الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر تصميم نظم البرمجيات | كلية الهندسة المعلوماتية | المحاضرة الرابعة

Design Patterns:
DAO, Repository, Factory Method, Abstract Factory Class

د. رياض سنبل

DAO and Repository Patterns

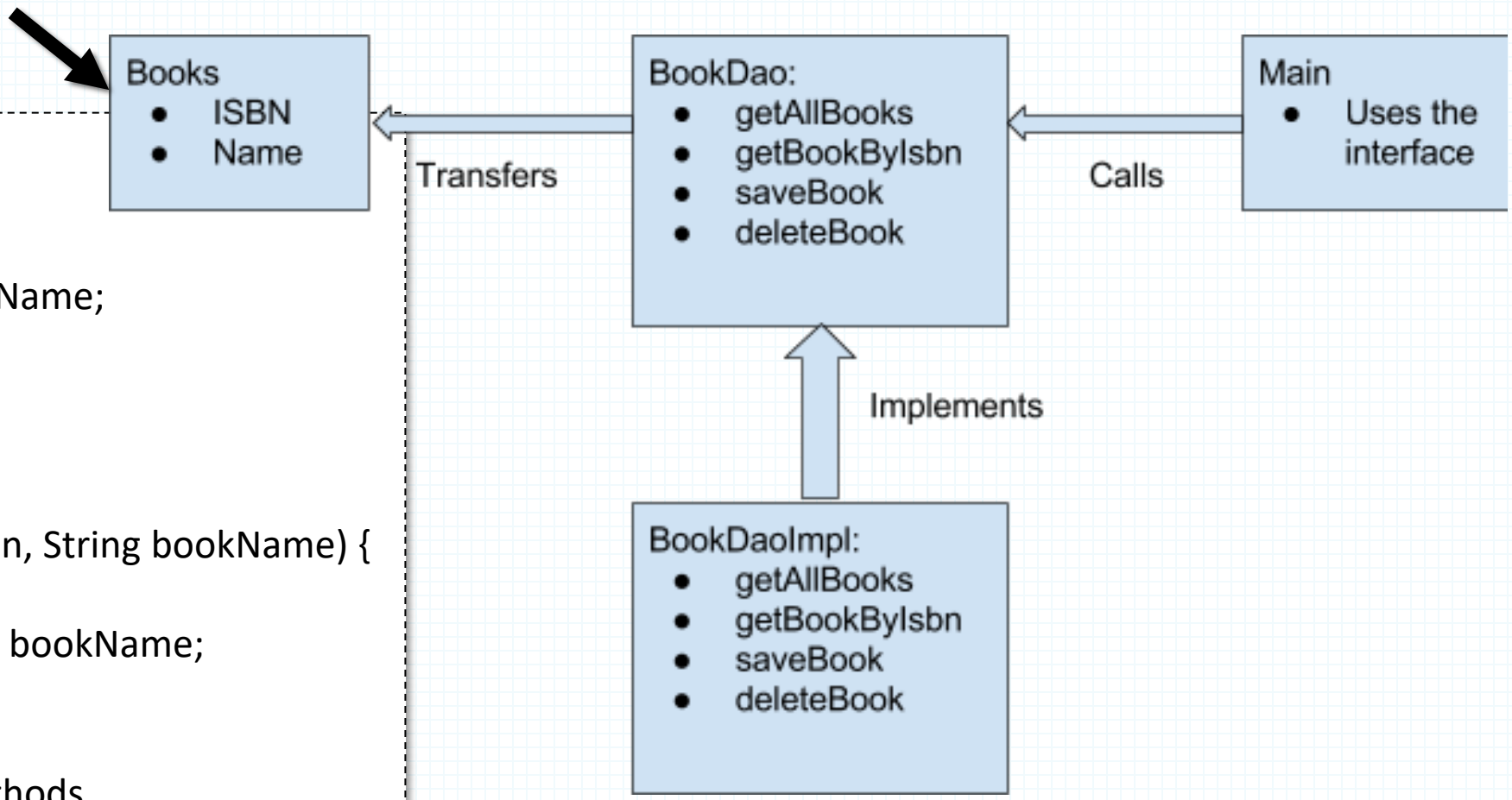
Note: These patterns can be utilized in all cases (not only for DAL), but they hold significant importance for DAL.

DAO pattern

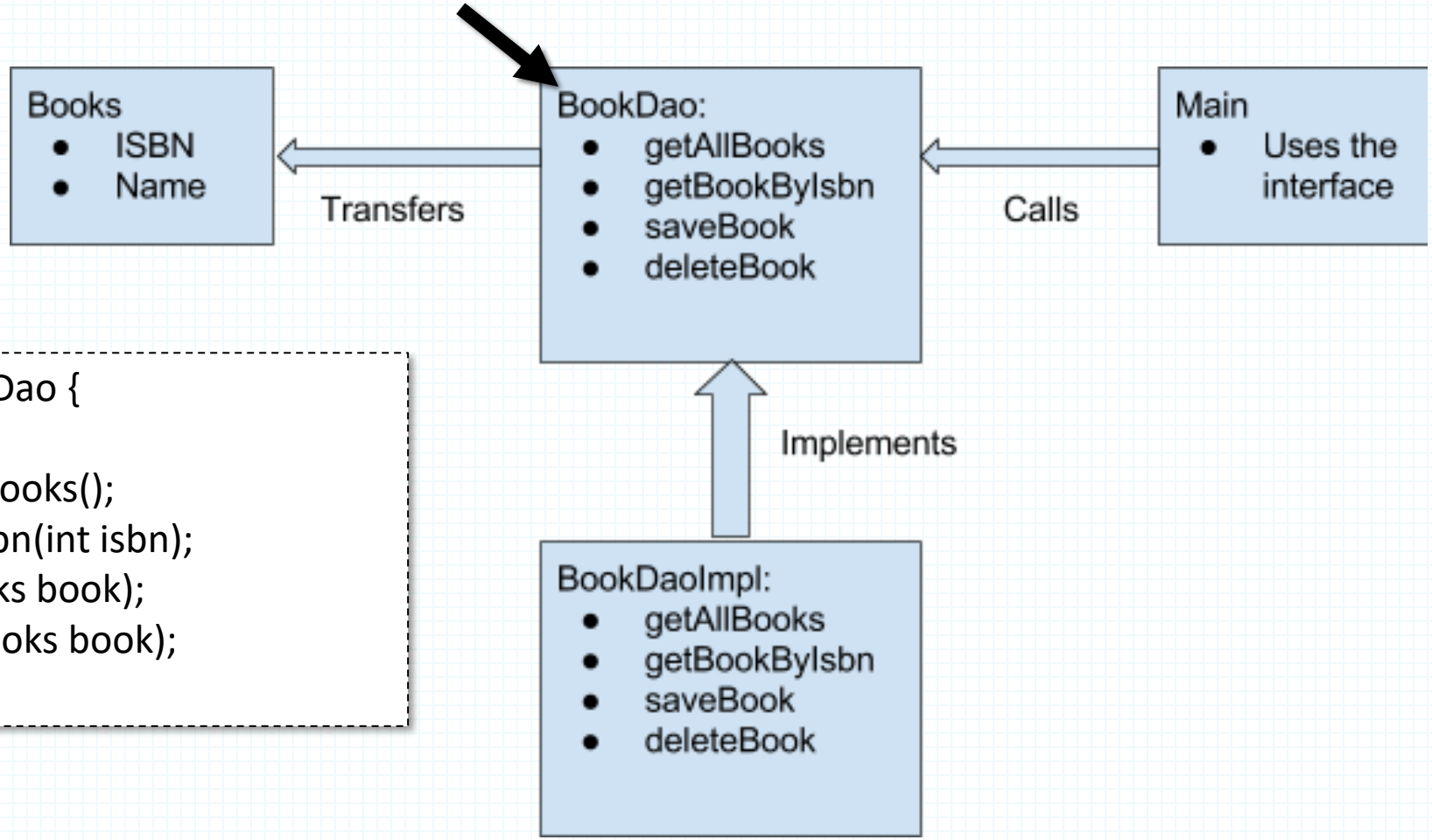
- DAO stands for Data Access Object.
- DAO Design Pattern is used to separate the **data persistence logic** in a separate layer.
- A DAO typically has a 1:1 map with a data store table or entity-set.
- With DAO design pattern, we have following components on which our design depends:
  - The model which is transferred from one layer to the other.
  - The interfaces which provides a flexible design.
  - The interface implementation which is a concrete implementation of the persistence logic.

Example

```
public class Books {

    private int isbn;
    private String bookName;

    public Books() {
    }

    public Books(int isbn, String bookName) {
        this.isbn = isbn;
        this.bookName = bookName;
    }

    // getter setter methods
}
```



Example




```
public interface BookDao {

   List<Books> getAllBooks();
   Books getBookByIsbn(int isbn);
   void saveBook(Books book);
   void deleteBook(Books book);
}
```

Repository Pattern

- A repository acts at a higher level of abstraction working with aggregations of business entities
- Abstraction layer between business logic layer and Data access layer.
- This repository acts as an in-memory collection of domain objects, providing a *clean* and *consistent* API for the application to perform CRUD (Create, Read, Update, Delete) operations.
- **DAO vs Repository:**
  - DAO would be considered closer to the database, often table-centric.
  - Repository would be considered closer to the Domain, dealing only in Aggregate Roots.
  - Repository could be implemented using DAO's, but you wouldn't do the opposite.

```java
public class Country {
    private Long id;
    private String name;
    private String abbreviation;
    private int statement;
}
```

```java
public interface CountryRepository {
    Country fetch(Long id);
    Long save(CountryEntity country);
    void update(CountryEntity country);
    int delete(Long id);
}
```

this pattern uses more than one DAO to aggregate data and publish it via domain object.

This pattern should be used to prepare complex domain objects so the business layer

```java
public class CountryRepositoryImpl implements CountryRepository {
    private CountryDAO countryDAO;
    private StateDAO stateDAO;

    Country fetch(Long countryId) {
            CountryEntity countryEntity =
countryDAO.fetch(countryId);
            StateEntity stateEntity = stateDAO.count(countryId);

            // Country Mapper to copy data from the country entity
and state entity and prepare country object (domain object)
    }

    Long save(Country country) {
        return countryDAO.save(country);
    }

    // ...
}
```

Note: Using Generic Programming

```
1 reference
public interface IRepository<T>
{
    0 references
    IEnumerable<T> GetAll();
    0 references
    T GetByID(int id);
    0 references
    void Add(T item);
    0 references
    void Update(T item);
    0 references
    void Delete(T item);
}
```

If you are using these operations on top of an ORM like EF or NHibernate, all this translates to an abstraction over another abstraction.

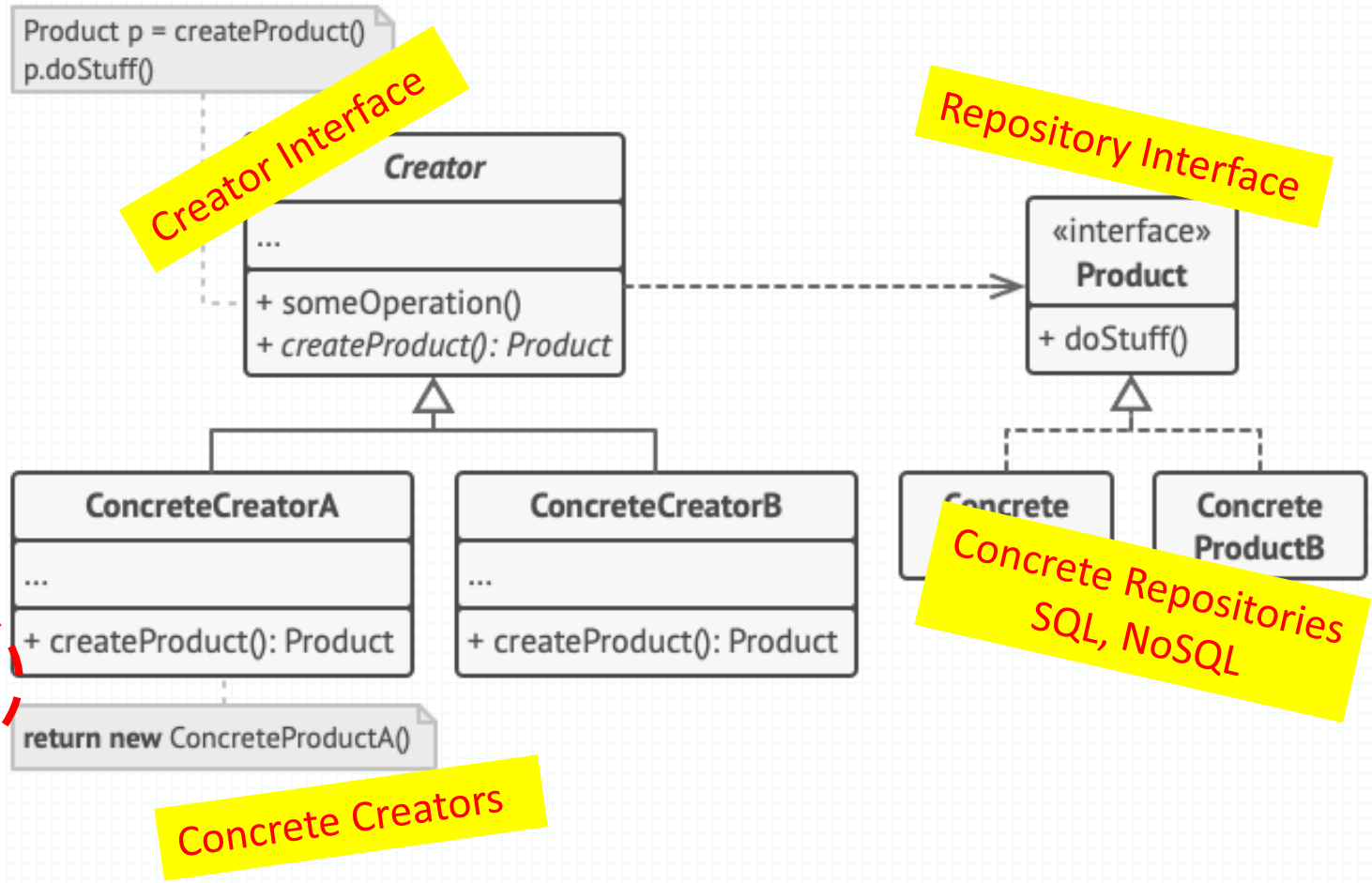
A Generic Repository doesn't define a meaningful contract. The operations defined as not that expressive.

```
public class AuthorRepository : IRepository<Autho

  {

     //Implemented methods of the IRepository
interface

  }
```

Factory Method Pattern

Factory Method Pattern

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- Just define an interface or abstract class for creating an object (ex. Repository) but let the subclasses (ex. SQL Repo. Or NoSQL Repo.) decide which class to instantiate.



```
// Interface for DataRepository with CRUD operations
interface DataRepository {
    void fetchData();
    void addData(Object data);
    void updateData(Object data);
    void deleteData(int dataId);
}
```

```
// Factory Method: DataRepositoryFactory
interface DataRepositoryFactory {
    DataRepository createRepository();
}
```

```
// Concrete impl for SQL databases with CRUD ops
class SQLDataRepository implements DataRepository
{
        …….
}
```

```
// Concrete implementation for SQL databases
class SQLDataRepositoryFactory implements DataRepositoryFactory {
    @Override
    public DataRepository createRepository() {
        return new SQLDataRepository();
    }
}
```

```
// Concrete impl for NoSQL databases with CRUD ops
class NoSQLDataRepository implements
DataRepository {
        …….
}
```

```
// Concrete implementation for NoSQL databases
class NoSQLDataRepositoryFactory implements DataRepositoryFactory
{
    @Override
    public DataRepository createRepository() {
        return new NoSQLDataRepository();
    }
}
```

Client Code

```
// Client code using Factory Method with CRUD operations
public class MainFactoryMethod {
    public static void main(String[] args) {
        DataRepositoryFactory sqlFactory = new SQLDataRepositoryFactory();
        DataRepository sqlRepository = sqlFactory.createRepository();

        sqlRepository.addData(new Object());
        sqlRepository.fetchData();

        DataRepositoryFactory noSqlFactory = new NoSQLDataRepositoryFactory();
        DataRepository noSqlRepository = noSqlFactory.createRepository();

        noSqlRepository.updateData(new Object());
        noSqlRepository.fetchData();
    }
```

create objects without specifying the exact class of the object that will be created

you can abstract the process of object creation from the client code (Separation of Concerns)

The creation logic is encapsulated Different databases may require different setup or configuration steps during the creation of the repository

Improve testability by facilitating the use of mock or test implementations. You can create mock repositories or substitute implementations to isolate and test specific components without relying on real database connections.

If you need to add support for a new type of data repository, you can create a new factory class without modifying existing client code

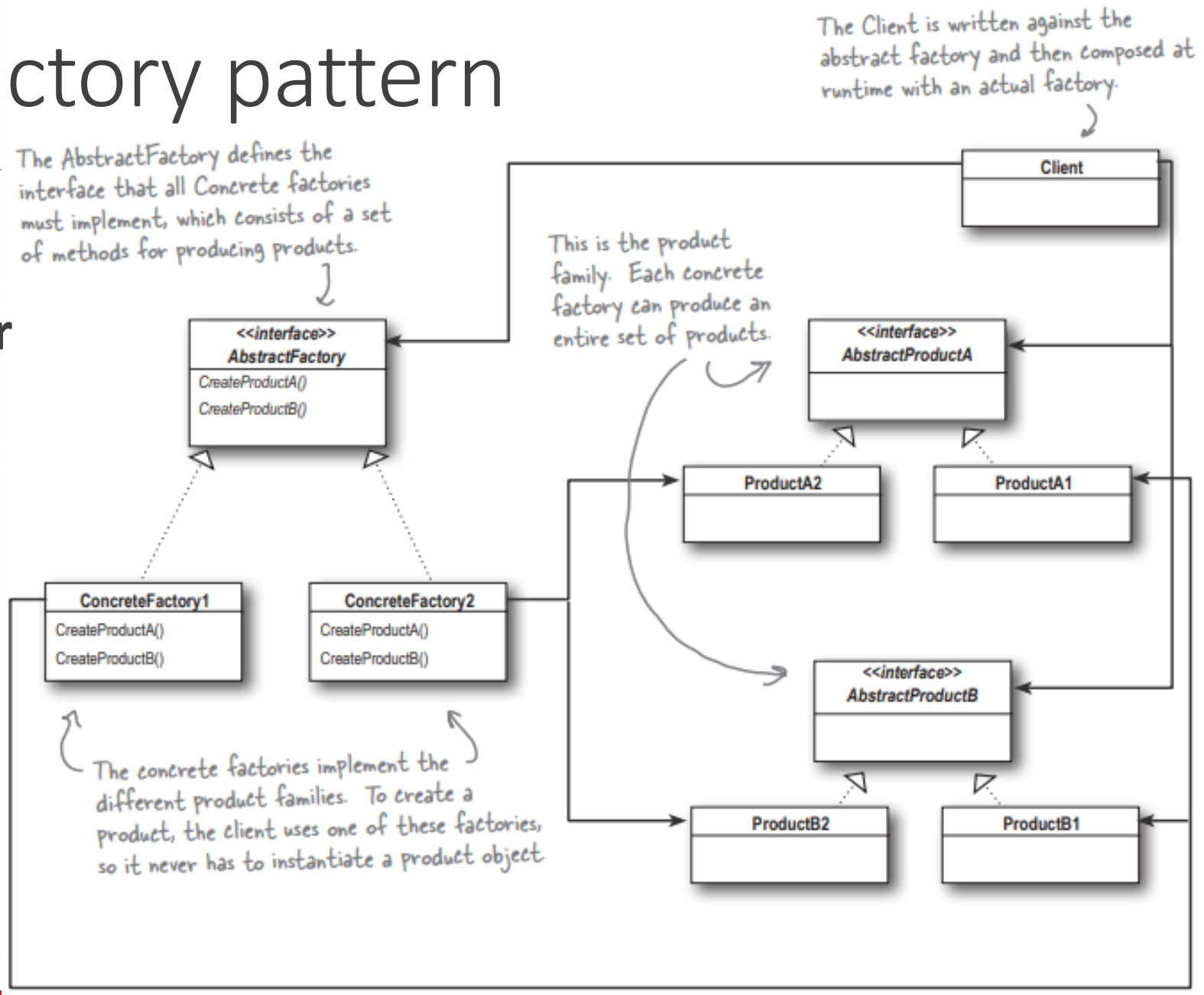
Factory Method Pattern

- **Benefits:**
  - **Flexibility:** Clients can work with the abstract creator interface, unaware of the specific repository implementations. This allows for seamless substitution of repository types without modifying client code.
  - **Encapsulation:** The creation logic is encapsulated within the concrete creator classes, promoting a clean separation of concerns.

The Abstract Factory pattern

The Abstract Factory pattern

- It provides an interface for **creating families of related or dependent objects** without specifying their concrete classes.
- In the context of the DAL, the Abstract Factory pattern extends the Factory Method concept to **create families of repositories**.



Example

How Can you redesign the data access layer in your project in a better way?