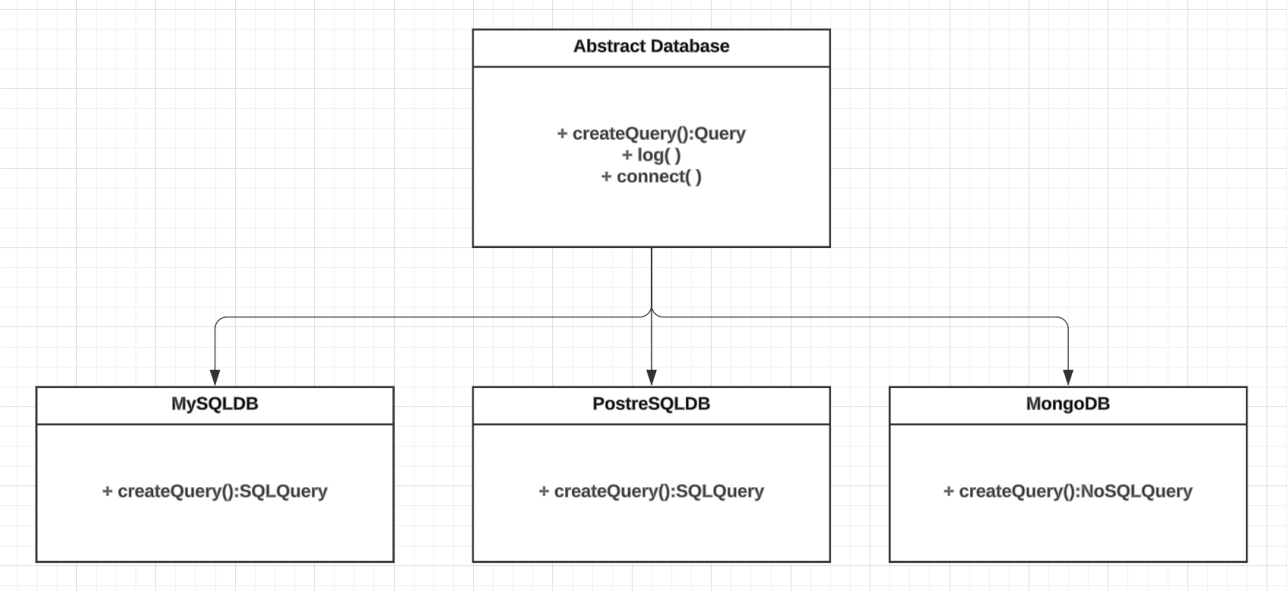
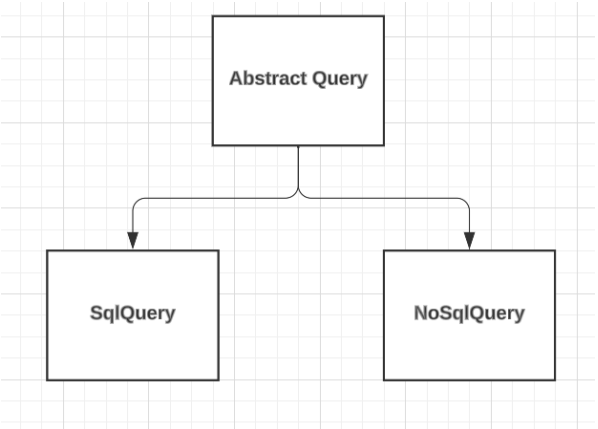
# 4. Factory Design Pattern

## Factory Method

- We have a below use case with abstract Database class with 2 child classes.
- **MySQLDb** & **PostgreSQLDb** createQuery() method will return **SQLQuery** object wherein **MongoDb** createQuery() method returns **NoSQLQuery** object.




- The abstract createQuery method in the parent Database class return the generic Query object, but the child classes of Database returns corresponding Query objects based on the database type. Because of the Inheritance parent class reference can be used to return child class objects.



- The Query method in the parent class is a factory method. The purpose if the createQuery() is to create an object of corresponding Query object based on the type of the database.

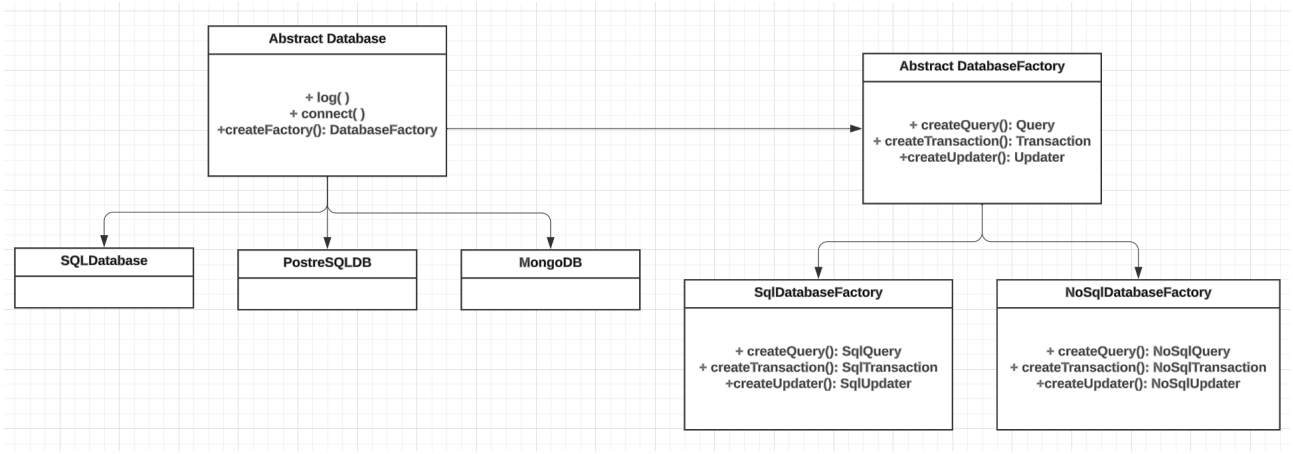
### Definition of Factory method

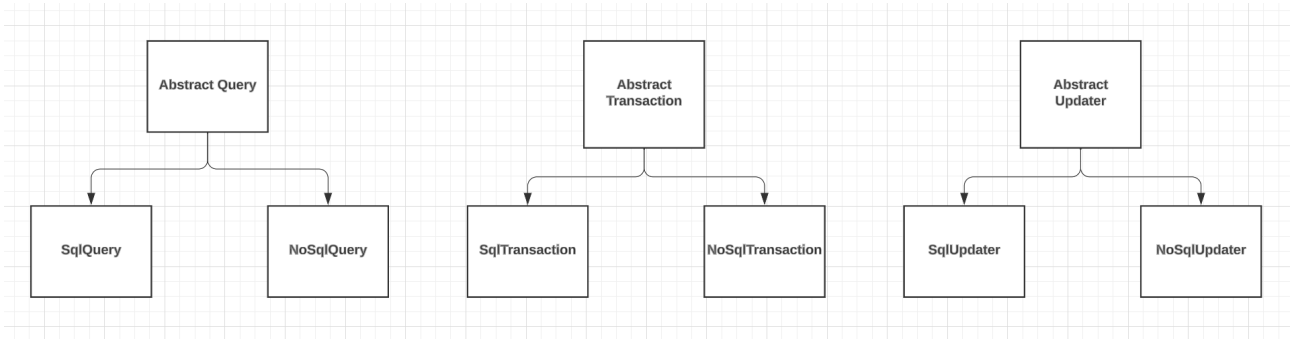
- Factory method is a method present in an interface or a parent class whose purpose is to create an object of the another/corresponding object.
- The benefits of using Factory method are,
- A new Query type can be added into the codebase easily.
- A new Database type can be added into the codebase easily.
- Code is more extensible as new features can be added quickly.
- For this pattern to work, the base Database class must work with abstract Query: a base class or an interface that all concrete Queries follow. This way the code within Database remains functional, whichever type of Queries it works with.
- For example, to add a new Database type to the app, you'll only need to create a new Database subclass and override the factory method in it.

## Abstract Factory

- Currently we have one factory method **createQuery** in the Database class & all the child classes of Databases will return corresponding **Query** objects.
- If we have few more factory methods
    - **createTransaction** - which returns **Transaction** object based on the type of the child classes
    - **createUpdater** - which returns **Updater** object based on the type of the child classes
- Now the **Database** class will have its own attributed & methods that are required it (connect method, username, password). Also it has a lot of factory methods to get an object based on the type of the Database.
- So, now the **Database** class violates single responsibility principle.

- **Abstract Factory** -> Separate the factory methods in the Database & have it all in a separate **DatabaseFactory** class. (Collection of factory methods)
- **Abstract Factory** is a creational design pattern that lets you produce families of **related objects** without specifying their concrete classes.
  - Families of related objects-> In our Database example,
  - SqlDatabaseFactory class which is child class of **DatabaseFactory** will produce objects of Query, Transaction, Updater but all those objects are related to one SQL database type.
- Have a concrete method **createFactory** in the Database class, which will return the object of **DatabaseFactory** based on some inputs
- The class structure looks like below after implementing the Abstract Factory design pattern, **DatabaseFactory** can be either abstract class or an interface



# Practical Factory:

- Whenever we have multiple variants of a particular class & we want to create an object of the correct variant based on some parameters.
- In the above Database scenario, the Database object should be created based on the type.
- We can have the practical factory implementation, which returns the type of Database object based on some parameters.