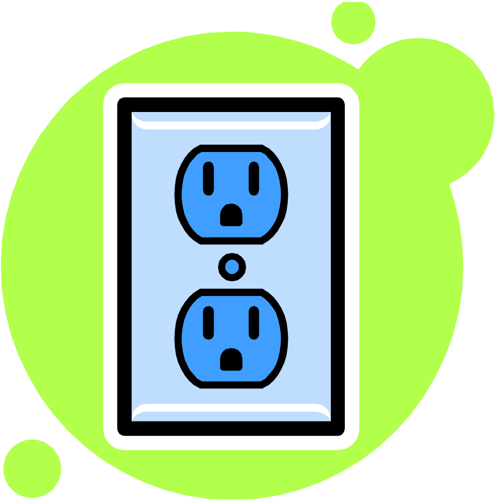
# Façade and Adapter

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

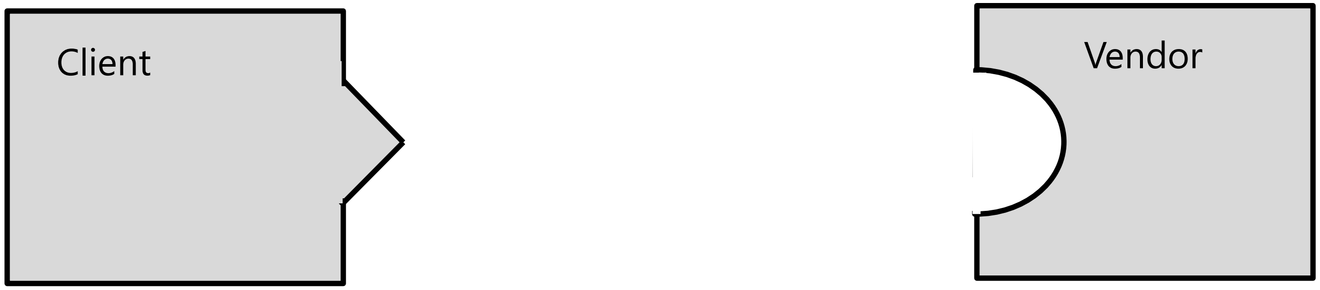
# Adapter Pattern

- Adapters in the real world
  - Power adapter: European -> British -> American
  - Transformers: Scale power up/down
  - PS/2 to USB



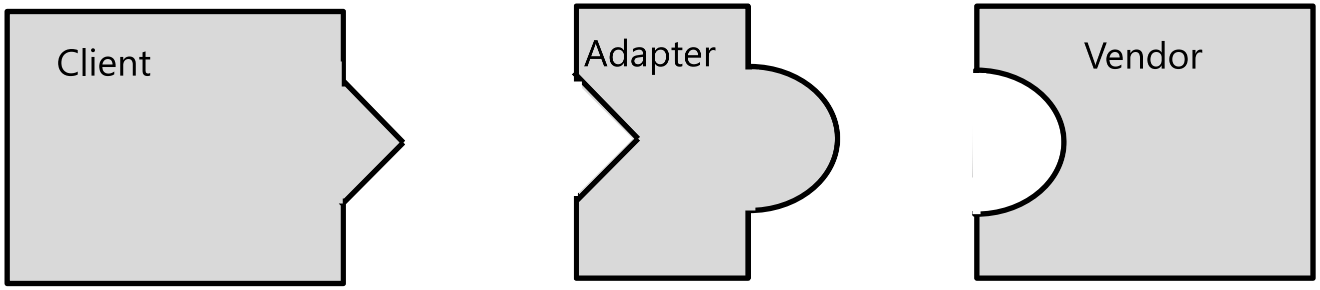
# In Software

- Vendor defines one interface
- Existing application is already coded against another



# In Software

- Vendor defines one interface
- Existing application is already coded against another



# Vendor

- Different Vendors define different interfaces
  - Client coded against one interface
  - Has to adapt to the other

```
interface IXmlDocument
{
    IStream Create();
    IStream Load();
    bool    Save();
}
```

```
interface IPdfDocument
{
    IStream Create();
    IStream Read();
    bool    Write();
}
```

# Adapter

- Adapter implements ‘expected’ interface
  - IXmlDocument in this case
  - Calls adapted interface methods

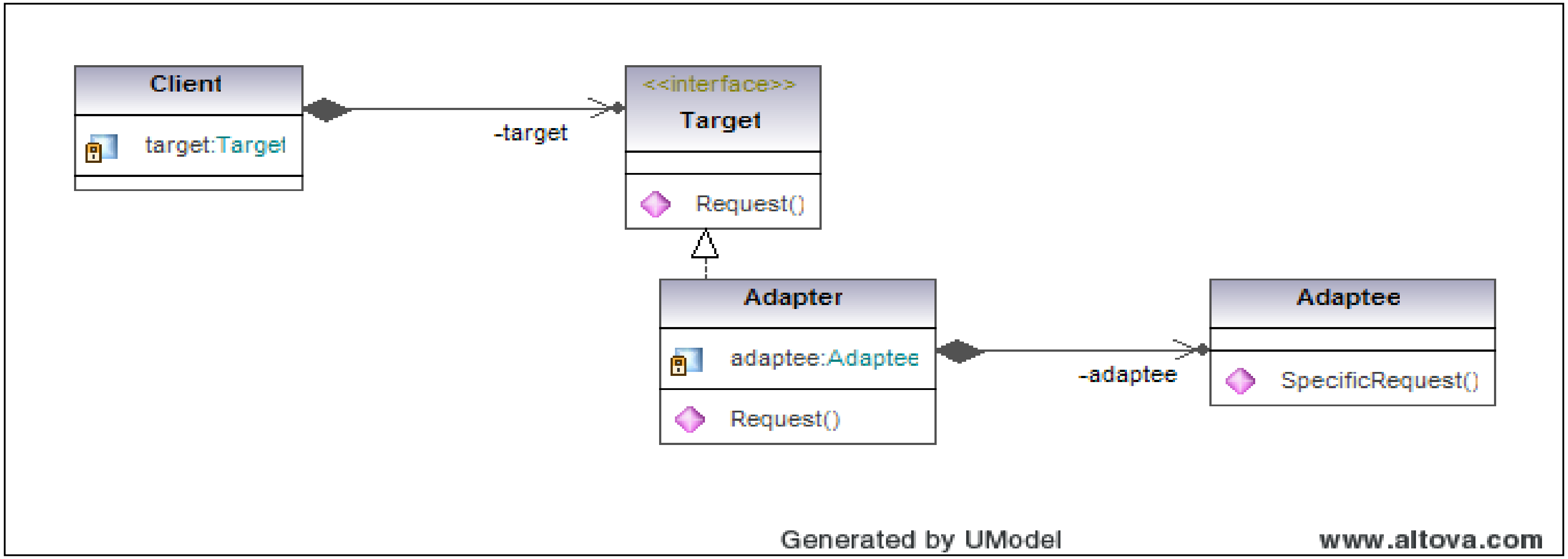
```
class PdfAdapter : IXmlDocument
{
    IPdfDocument pdf;

    public PdfAdapter(IPdfDocument pdf)
    {
        this.pdf = pdf;
    }

    IStream Create(){return pdf.Create();}
    IStream Load()  {return pdf.Read();}
    bool    Save()  {return pdf.Write();}
}
```

# Adapter defined

The Adapter Pattern converts the interface of a class to the interface a client expects

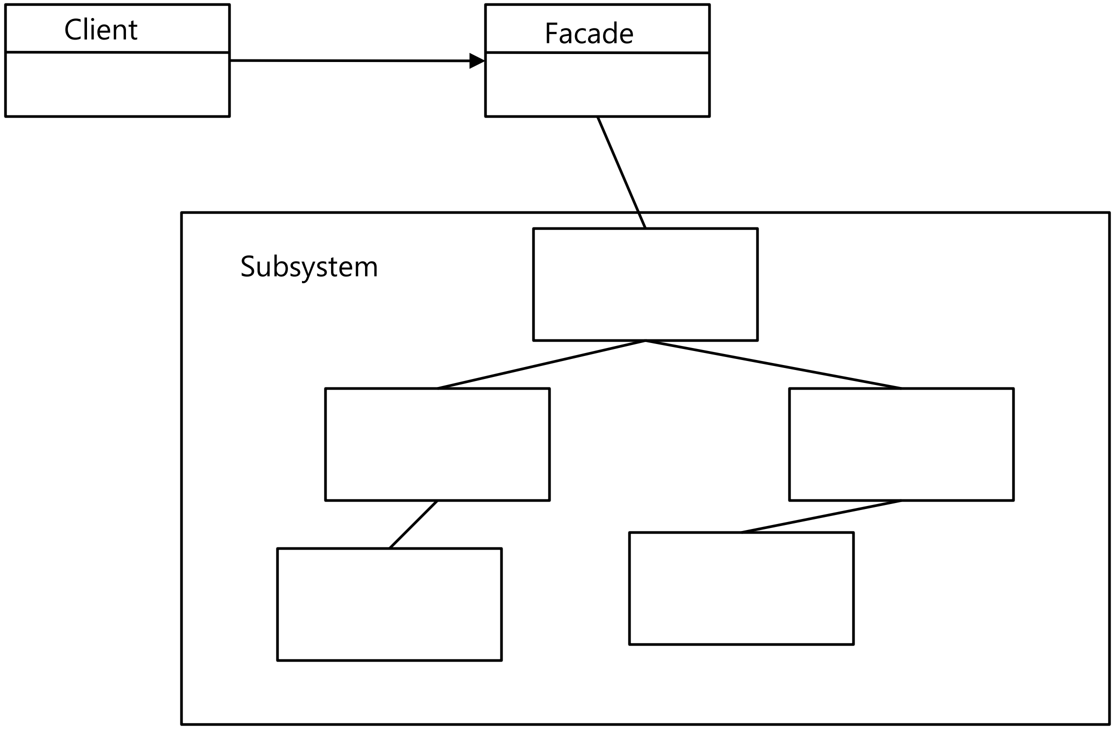


# Facade Pattern

- Defines a high level interface
  - Provides a simplified interface to other systems
  - makes sub-systems easier to use
- May add other functionality
- May have more than one facade to an underlying subsystem

# Facade Pattern

The Facade Pattern provides a unifying interface to a set of interfaces in a subsystem. Facade provides a high level interface that makes a subsystem easier to use.



# Complex Business Process

- Transferring Money
  - Get database connection
  - Get account 1
  - Get account 2
  - Create a transaction
  - Check balance in account 1
  - Withdraw from account 1
  - Deposit in account 2
  - Commit transaction

# Complex Business Process

- Lots of code to transfer money
  - Client does not want to call this every time it has to update accounts

```
class Client {
    public void DoTransfer() {
        IDbConnection conn = GetDataConnection();

        Account account1 = Account.GetAccount(1);
        Account account2 = Account.GetAccount(2);

        using (TransactionScope scope = new TransactionScope()){
            double amount = 100;
            if (account1.Balance >= amount)    {
                account1.Withdraw(conn, amount);
                account2.Deposit(conn, amount);
            }
        }
    }
}
```

# Providing a Facade

- Underlying code still has to be written
  - But is now hidden from the client

```
class Client {
    public void DoTransfer() {
        AccountFacade facade = new AccountFacade();
        facade.TransferMoney(1, 2);
    }
}
```

```
class AccountFacade {
    public void TransferMoney(int accountId1, int accountId2) {
        IDbConnection conn = GetDataConnection();
        Account account1 = Account.GetAccount(accountId1);
        Account account2 = Account.GetAccount(accountId2);

        using (TransactionScope scope = new TransactionScope()) {
            double amount = 100;
            if (account1.Balance >= amount) {
                account1.Withdraw(conn, amount);
                account2.Deposit(conn, amount);
                scope.Complete();
            }
        }
    }
}
```

# Facade Issues

- Access to underlying classes
  - Client can still use underlying classes directly
- Extra functionality
  - Facade can add extra functionality if necessary
- Multiple Facades
  - Multiple facades can be defined for a subsystem
- Decoupled
  - Client is now decoupled from the underlying implementation

# Principle of Least Knowledge

- Also known as the Law of Demeter
- Principle requires that a method of an object may invoke methods on
  - the object itself
  - any parameters passed to a method
  - any object the method creates
  - any components of the object
- In particular, an object should avoid invoking methods of a member object returned by another method

Only talk to your immediate friends

# Example

```
class Customer
{
    Account deposit;

    public void Transfer(Account from){
        SqlConnection connection = new SqlConnection();
        connection.CreateCommand(); // method on object we created

        from.GetBalance();          // method on parameter
        deposit.GetBalance();       // method on component

        UpdateAmount();             // method on class
    }

    private void UpdateAmount() {
    }
}
```

# Examples of the Principle

- Don't do this

without principle

```
public float GetAccountBalance()
{
    Customer customer  = bank.GetCustomer(1234567);
    return customer.GetBalance();
}
```

with principle

```
public float GetAccountBalance()
{
    return bank.GetBalance(1234567);
}
```

# Disadvantages of the Principle

- Will write many wrapper methods
  - so that each wrapper calls a method on its parameters

# Benefits of the Principle

- Software is more adaptable and maintainable
  - fewer couplings
  - fewer dependencies
  - object containers can be changed without affecting
  - easier testing

# Easier Testing

- Fewer dependencies mean fewer objects to create for test
  - Imagine testing the GetAccountBalance method

without principle

```
public float GetAccountBalance()
{
    Customer customer  = bank.GetCustomer(1234567);
    return customer.GetBalance();
}
```

Need to Create test Bank and Customer, objects along with and whatever the customer uses

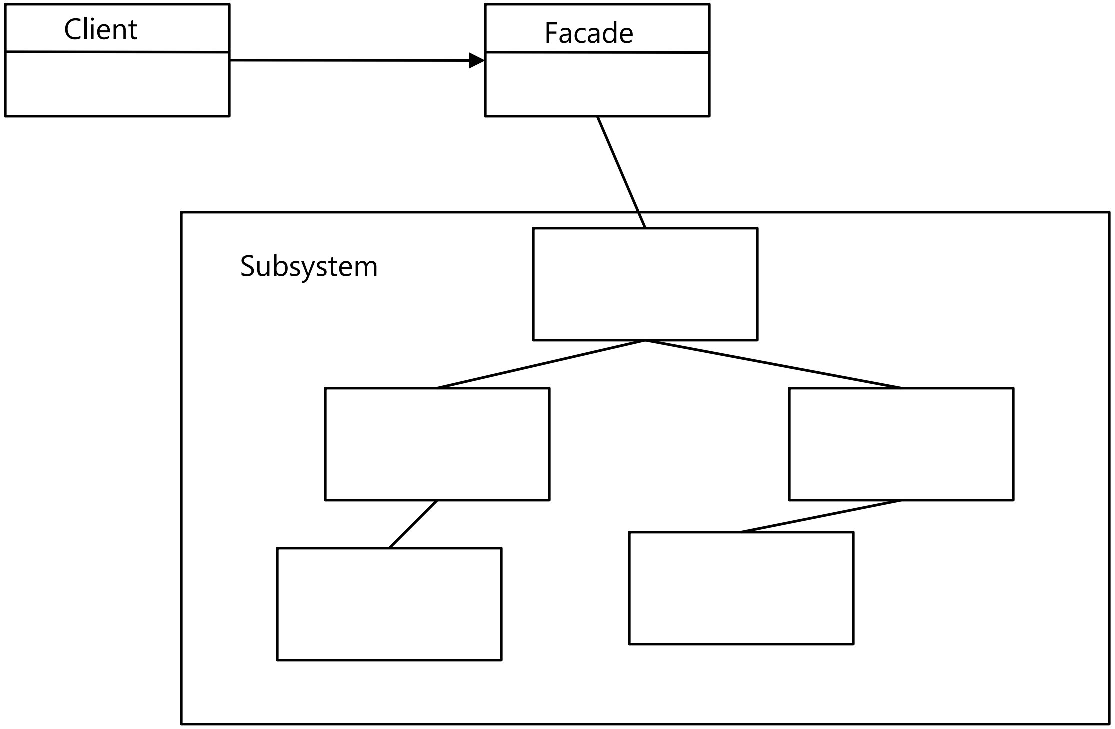
with principle

```
public float GetAccountBalance()
{
    return bank.GetBalance(1234567);
}
```

Only create test Bank object

# Facade and Principle of Least Knowledge

- Client only has one object to interact with
  - Highly decoupled
  - Can change subsystem without affecting client



# Summary

- Use adapters to adapt existing classes to your client
- Use facade to hide complex subsystems from your client
- Remember the Principle of Least Knowledge