Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Software Engineering

The Factory Method Design Pattern

For details see Gamma et al. in “Design Patterns”

TECHNISCHE UNIVERSITÄT DARMSTADT

The Factory Method Design Pattern

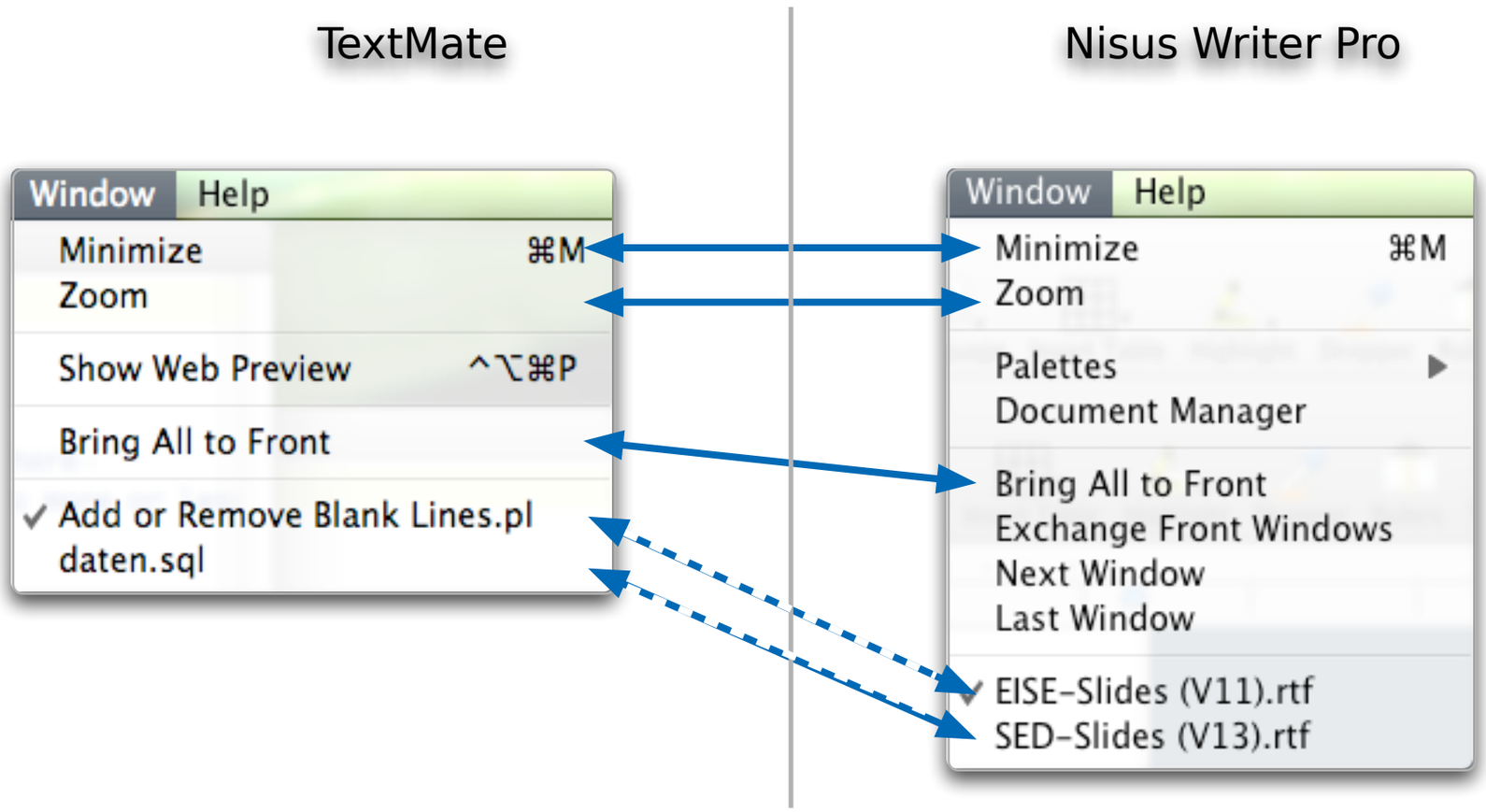
Example / Motivation

- Let's assume we want to develop a framework for applications that can present multiple documents to the user (MDI style).
- We want to support a wide variety of applications:
  - Text editors
  - Word processors
  - Vector drawing applications
  - Document Viewers
  - ...
- Our framework should - in particular - be able to manage the documents.

The Factory Method Design Pattern

Example / Motivation -

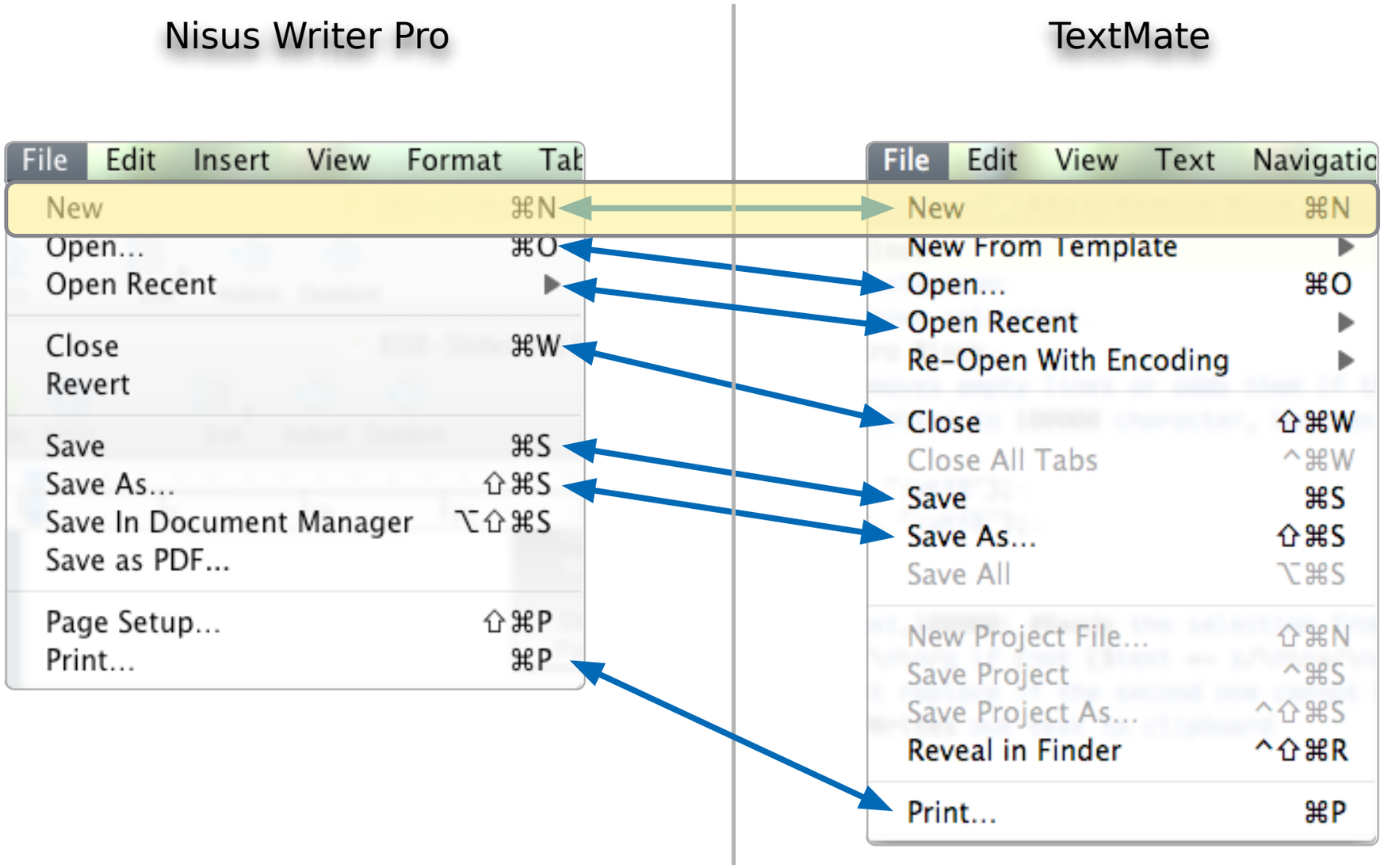
Common functionality for handling documents



The Factory Method Design Pattern

Example / Motivation -

Common functionality for handling documents



(In the following, we focus on the implementation of “New”.)

The Factory Method Design Pattern

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate.

(Factory Method lets a class defer instantiation to subclasses.)

The Factory Method Design Pattern

Example / Motivation -

A Possible Implementation of the Framework


```
public abstract class Document {
   public abstract void open();
   public abstract void close();
}
```

```
public abstract class Application {
   private List<Document> docs = new ArrayList<Document>();
   public void newDocument() {
      Document doc = createDocument();
      // the framework manages the documents
      docs.add(doc);
      doc.open();
   }
   ...
   public abstract Document createDocument(); // factory method
}
```

The Factory Method Design Pattern

Example / Motivation -

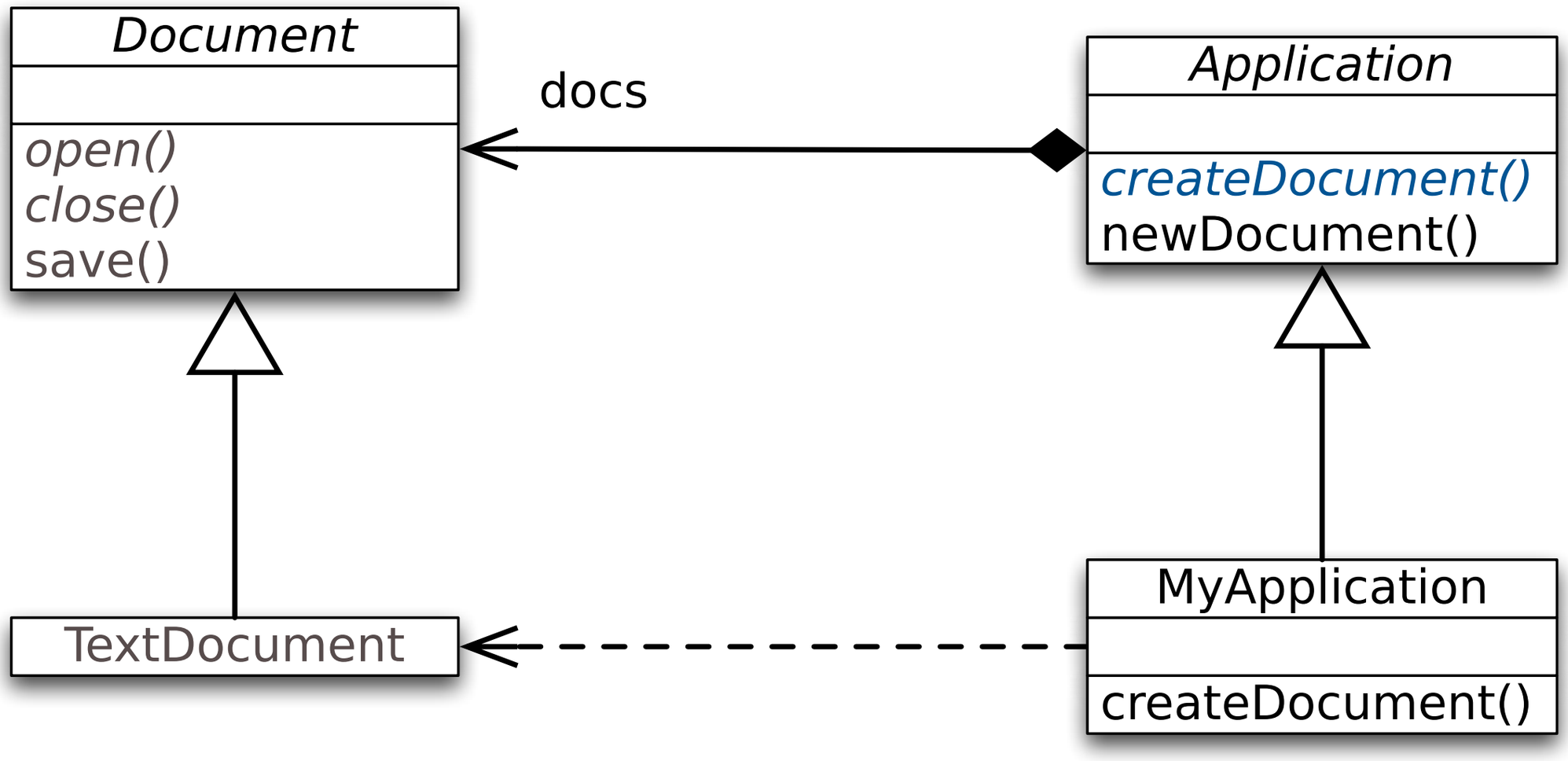
Implementation of an Application Using the Framework


```
public class TextDocument extends Document {
   … // implementation of the abstract methods
}

public class MyApplication extends Application {

   public Document createDocument() {
      return new TextDocument();
   }

}
```

The Factory Method Design Pattern

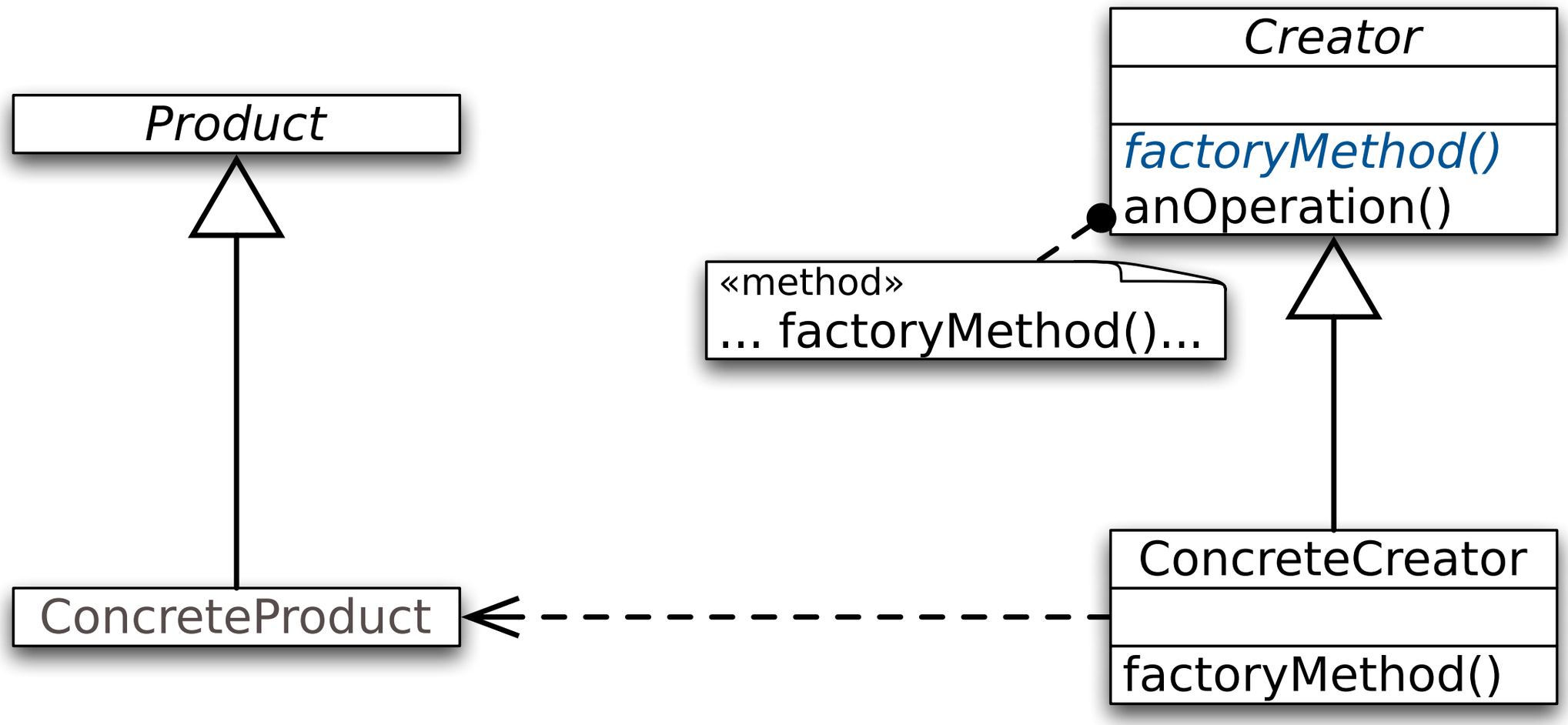
Example / Motivation -

Class Diagram of an Application Using the Framework



The Factory Method Design Pattern

Structure



The Factory Method Design Pattern

Participants

- `Product`
  … defines the interface of objects the factory method creates.
- `ConcreteProduct`
  … implements the Product interface.
- `Creator`
  … declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- `ConcreteCreator`
  … overrides the factory method to return an instance of a ConcreteProduct.

The Factory Method Design Pattern

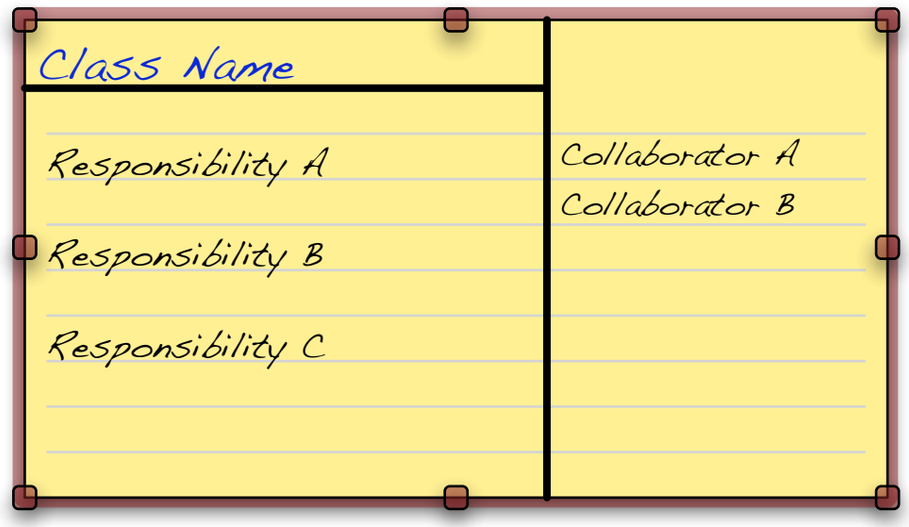
Consequences (I)

- **The framework's code only deals with the Product interface**; therefore it can work with any user-defined ConcreteProduct class.
- **Provides a hook for subclasses**
  The hook can be used for providing an extended version of an object.

The Factory Method Design Pattern

Consequences (II)

- Connects parallel class hierarchies



The Factory Method Design Pattern

Implementation

Two major variants:

- Creator is abstract
- Creator is concrete and provides a reasonable default implementation

The Factory Method Design Pattern

Implementation - Parameterized factory methods

(E.g. imagine a document previewer which can handle very different types of documents.)

General form:

```
public abstract class Creator {
     public abstract Product createProduct(ProductId pid);
}
```

Applied to our example:

```
public abstract class Application {
     public abstract Document createDocument(Type e);
}
public class MyApplication extends Application {
     public Document createDocument(Type e){
          switch(e) {
               case Type.JPEG : return new JPEGDocument();
               case Type.PDF : return new PDFDocument();
          }
}    }
```

The Factory Method Design Pattern

Implementation - Parameterized factory methods


```
public abstract class Application {

   private Class<? extends Document> clazz;

   public Application(Class<? extends Document> clazz){
      this.clazz = clazz;
   }

   public abstract Document createDocument(){
      return clazz.newInstance();
   }
}
```

It is possible to use Java reflection in a type safe way.

The Factory Method Design Pattern

Related Patterns

- Factory Methods are usually called within Template Methods
- Abstract Factory is often implemented with factory methods

Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Software Engineering

The Abstract Factory Design Pattern

For details see Gamma et al. in “Design Patterns”

TECHNISCHE UNIVERSITÄT DARMSTADT

How to create families of related classes that implement a (set of) common interface(s)?

The Abstract Factory Method Design Pattern

Motivation / Example Scenario

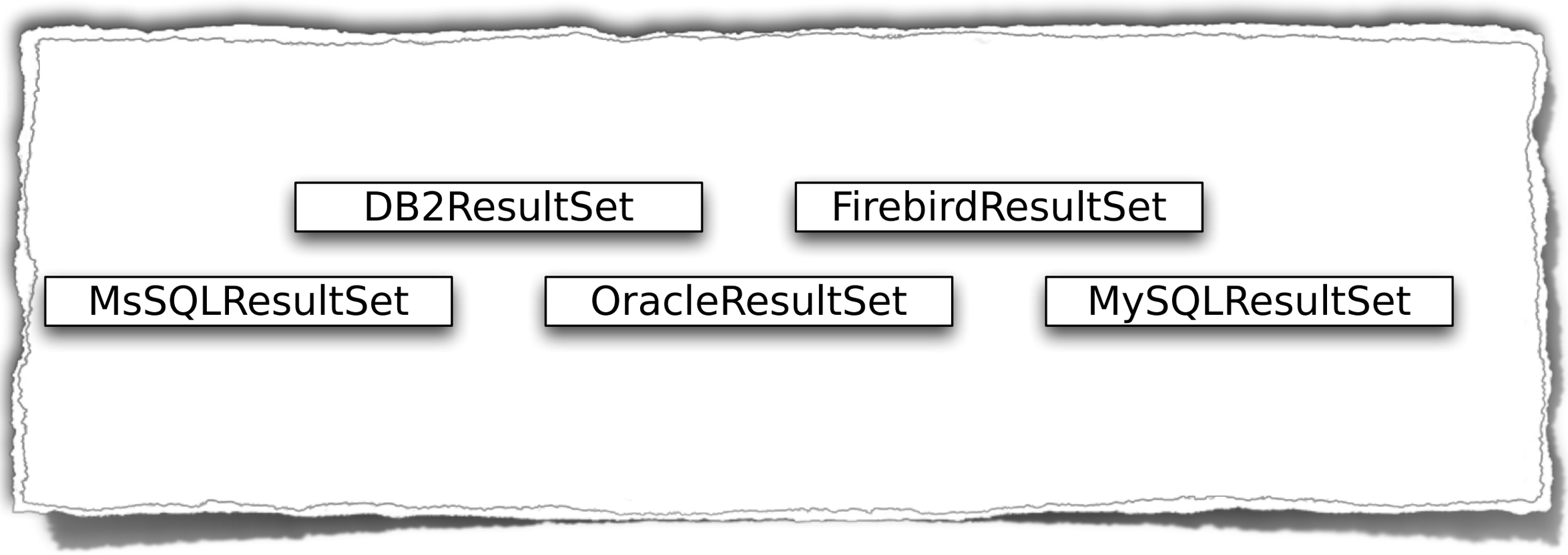
Our goal is to support different databases.

Requirements:

- The application should support several databases
  (We want to be able to change the database at startup time.)
- We want to support further databases
  (We want to make the implementation unaware of the specific database(s).)

Supporting Variety

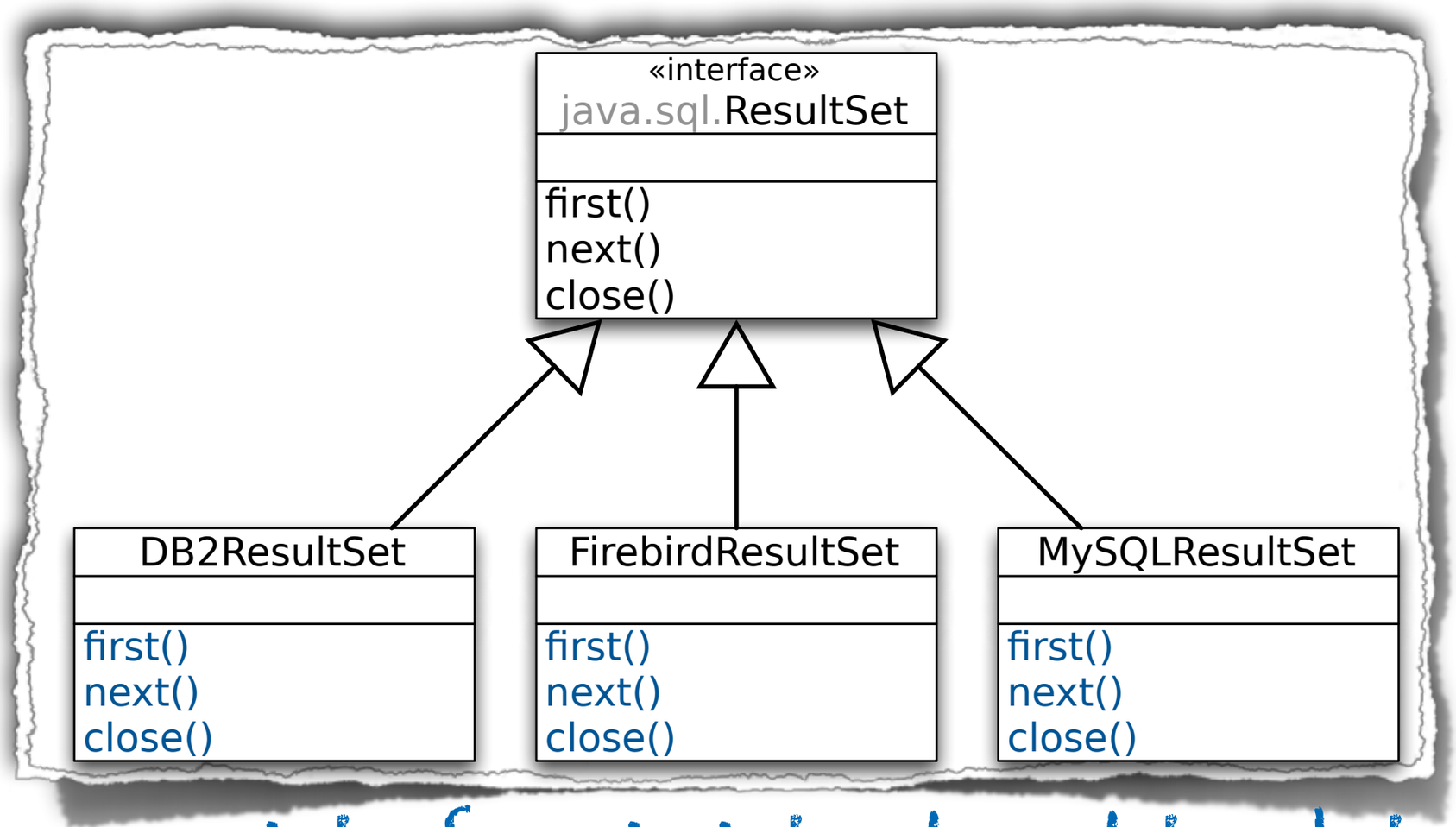
A result set enables the iteration over the result of an SQL query.



How to provide an interface to all of these different kinds of ResultSets?

Supporting Variety by Providing a Common Interface

A result set enables the iteration over the result of an SQL query.



A common interface is introduced to abstract from the concrete classes.

The Abstract Factory Method Design Pattern

Motivation / Example Scenario

- To complete the abstraction of the database, one also needs to create class hierarchies for: `CallableStatements`, `PreparedStatements`, `Blobs`, ...
- The code interacting with the database can now deal with `ResultSets` and SQL statements without referring to the concrete classes, e.g., Firebird-ResultSet
- However, we still have to know the concrete implementation subclass at creation time!

The Abstract Factory Method Design Pattern

Issues

- How can we avoid to know about the concrete product types at creation time?

  We want to avoid to write:

  PreparedStatement = new FBPreparedStatement();

- Hard-coding product types as above makes it impossible to select a different database
- Even offline changes are difficult as it is easy to miss one constructor and end up with FireBird's FBPreparedStatement while a DB2 database is used

Issues -

How can we avoid to know about the concrete product types at creation time?

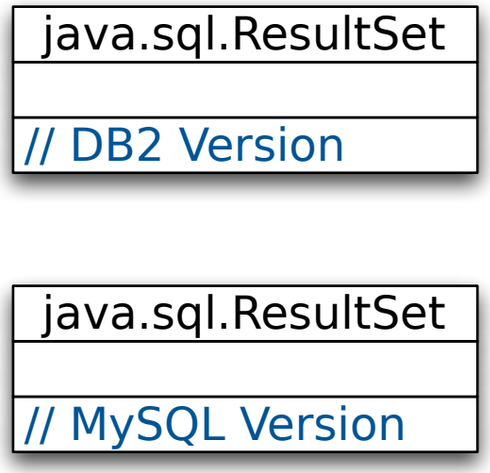
Solution

Swapping Code

- Swap in and out different files when compiling for a different database
- Does neither require subclassing nor a special creation logic

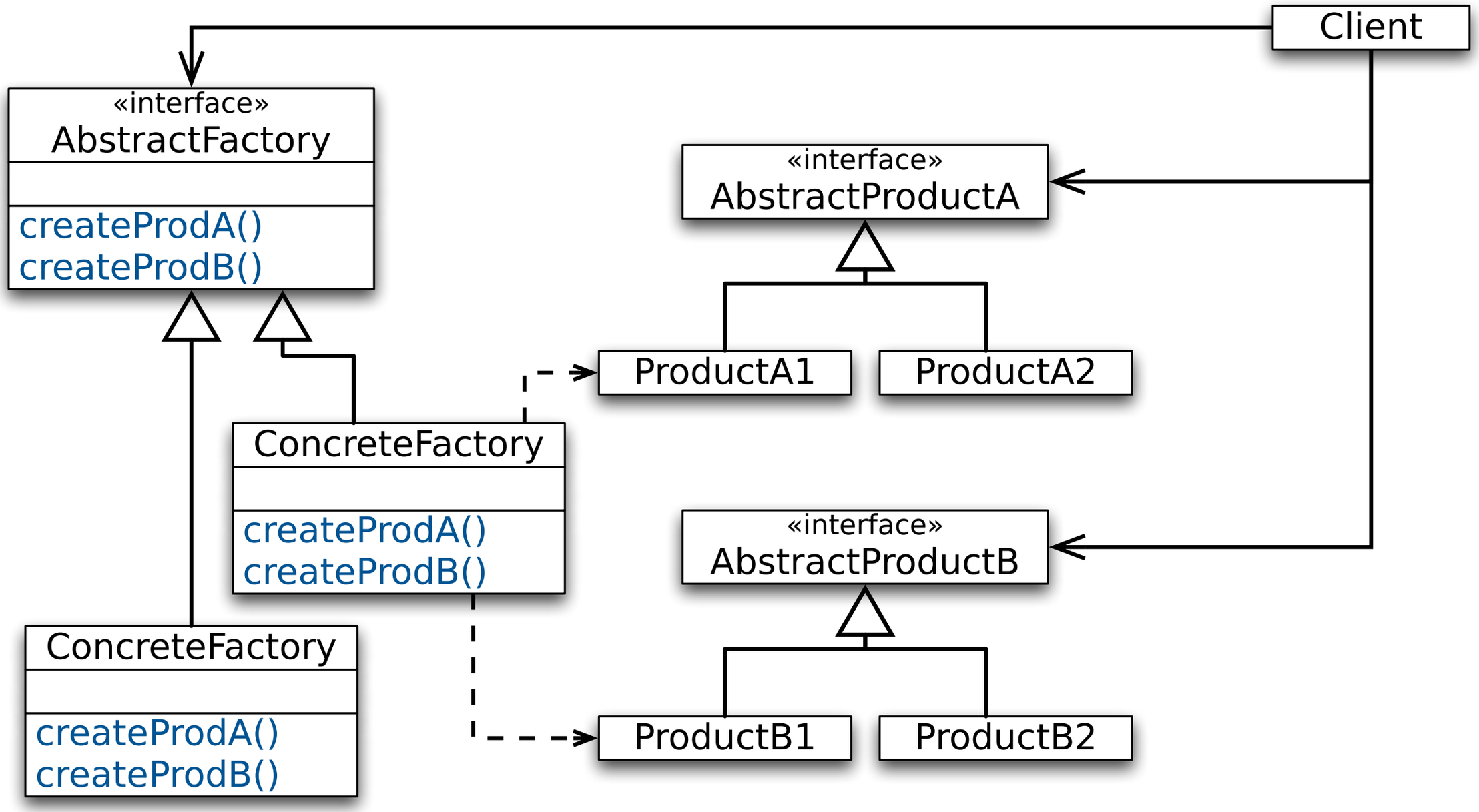
Trade-offs

- Application code is completely unaware of different databases
- Needs configuration management of source files
- Does not allow different databases to be chosen at startup, e.g., if more than one is supported
- Does not allow multiple databases to be used at runtime



The Abstract Factory Method Design Pattern

Structure



The Abstract Factory Method Design Pattern

Participants

- `AbstractFactory`
  … provides an interface for creating products of a family
- `ConcreteFactory`
  … implements the operations to create concrete products
- `AbstractProduct`
  … declares the interface for concrete products
- `ConcreteProduct`
  … provides an implementation for the product created by the corresponding ConcreteFactory
- `Client`
  … creates products by calling the ConcreteFactory;
  uses the AbstractProduct interface

The Abstract Factory Method Design Pattern

Consequences

- Abstracts away from concrete products
  (Clients can be ignorant about concrete products they are using, even at creation time.)
- Exchanging product families is easy
  (Changing one line can completely swap the behavior of a whole product family.)
- Ensures consistency among products
  (As family selection is concentrated to one line, one may not accidentally mix product types.)
- Supporting new kinds of products is difficult
  (Adding new products involves changing the abstract factory and all of its subclasses.)
- Creation of objects is non-standard
  (Clients need to know to use the factory rather than a constructor.)

The Abstract Factory Method Design Pattern

Issues -

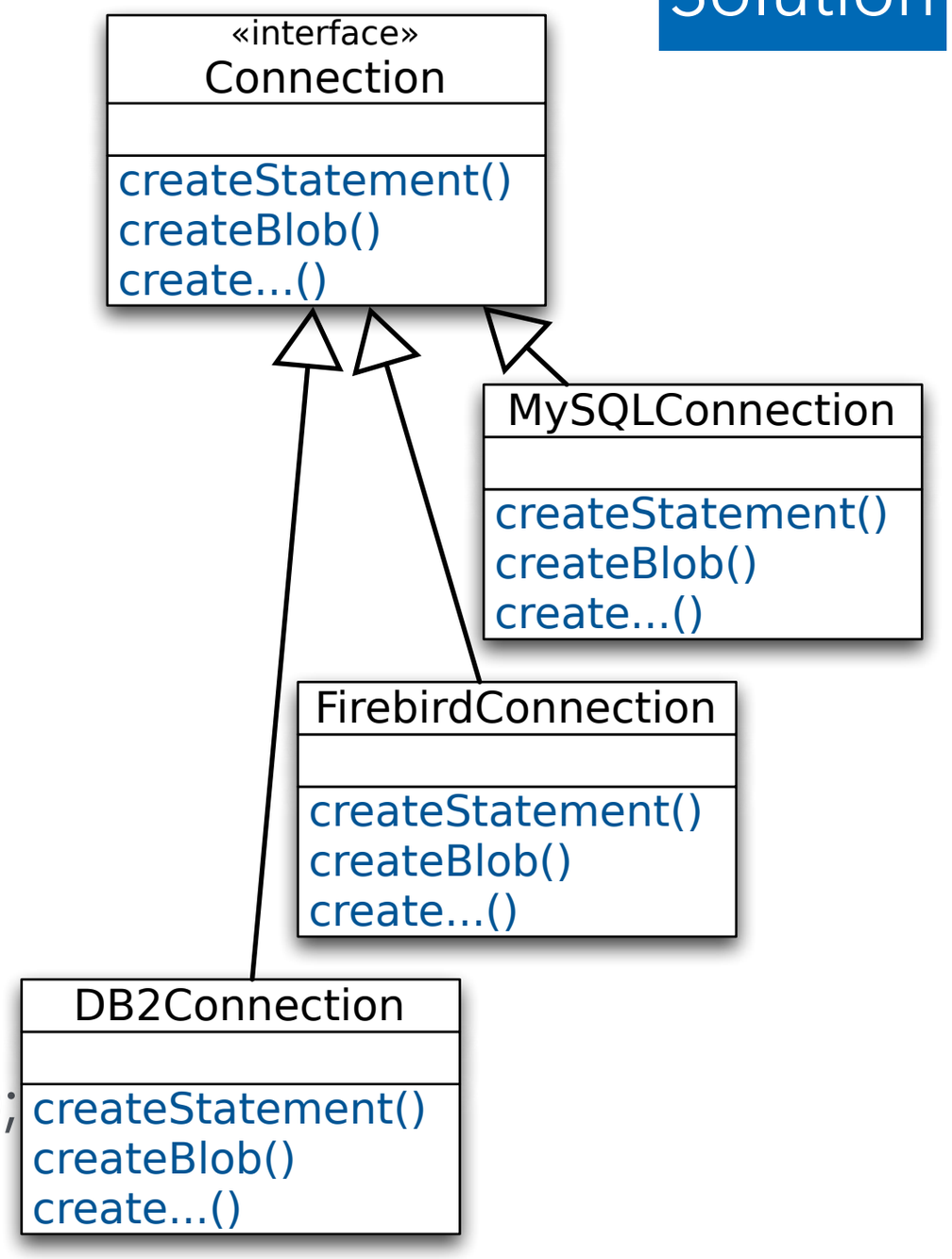
How can we avoid to know about the concrete product types at creation time?

Solution

Factory Class

- Group creation functions into a special "factory" class responsible for creating the objects to interact with the database on request.
- Has functions like... createStatement(), createBlob() and prepareStatement() as part of its interface
- Different factory subclasses provide implementations for different databases.

Statement s = connection.createStatement();



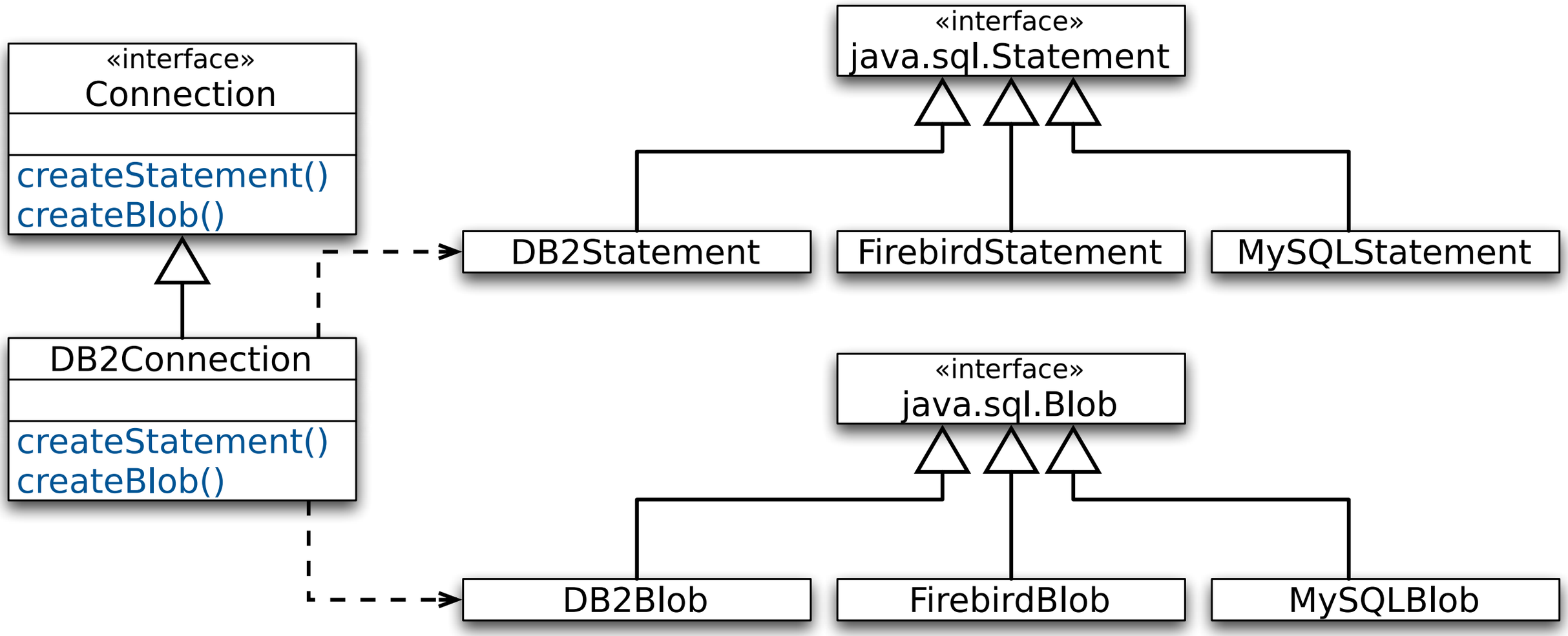
The Abstract Factory Method Design Pattern

Product Creation

- Creation of database objects is done by accessing the global variable connection of type Connection (the "factory")
  `Statement = connection.createStatement();`
- To interact with a different database the connection is initialized differently:
  ```
  connection =
   DriverManager.getConnection("org.postgresql.Driver")
  ```
  or
  ```
  connection =
   DriverManager.getConnection("org.mysql.Driver")
  ```
- We can make the initialization value for `DriverManager.getConnection` a parameter of the application

The Abstract Factory Method Design Pattern

Applied

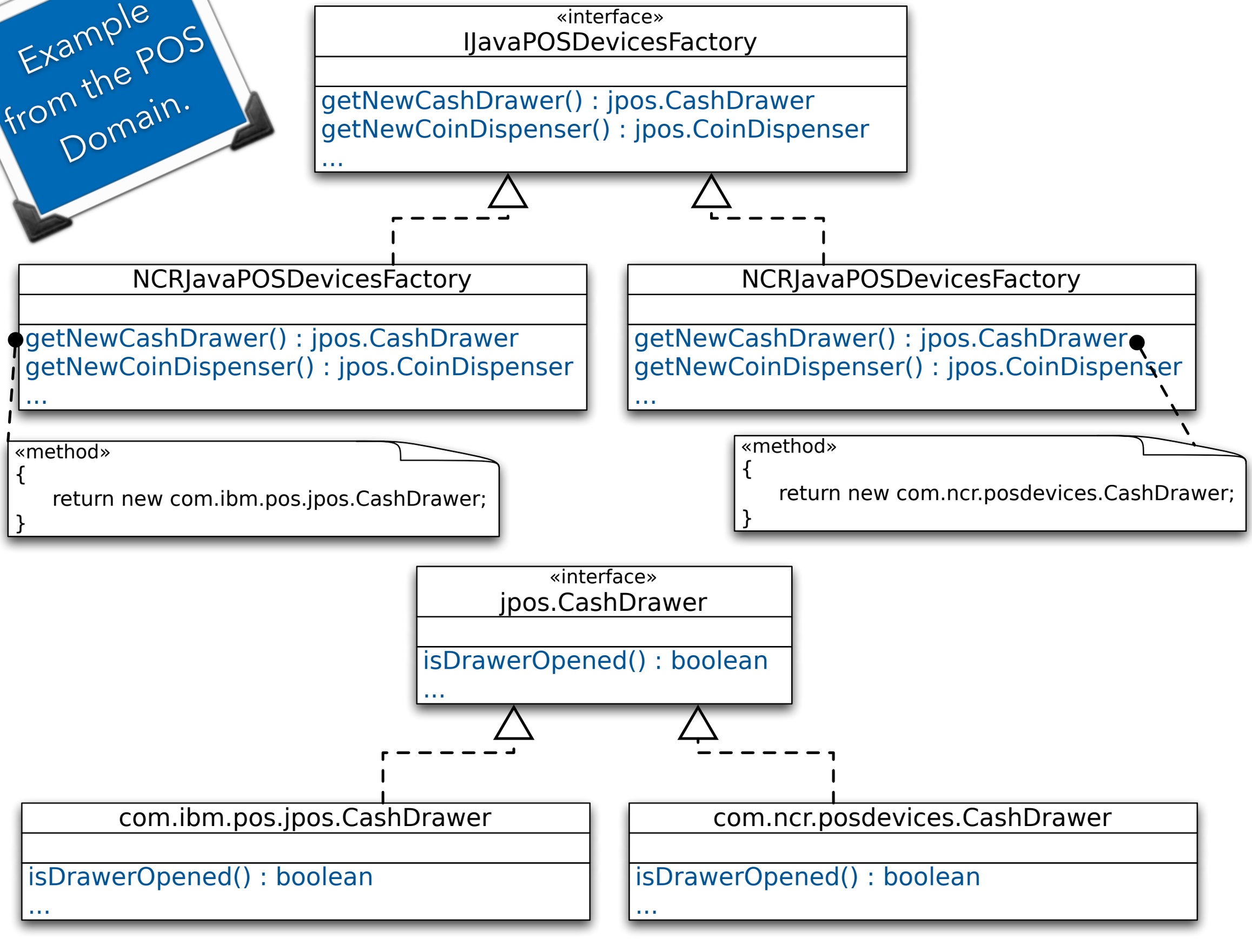


The Abstract Factory Method Design Pattern

Summary

- Application code can be ignorant about different databases
- Only one line of code (or configuration parameter) must vary to support various databases
- Allows different databases to be chosen at startup
- Enforces creation of consistent product families
  (Prevents FBBlob from being used with a DB2 database.)
- Code must follow a new convention for creating products from a family
  (Instead of using the standard constructor.)

Example from the POS Domain.



The Abstract Factory Method Design Pattern

Related Patterns

- A concrete factory is often a singleton