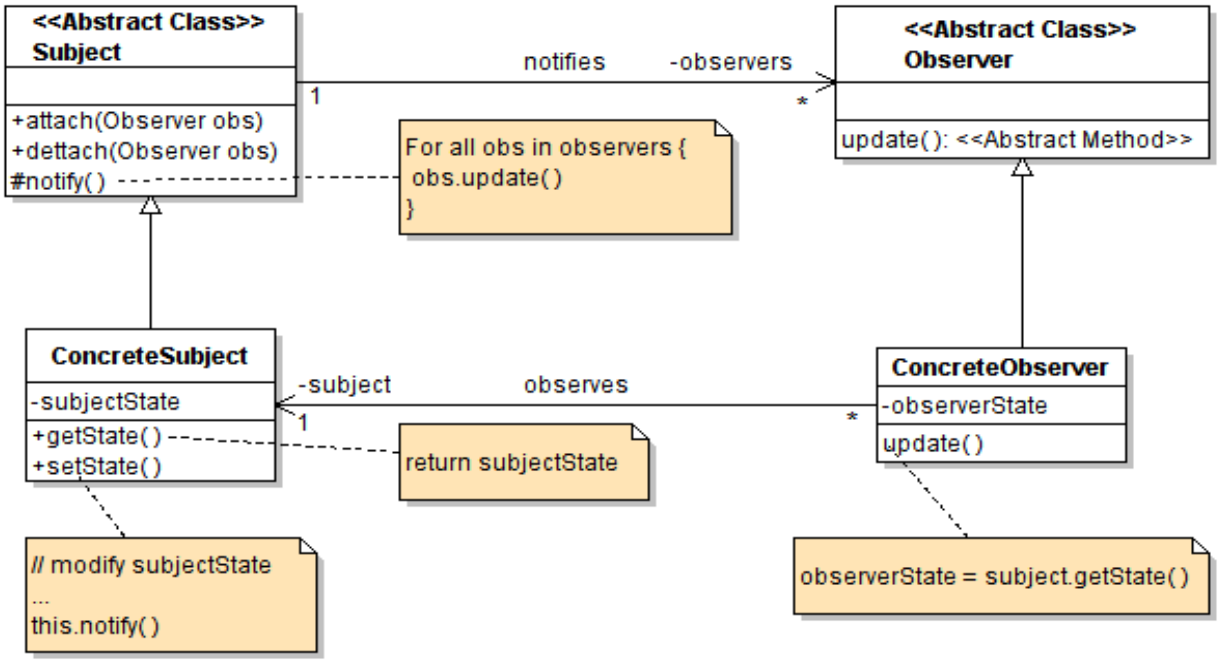
Se o objeto subject tivesse mudado por motivos internos, por exemplo, no caso em que tivesse recebido informação remota alterando alguns dos seus valores através do método setState(), o procedimento é análogo: como o estado do objeto subject foi alterado, notifica cada janela observadora da mudança e solicita que elas se atualizem à luz das alterações feitas nele.

Os objetos-chave nesse padrão são os do tipo Subject e do tipo Observer. Um objeto subject pode ter qualquer número de objetos observadores dependentes. Todos objetos observers são notificados sempre que o subject sofrer uma mudança do seu estado. Em resposta a essa notificação, cada objeto observer irá solicitar ao objeto subject o seu estado atual, para que ele possa atualizar o seu próprio estado.

Isso caracteriza um modelo chamado ***pull***: o objeto subject apenas informa os objetos observers de sua lista de objetos observadores (lista observers) de que o seu estado mudou, deixando para os objetos observadores a tarefa de buscar a informação de mudança do estado no próprio objeto subject.

Com isso, preserva-se a convenção definida para o padrão de arquitetura em camadas, onde a camada superior é quem sempre aciona a inferior. Neste caso, a camada superior é apenas notificada de evento para que ela mesmo tome uma ação direta junto à camada inferior.

A linguagem Java disponibiliza um pacote com as classes necessárias para utilizar, dentre outros, o padrão de projeto *Observer*. Contudo, não estaremos interessados neste momento em fazer uso desse pacote Java.



4. **[5 minutos]** Tente entender o diagrama de classes do padrão Observador da figura acima.
5. **[5 minutos]** [Papel Explicador] Explique para o seu colega o seu entendimento do padrão mostrando o diagrama a ele! [Papel de Chato] Você não está entendendo nada da explicação do seu colega e você faz perguntas fruto desse desentendimento! [Vocês invertem os papéis que tiveram anteriormente]
6. **[5 minutos – em dupla]** Supondo que um objeto aConcreteObserver do diagrama geral do padrão Observer envia uma mensagem setState(~) ao objeto aConcreteSubject, desenhe um diagrama de sequência com as colaborações decorrentes do funcionamento do padrão Observer. Utilize os objetos pertinentes ao caso.
7. **[10 minutos]** Para o diagrama de classes da questão 1 acima e utilizando o padrão *Observer* também acima descrito e ilustrado na figura acima, inclusive com diagrama de sequência, faça a adequação necessária para que a camada IU seja notificada; ou seja, receba e exiba a mensagem numa tela ou campo da interface com o usuário, de acordo com o exposto acima. Será preciso criar novas classes? Que classes do diagrama de classes do sistema cumprirão os papéis do padrão Observer?!?

Sim, é preciso criar novas classes para injetar a interface de comunicação para notificação. No caso, foram criadas duas classes abstratas, uma para ser pai de RN-Usuario e outra para ser pai de UI-TelaConversa. Assim, o RN-Usuario é um subject que notifica UI-TelaConversa quando há mudança de estado para ser impresso na tela e UI-TelaConversa pede para RN-Usuario a informação.

8. **[5 minutos] >>> Discuta sua solução com um colega**
9. **[5 minutos – em dupla]** Como ficaria o diagrama de sequência no caso do exercício 12, sabendo que quem enviou a mensagem setState(~) foi um objeto da classe SO-SistemaCliente?
10. **[10 minutos]** Identifique quais dos padrões de projeto básicos experimentados anteriormente aparecem no diagrama do padrão Observer:

Interface, Accessor Methods, Constant Data Manager, Abstract Parent Class, Null Objects, Hook Methods, Hook Classes, Private Methods, Immutable Object, Private Class Data Pattern, Strategy Pattern e Template Method

Abstract Parent Class

11. **[20 minutos – Zipar o código junto com todas as questões de implementação e depositar na atividade do TIDIA]** Implementar em Java o exposto no diagrama de classes da solução da questão 7 onde o update() consiste apenas em escrever no console. Suponha que seja a classe SO-SistemaCliente que envia a mensagem setState(~) a RN-Usuario. Suponha também que no primeiro evento existem dois objetos observadores e que para o segundo evento um dos objetos observadores é removido. Usar um Id:String para identificar os observadores: os “Obs1” e “Obs2” na saída, que deve ter o seguinte formato:

```
Gerar primeiro evento
---------------------
Obs1 notificado
>>> Olá, como vai?
Obs2 notificado
>>> Olá, como vai?
---------------------

Gerar segundo evento
---------------------
Obs2 notificado
>>> Por aqui vai tudo bem!
```

12. **[30 minutos – Zipar o código junto com todas as questões de implementação e depositar na atividade do TIDIA]** A linguagem Java disponibiliza a classe java.util.Observable e interface java.util.Observer, necessárias para se utilizar, de forma nativa, o padrão de projeto *Observer*.

    a. Redefinir o diagrama de classes da questão 7 para o padrão de projeto Observer considerando a classe Observable e a interface Observer de Java!

b. Implementar o problema usando o ferramental do padrão *Observer* disponibilizado em Java, para dar a mesma saída da questão 11.

http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html

```
public class Observable {
   public void notifyObservers();
   public void addObserver(Observer obs);
   public void deleteObserver(Observer obs);
   ...
}
```

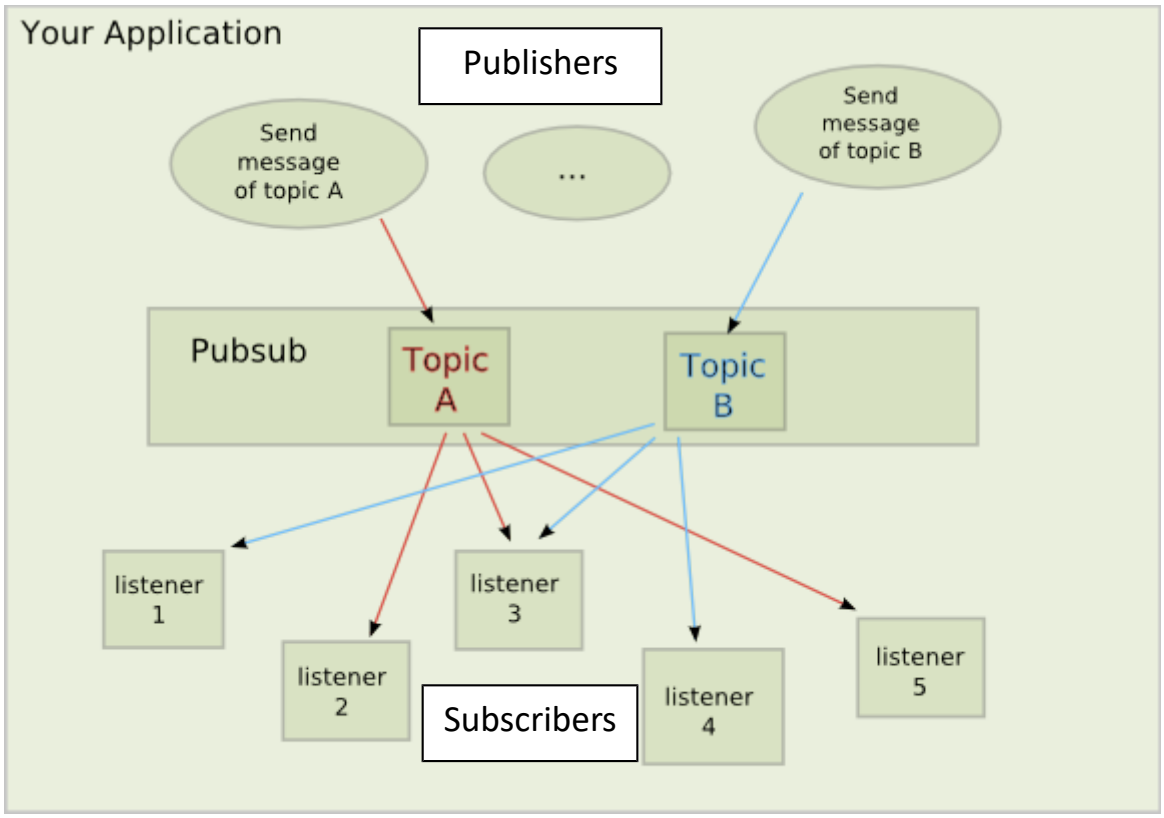
http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html

```
public interface Observer {
   public void update(Observable obs, Object arg);
}
```

13. **[10 minutos]** Apresente o diagrama de classes equivalente ao da questão 7, para o padrão *Observer* usando classe java.util.Observable e interface java.util.Observer.
14. **[5 minutos – em dupla]** Como ficaria o diagrama de sequência no caso do exercício 13, sabendo que quem enviou a mensagem setState(~) foi um objeto da classe SO-SistemaCliente?
15. [5 minutos] Explique o que é o modelo push do padrão Observer. Em que situação é mais apropriado usar? Procure na Internet, se não souber!

O modelo push do Observer já passa direto a informação do evento ao invés de passar o objeto subject para depois o observer usar o getState() para pegar a informação. Se for somente de um tipo o evento, podemos usar o push.

16. O propósito deste exercício é mostrar o padrão Observer aplicado em um contexto diferente e escala maior: como uma parte de um todo maior. Uma semântica de comunicação comum é chamada Publisher/Subscriber. Tipicamente o funcionamento é o seguinte:



Onde as “caixinhas” sender & listener (ou Publisher & Subscriber) podem ser processos ou threads no mesmo computador ou em computadores diferentes, ou podem ser objetos diferentes no mesmo processo e/ou thread (i.e., o contexto é um framework de componentes distribuídos como CORBA, ICE, ou ROS, e tanto faz onde os objetos sender e listeners estão executando - o framework garante que todos se comunicam).

O conceito é que cada Publisher publica mensagens em um tópico específico (ou seja, há um “rótulo” em cada mensagem), e cada Subscriber recebe apenas as mensagens do tópico em que ele está registrado. (Ou seja, há um conceito de “cadastro”, ou inicialização da comunicação, como no método attach() do pattern Observer).

A caixinha pubsub pode ser um processo também (como o EventService de Corba ou o ICEStormService do ICE) que gerencia os tópicos, ou pode não existir nenhum processo intermediário: cada Subscriber gerencia os seus Observers, onde o framework se encarrega de conectar o Subscriber a(os) Publishers(s) que publicam o tópico correto durante o registro.

Suponha um processo SenderGPS, que acessa um objeto para enviar mensagens de dados GPS,:

```
class SenderGPS {

// setup (chamado em constructor ou inicialização)
void setup() {
    // passos p/ inicializar o sistema que escondemos aqui.
    // a próxima linha cria um Publisher
    Publisher pub = AppropriateFrameworkClass.PublisherFactory("GPS");
    // "GPS" é o nome do tópico. As mensagens publicadas por esse
    // Publisher serão marcadas com esse rotulo.
}

void do_work() {
      // loop de envio de mensagens
      while (readGPS() == OK)
            pub.send(getGPSDataAsMessage());
      // supoe que messagem é um tipo simples como string, ou que a
      classe ja gera mensagens no formato apropriado. O que exatamente
      é enviado não é importante agora.
}

}//class
```

.....

Supondo agora outro processo, com uma classe GPSReceiver, que recebe as mensagens.

```
class GPSReceiver {

void setup { //(tipicamente em construtor ou inicialização)
      // inicialização do framework que omitimos aqui.
      // método do framework que cria um subscriber
      Subscriber sub = AppropriateFrameworkClass.SubscriberFactory();
      sub.subscribe("GPS",&GPSReceiver::callback,this);
      // a primeira string indica que apenas as mensagens com esse
identificador serão recebidas.
      // o 2o argumento é um ponteiro de função, indica qual método
será chamado quando uma mensagem é recebida (mas falta indicar um
objeto especifico).
      //o terceiro argumento indica o próprio objeto.
}

void callback (Message msg) {
      // do something nice with the message!!!
}

}//class
```

Temos portanto processos que se comunicam usando classes intermediarias:
class SenderGPS -> class Publisher --> comunicacao escondida entre processos --> class Subscriber -> class Receiver

Que semelhanças e diferenças existem entre este conceito e o padrão observer?

A API mostrada acima corresponde melhor à variação push ou pull? Quem faz o papel de Subject, e de Observer? (Dica: algo análogo a Observer pode aparecer mais de uma vez)
Como as responsabilidades são alocadas entre as classes, e essa alocação é diferente do padrão Observer original? Responsabilidades: notificação de novos eventos; enviar as mensagens; efetuar o registro inicial ligando observer e subject.

Pode ver exemplos reais (a inicializacao omitida aparece) mas similares em:
http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers
https://en.wikipedia.org/wiki/Publish–subscribe_pattern
https://doc.zeroc.com/display/Ice36/Implementing+an+IceStorm+Publisher

Resp.:
Podemos responder pensando no todo ou nas partes.
Pensando nas partes primeiro:
Setup:
GPSSender (Subject), se registra no Publisher (Observer). Isso é diferente, pois no observer original era o Observer que se registrava no Subject.
GPSReceiver (Obsever), se registra no Subscriber (Subject).
Como o Publisher envia as mensagem marcadas com “GPS” para todos os Subscribers que esperam mensagens com o mesmo identificador, por trás há algum tipo de registro (envolve busca do serviço na rede, há mecanismos pra isso), onde o Publisher tem o papel de Subject e o Subscriber o papel de Observer.
Execução:
GPSSender (Subject) chama diretamente o Publisher (Observer), não para apenas notifica-lo, mas para enviar já a mensagem. Isso é o modelo push.
A comunicação entre processos está escondida, mas o Publisher deve enviar a mensagem para o(s) Subscribers corretos. Provavelmente envolve um protocolo mais complexo (middleware). Aguarde um momento para responder.

Uma vez recebida a mensagem, o Subscriber deve invocar um método da classe GPSReceiver. De qualquer forma, não é uma notificação, é já o envio dos dados, portanto corresponde também ao modelo push.
Se é push dos dois lados, faz mais sentido que a comunicação intermediaria Pusblisher-Subscriber também seja conceitualmente push... O Publisher tenta imediatamente enviar os dados a todos os Subscribers.

Pensando no todo (abstraindo que existe uma comunicação intermediária):
Pode-se abstrair como se o Subscriber fosse um Observer e o Publisher um Subject. Quando há um evento no Subscriber, o Observer recebe a mensagem. Como o Subject toma a iniciativa e já envia os dados, segue o modelo push.

17. Implemente uma classe que simule um serviço de Eventos (outro nome para Publisher/Subscriber) dentro de um mesmo processo (não precisa sequer mockar IPC – a mesma classe pode fazer o papel do Publisher e do Subscriber), usando o padrão Observer. Para simplificar, também não precisa identificar as mensagens (todos os subscribers recebem todas as mensagens). Ou seja, podem haver vários subjects que enviam mensagens para a classe, e todas as mensagens são recebidas por todos os observers cadastrados.

O objetivo é implementar um demo onde 2 publishers enviam mensagens e 2 subscribers as recebem:

```
PB = new PublishSubscriber();
sub1 = new Subject(); sub2 = new Subject();
obs1 = new Observer(); obs2 = new Observer();
// faça o registro para PB ser notificado se sub1 ou sub2 mudarem
// faça o registro para obs1 e obs2 serem notificarem se PB mudar
// agora faça um demo de comunicação:
sub1.setValue("msg do sub1");  // PB observa sub1 e recebe a mensagem.
/* nesse momento PB repassa a mensagem para obs1 e obs2, que ao
receber imprimem a mensagem. Aparece na tela:
       obs1: recebi "msg do sub1"
       obs2: recebi "msg do sub1"
*/

// analogamente, agora sub2 é mudado
sub2.setValue("msg do sub2");
/* nesse momento PB repassa a mensagem para obs1 e obs2, que ao
receber imprimem a mensagem. Aparece na tela:
      obs1: recebi "msg do sub2"
      obs2: recebi "msg do sub2"
```

* /

18. teste