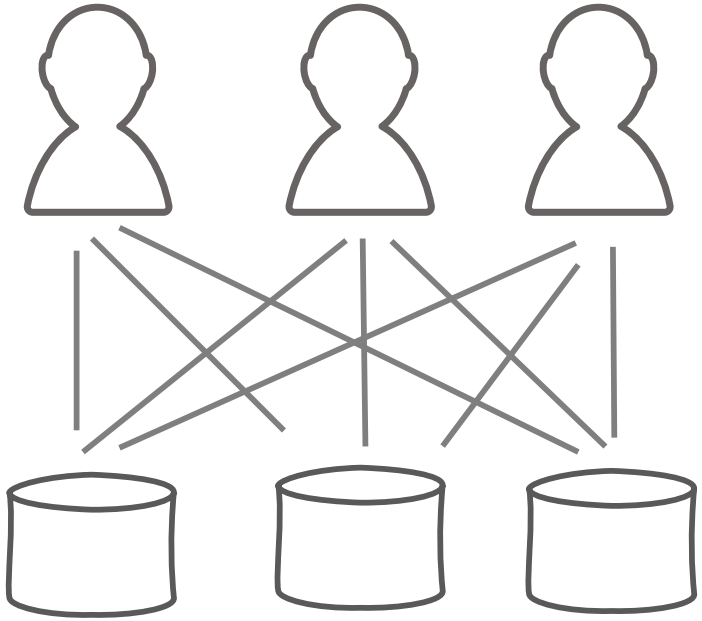
Behavioral Patterns

Mediator

MEDIATOR



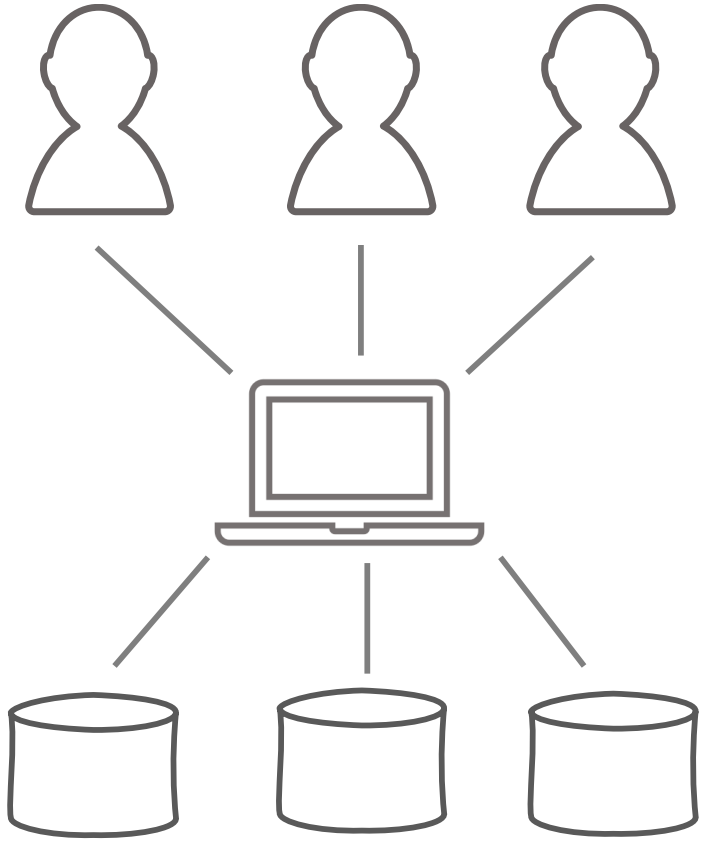
Wzorzec Mediator daje **jednolity interfejs** do przesyłania komunikatów między klasami, **zmniejsza ilość zależności** między nimi, jeśli np potrzebujemy jakąś zależność lub element z innej klasy, wysyłamy polecenie do interfejsu mediatora, a ten zwraca nam odpowiednią wartość.

Zapis zależności w tradycyjny sposób = pajęczyna zależności. Np. dla trzech userów, którzy posiadają rożnego rodzaju uprawnienia do danych to diagram wyglądałby jak na rysunku po lewej.

CEL

Uproszczenie komunikacji wielu obiektów.
Hermetyzacja mechanizmu wymiany komunikatów.

Udostępnianie interfejsu pośredniczącego w przekazywaniu informacji pomiędzy różnymi, niepołączonymi bezpośrednio ze sobą obiektami klas, które dziedziczą po wspólnym interfejsie. Klasa pośrednicząca zawiera **tablicę interfejsów klas**.



ZALETY

- zależności między obiektami są elastyczne, **łatwe w rozbudowie**
- cała **logika jest zahermetyzowana** w klasie mediatora, aby dodać, zależność do jakiejś klasy potrzebujemy jedynie rozszerzyć klasę mediatora
- **uproszczona komunikacja**, komunikacja między klasami sprowadza się do wysyłania polecenia do klasy mediatora

WADY

- istnieje możliwość znacznej rozbudowy klasy Mediatora, co będzie wbrew pierwszej zasadzie **SOLID**, np. jeśli zależności obsługują komunikację to nie dodawać innych zależności
- wyłączna odpowiedzialność obiektu Mediator

case study

CHAT

Użycie **mediatora w chacie**. Jeśli jakaś osoba wysyła wiadomość to jest ona przekazywana do klasy mediatora i następnie przekazywana do odpowiedniej osoby, która miała dostać wiadomość. I wszystkie zależności mamy w jednej klasie.

CONTROL TOWER

Wieża kontroli lotu, nadzoruje lot każdego samolotu w obrębie radaru. Samoloty nie komunikują się ze sobą, nie dogadują się kto gdzie ma lecieć, żeby nie było kolizji.

CODE

```
class iMessage{
   protected:
      std::string name;
      iMediator* mediator;
      bool registerUser();
   public:
      iMessage(std::string name, iMediator*
mediator) : name(name),
mediator(mediator){
          registerUser();
      }
      virtual void getMessage(std::string
from, std::string message) = 0;
   };
```

```
class iMediator{
   public:
      virtual bool registerObject(std::string
name, iMessage* msg) = 0;
      virtual void sendMessage(std::string
to, std::string from, std::string message) =
0;
   };
```

```
class Mediator : public iMediator{
      std::map<std::string, iMessage*> registeredObjects;
   public:
      void sendMessage(std::string to, std::string from, std::string
message){
         std::map<std::string, iMessage*>::iterator iTo=
registeredObjects.find(to);

         if( iTo != registeredObjects.end() ){
            iTo->second->getMessage(from, message);
         }else{
            std::cout << to << " not exist!!!" << std::endl;
         }
      }

      bool registerObject(std::string name, iMessage* msg){
         if(msg){
            std::map<std::string, iMessage*>::iterator iName =
registeredObjects.find(name);
            if(iName == registeredObjects.end()){
               std::cout << name << " is registered" << std::endl;
               registeredObjects[name] = msg;
               return true;
            }
            std::cout << name << " is already registered!!!" <<
std::endl;
            return false;
         }
      }
   };
```

Piotr Kowańdy

Źródła:

1. https://www.obliczeniowo.com.pl/index.php?id=844
2. http://zasoby.open.agh.edu.pl/~09sbfraczek/mediator%2C1%2C45.html
3. http://devman.pl/pl/techniki/wzorce-projektowe-mediatormediator/