Facade [GoF]

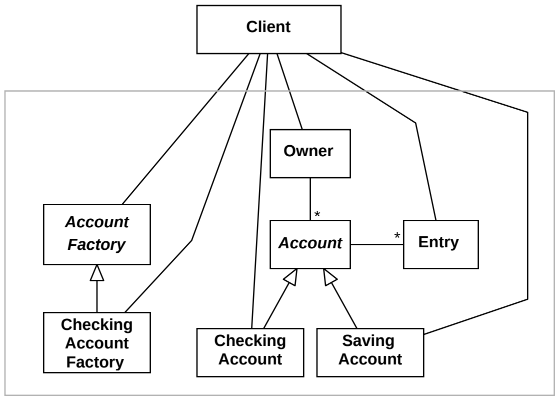
Intent

Substitute the interfaces of a set of classes by the interface of one class. Facade hides implementation classes behind one interface.

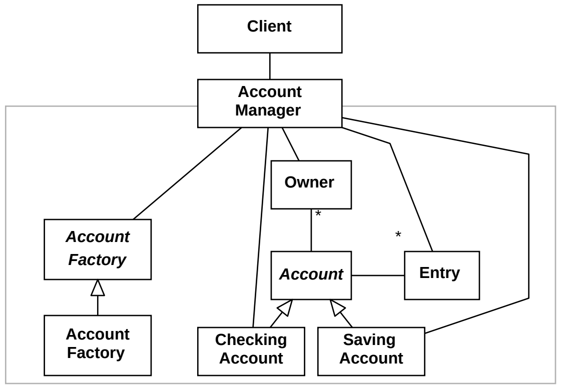
Motivation

Subsystems consist of patterns and classes. Structuring systems with subsystems aims at reducing the dependencies among the parts of a system. Suppose, however, that a subsystem consists of one or more patterns which in turn may involve many classes. In a first approach, a client using this subsystem needs to bind to some (more than one) classes of the subsystem, making the client dependable of many classes of the subsystem.

Consider, for example, a client of a banking application:



Class `Client` directly uses many classes of the banking subsystem, making changes in the subsystem more difficult. To reduce the number of dependencies, introduce a new class with a unified interface for the banking subsystem:



Class `AccountManager` denotes the only class that gives access to the components of the subsystem:

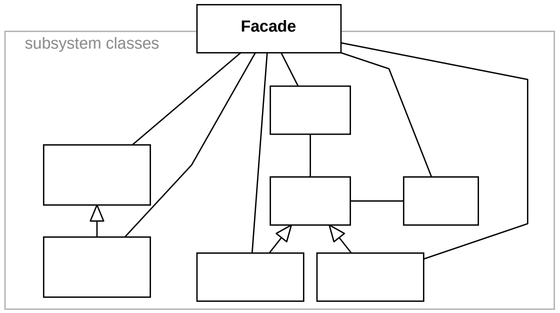
```
public class AccountManager {
   public AccountID createAccount(String kind);
   public void associateAccount(AccountID aid, ClientID cid);
   public void makeBooking(Credentials cred,
                           AccountID debit, AccountID credit,
                           BookingInfo bi);
   public Statement makeStatement(AccountID aid, Date from,
                                  Date to);
   ...
}
```

Applicability

Use a facade when

- it is required to provide a simple interface to a complex system.
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems. Facade classes define entry points to each subsystem layer.

Structure



Participants

- `Facade (AccountManager)`:
  - knows which subsystem classes are responsible for a request
  - dispatches client requests to appropriate subsystem objects

3. Subsystem-structuring with a facade reduces complex dependencies, and can be used to eliminate circular dependencies. Both consequences make subsystems easier to test.
4. (C++) The facade pattern reduces the type dependencies in clients, thus, shortening compilation time and minimizing recompilation in large systems.

Implementation

Consider the following issues:

1. **Subsystem parameterization:** Clients can profit from parameterized `Facade` objects in one of the following ways:
   - An abstract `Facade` class provides the generic interface to the subsystem; concrete `Facade` subclasses provide the actual implementation.
   - A `Facade` object can be configured with different subsystem objects. To customize a `Facade`, simply replace one or more of its subsystem objects.
2. **Parameterized dispatching:** `Facade` objects can dispatch requests from clients to different subsystem objects depending on the current client making the request. Client identification data can be provided implicitly within the request, or explicitly as an additional parameter within the request.

  - optional: performs security-relevant checking prior to the dispatching
- `Sybsystem classes (Account, AccountFactory, Owner, etc.)`:
  - implement the behavior of the subsystem
  - have no knowledge about the `Facade` object

Collaborations

- Clients communicate with subsystems by applying methods on the `Facade` object.
- Clients can provide additional information (i.e., a `Credential` object) for the `Facade` object to validate the request.
- Clients that use the `Facade` object do not have access to its subsystem objects directly.

Consequences

1. The facade pattern shields the client code from the implementation of the subsystem, thereby reducing the number of objects that clients deal with.
2. The facade pattern promotes weak coupling between the subsystem and its clients. In contrast, the objects of a subsystem are strongly coupled, and therefore, difficult to change. Weak coupling lets you vary the subsystem's objects without affecting its clients.

Sample Code

A sketch of our banking application is given next. Lets start with the account factory classes:

```
public class AccountFactory {
   public abstract Account createAccount();
}

public class CheckingAccountFactory extends AccountFactory {
   public Account createAccount() {
      Account acc = new CheckingAccount();
      return acc;
   }
   // Singleton method:
   public static CheckingAccountFactory instance() { ... }

   ...
}
```

Objects of concrete subclasses `Account` implement `Account`'s interface:

```
public abstract class Account {
   public abstract void deposit(Date date, double amount,
                              String text);
   public abstract withdraw(Date date, double amount,
                           String text);
   public abstract double getBalance();
   public abstract Statement performStatement(Date from,
                                              Date to);
}
```

Class `AccountManager` then might look like:

```
public class AccountManager {

   private HashMap accounts;
   private HashMap clients;

   public AccountMangager() { ... }

   // class AccountManager continued...
```

```
   public void makeBooking(Credentials cred,
                           AccountID debit, AccountID credit,
                           BookingInfo bi) {
      Account debitAcc = accounts.get(debit);
      Account creditAcc = accounts.get(credit);
      if ( checkTransaction(cred, debitAcc, creditAcc, bi) ) {
         debitAcc.deposit(new Date(), bi.getAmount(),
                          bi.getText());
      }
   }

   public Statement makeStatement(AccountID aid, Date from,
                                  Date to) {
      Account acc = accounts.get(aid);
      Statement stm = new Statement();
      List list = acc.performStatement(from, to);
      stm.setEntries(list);
      return stm;
   }
}
```

```
   public AccountID createAccount(String kind) {
      Account acc = null;
      if (kind.equals("CHECKING")) {
         acc = CheckingAccountFactory.instance()
                  .createAccount();
         AccountID aid = acc.getAccountID();
         accounts.put(aid, acc);
      }
      return aid;
   }

   public void associateAccount(AccountID aid, ClientID cid) {
      Owner owner = clients.get(cid);
      Account acc = accounts.get(aid);
      if ((owner != null) && (acc != null)) {
         owner.setAccount(acc);
   }
   // class AccountManager continued...
```

Notice that in the above code example, the credentials are given and checked only in one method, method `makeBooking`. In a real example, other methods would need this, too.

Related Patterns

Abstract factory can be used for the construction of the subsystem.

If there is only one instance of a subsystem, then usually only one `Facade` object is required. Thus, facade objects can then be singletons.