SWEN 262

Engineering of Software Subsystems

Similar Dependencies

- You are updating some code that stores customer data in a database.
- The code currently uses a library to connect to a MySQL database.
- Many of your customers have requested that your application also support PostgreSQL, Oracle, and SQL Server
  - Each database provides a library with essentially identical functionality but slightly different APIs.

Q: What is the best way to update the code to support all of the required databases? Is there a way to do it so that other databases are easy to add in the future?

Conditionals

```
public void dbMethod() {
  switch(dbType) {
    case MY_SQL:
      // MySQL-specific code
    case POSTGRESQL:
      // PostgreSQL-specific code
    case ORACLE:
      // Oracle-specific code
    case SQL_SERVER:
      // SQL Server-specific code
   }
}
```

A: Create a single, long method and use a type code to determine which API to use to connect to the database.

Q: What are the drawbacks to this approach?

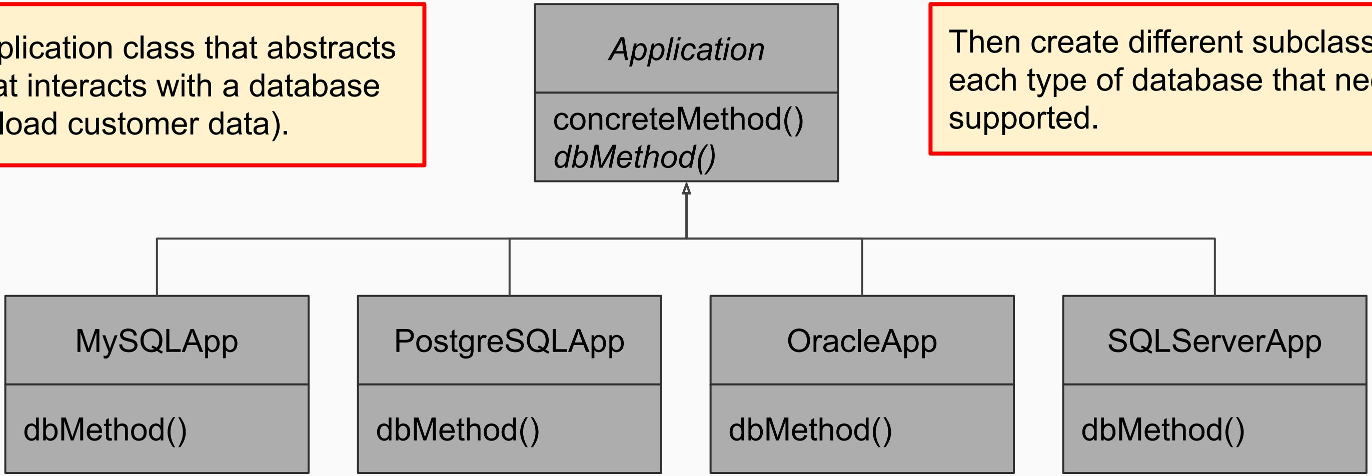
A: Every time a new database is needed, the application has to change (violates OCP).

A: This solution also suffers from a lack of cohesion (tries to know about and do too many things).

A: This also creates a significant amount of coupling between this class, the database APIs, and any class that needs to use the database.

Subclassing

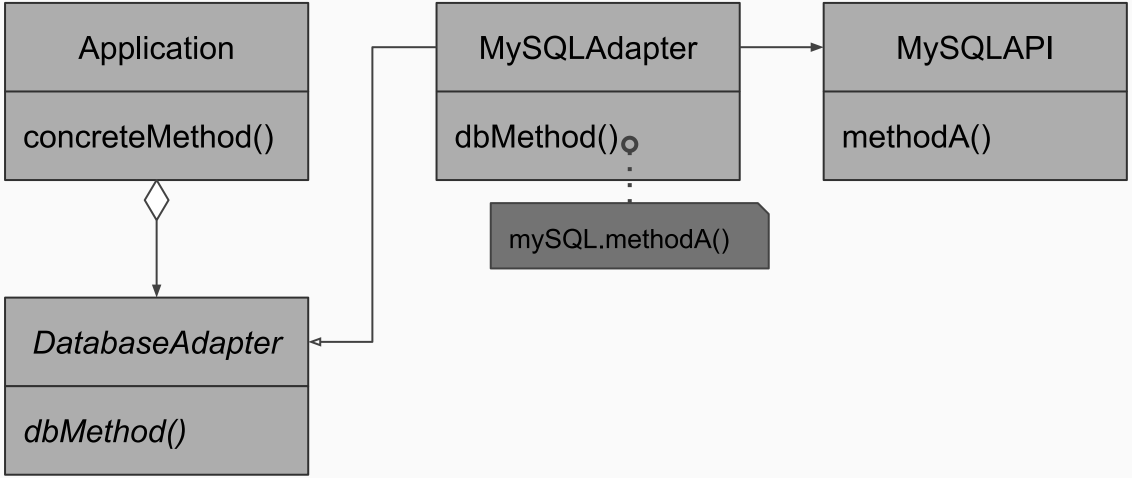
A: Create an application class that abstracts any behavior that interacts with a database (e.g. to store or load customer data).



Then create different subclasses, one for each type of database that needs to be supported.

Q: What are the potential drawbacks to *this* approach?

A Layer of Abstraction



A: Create a *layer of abstraction* in between the application and the database specific API by defining an *adapter* as an interface.

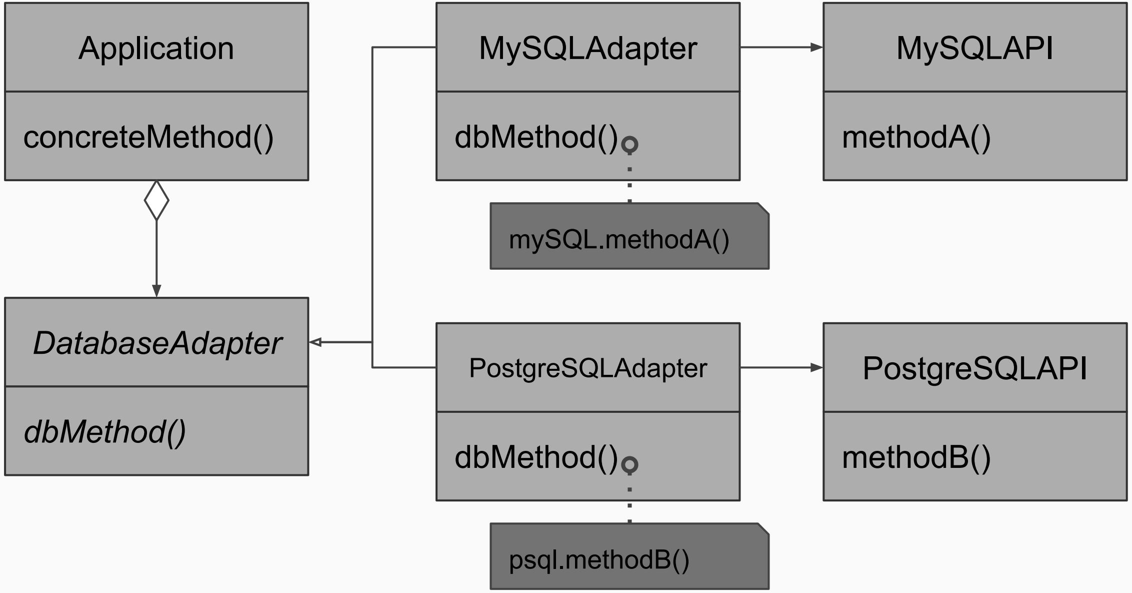
The application aggregates an instance of the *DatabaseAdapter* interface, and uses it whenever it needs to use the database.

An *application specific* implementation of the interface is written for the application to use to connect to MySQL.

Whenever the *dbMethod()* is called on the *MySQLAdapter*, it *adapts* the call to the MySQL API.

All of the platform-specific code is kept within the MySQLAdapter and out of the Application.

A Layer of Abstraction

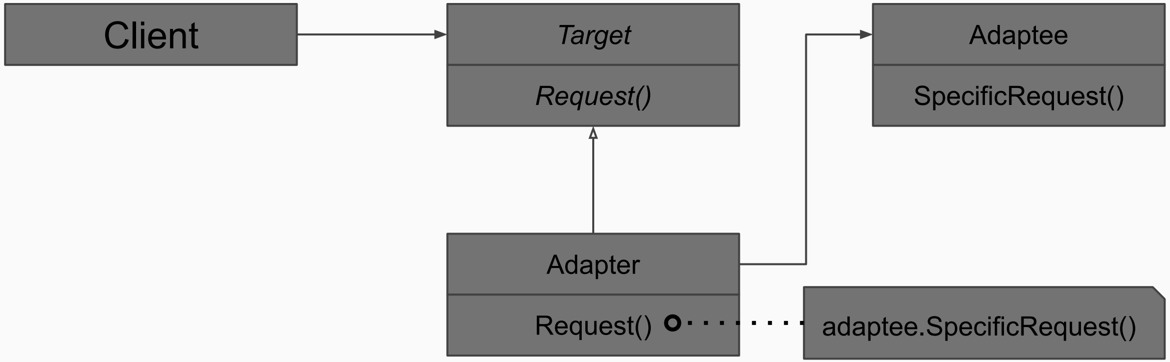


It’s also easy to add support for new databases by creating platform-specific implementations of the *DatabaseAdapter* for each.

The application can be configured to connect to a different database simply by swapping one adapter implementation for another.

No code changes are required.

Adapter (Object)

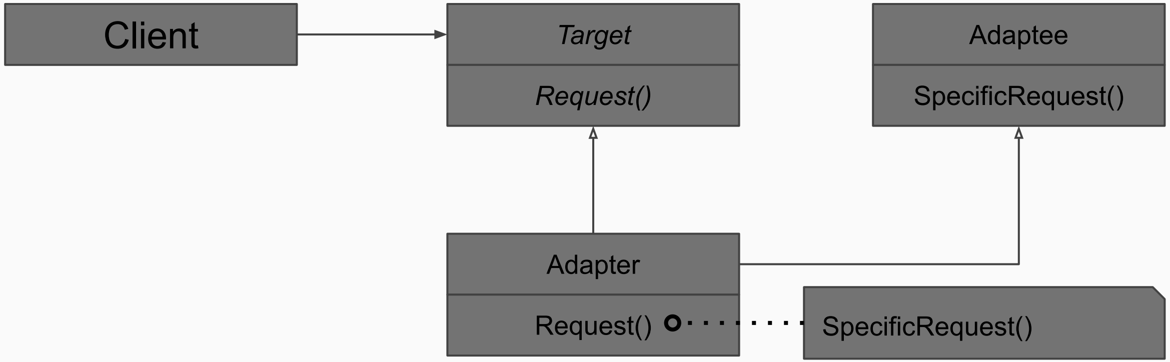


Intent
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

(Structural)

In this variation, the Adapter aggregates an instance of the Adaptee. When the Request() method is called, it is translated to the SpecificRequest() on the Adaptee.

Adapter (Class)



Intent
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
(Structural)

In this variation, the Adapter uses multiple inheritance to both implement the Target interface and extend the Adaptee. When the *Request()* method is called, the Adapter calls the *SpecificRquest()* on itself.

Consequences

Class Adapters

- ***since adapter is a subclass, it can override some of adaptee's behavior.***
- ***introduces only one new object (per adaptee)***
- ***a class adapter won't work when we want to adapt a class and its subclasses.***

Object Adapters

- ***lets a single adapter work with many adaptees; that is the adaptee itself and all of its subclasses.***
- ***makes it harder to override adaptee behavior.***