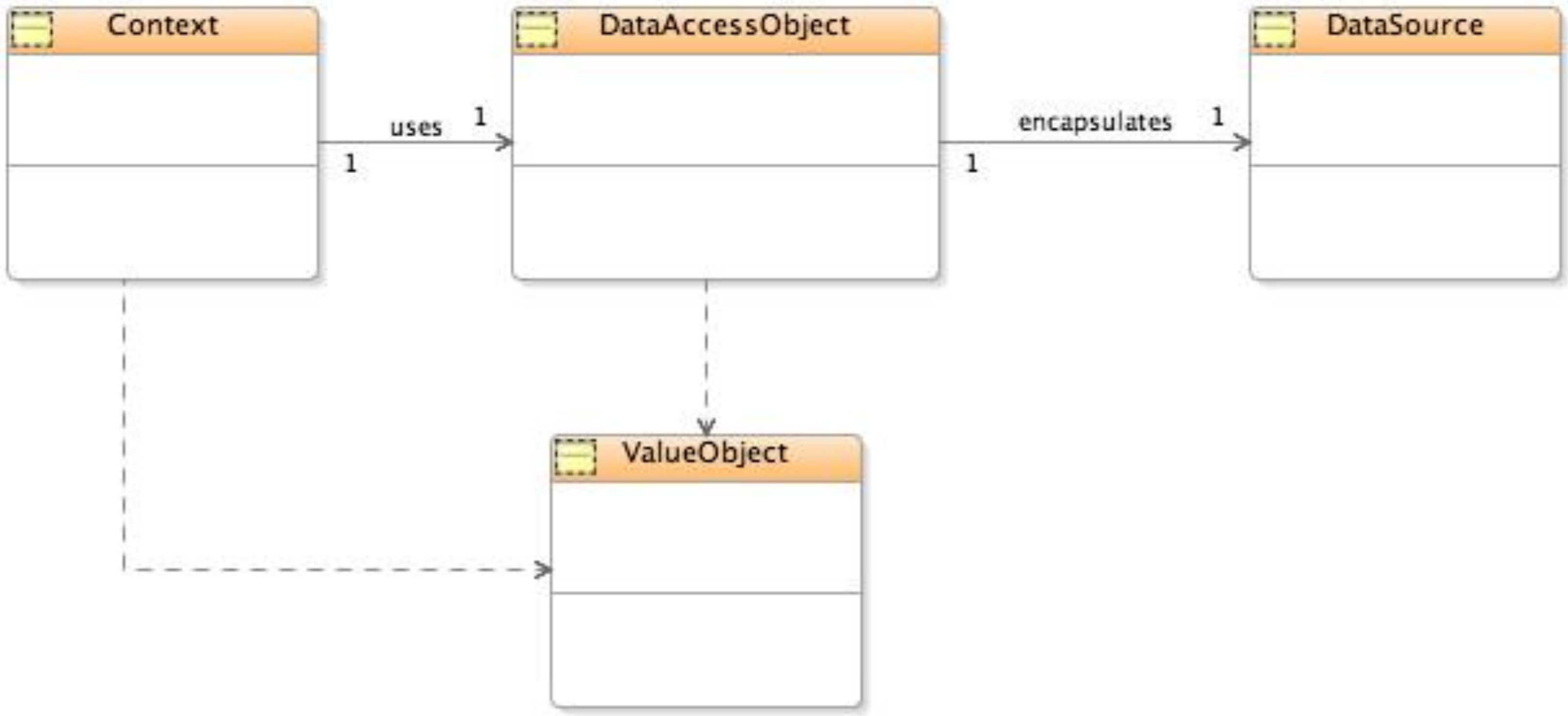
DAO and JNDI & DataSource

Topics

- DAO Pattern
  - Factory Patterns
- JNDI
  - Initialization
  - Using DataSource

DAO Pattern



Context
DataAccessObject
DataSource
ValueObject
uses
1
1
encapsulates
1
1

DAO Components

- `Context`
  - Data client
- `DataAccessObject`
  - Abstracts the access implementation. Data load and store delegate.
- `DataSource`
  - Data source implementation.
- `ValueObject`
  - Data carrier.

Implementation

- `DataAccessObject`
  - Defines methods for retrieving and storing object or its properties
  - Choice of methods defined by client need
- `ValueObject`
  - Value only
    - Simple JavaBean
    - May simply consist of public attributes
  - `Serializable`

More About Factories

- Factory Method
  - Parameterized Factory
- Abstract Factory

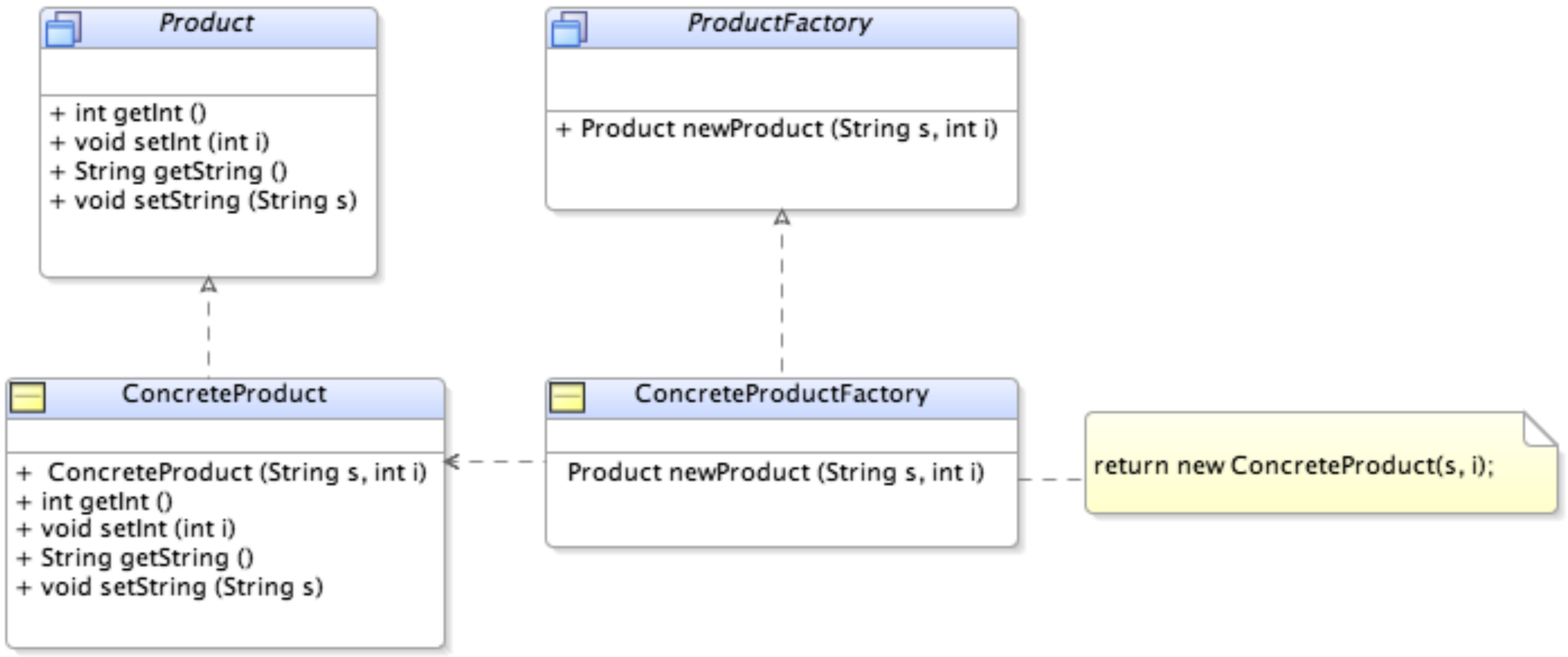
Factory Method Problem

- How to create instances of as yet undefined class which implement an interface or extend an abstract class
- Consider
  - Data access interface
    - XML implementation
    - RDBMS implementation
    - User defined persistence implementation
  - When accessing data what class needs to be instantiated

Factory Method Solution

- A class for creating instances of other classes
  - Set of methods for creating objects
- Factory variants
  - Concrete factory
    - Creates predefined default implementation
  - Abstract factory
    - Subclasses responsible for creating objects
  - Parameterized factory
    - Factory method argument dictates the class of the object to be created

Factory Method



Product
+ int getInt ()
+ void setInt (int i)
+ String getString ()
+ void setString (String s)
ProductFactory
+ Product newProduct (String s, int i)
ConcreteProduct
+ ConcreteProduct (String s, int i)
+ int getInt ()
+ void setInt (int i)
+ String getString ()
+ void setString (String s)
ConcreteProductFactory
Product newProduct (String s, int i)
return new ConcreteProduct(s, i);

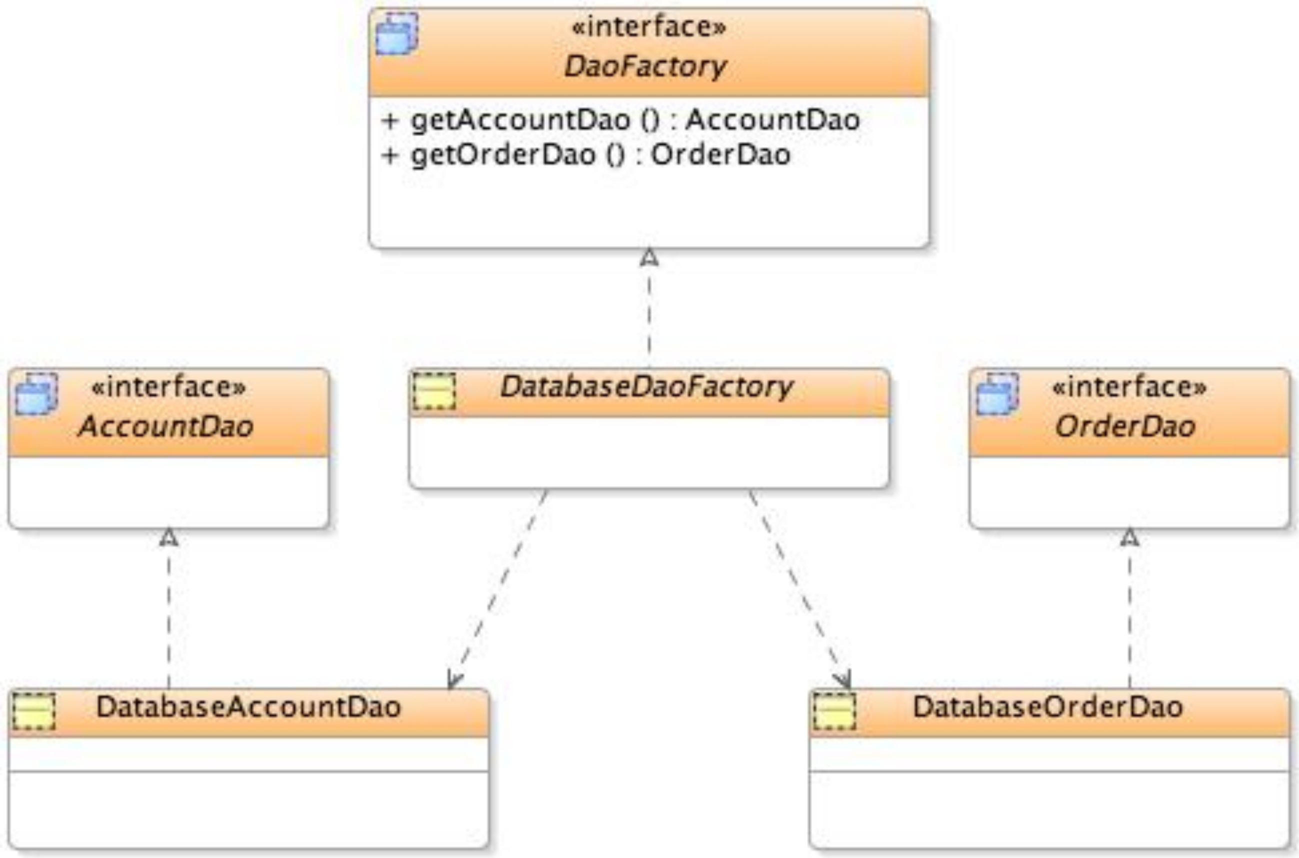
Factory Method Solution

- Defining Characteristics
  - Factory class defines method for creating an abstract or at least non-final class
  - Subclasses of Factory override the factory method
  - Simplest form may simply provide descriptive methods for creating instance
- Discussion
  - An example
    - Various implementations of data persistence
    - Allow for introduction of new implementations

Factory Method Consequences

- Factory method provides a “hook” used to create instances of subclasses of the predefined classes
- Can connect parallel class hierarchies
  - Each class in the hierarchy requires a unique “helper” class
  - Each class’ factory method creates an instance of its unique helper class
- Required to define the concrete subclass where the factory class is abstract
- Configure application with proper factory

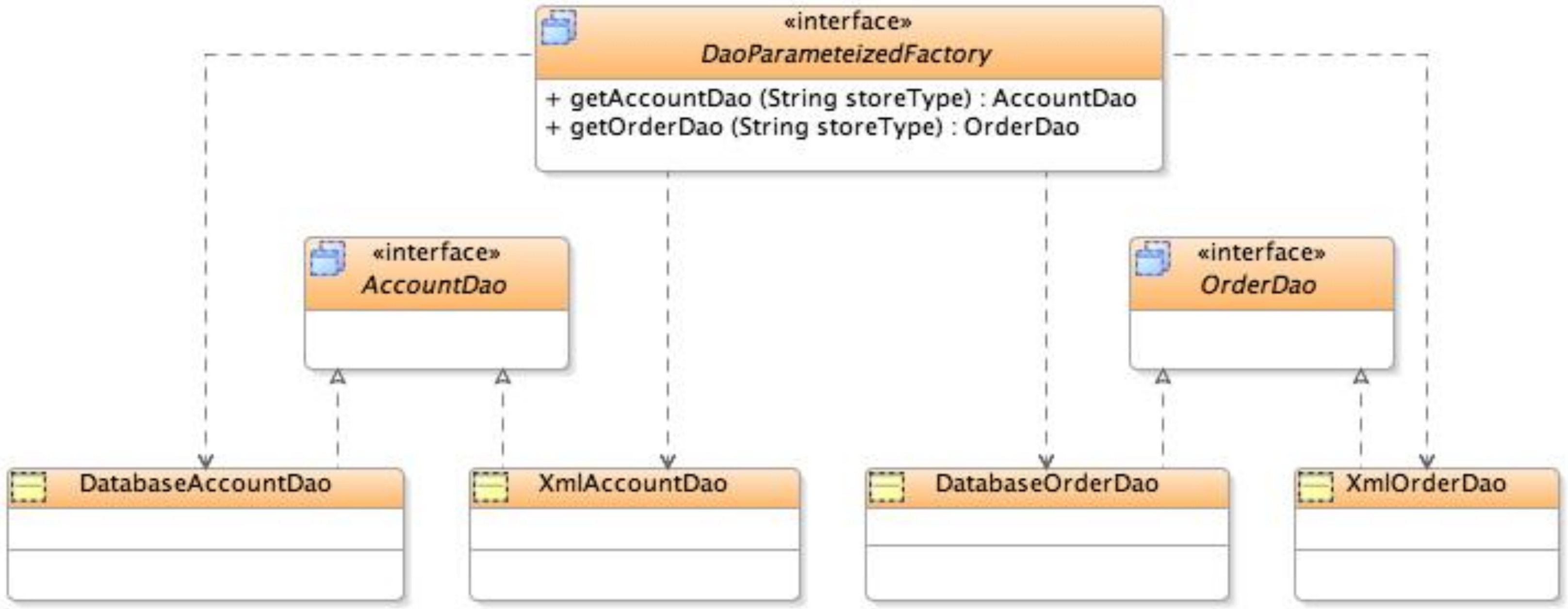
DAOFactory (Factory Method)



DAO Parameterized Factory

- Factory implementation is provided
- Interface and default implementations defined for product
- User may select implementation of product the factory creates

DAO Parameterized Factory



Abstract Factory Problem

- Family of classes work together
  - Need to replace the entire family as a set
- Consider:
  - A word processor having spelling and grammar checkers
  - Must replace spelling and grammar checker classes to support different languages
  - Want to support many languages

Abstract Factory Solution

- Another factory pattern at a higher level of abstraction than Factory Method
  - A collection of factories
- Specifies an interface or abstract class defining operations for creating families of related objects
  - Operations create abstract types
  - Parallel set of classes corresponding to those specified by the create operations
- Subclasses of the factory create a specific family of objects

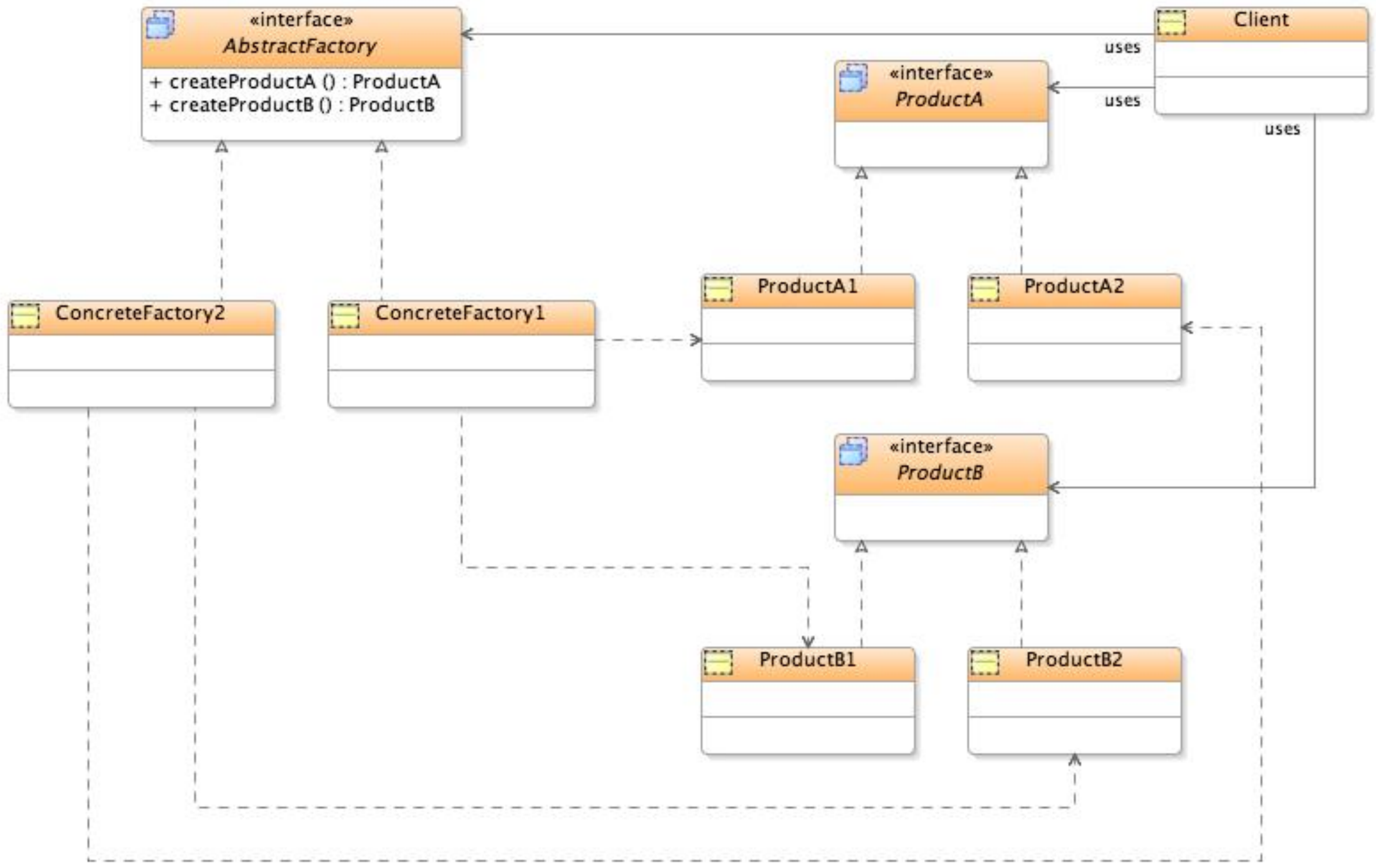
Abstract Factory Solution

- Defining Characteristics
  - A family of interfaces identifies all the abstract types to be used by the client
  - Abstract Factory interface specifies operations for creating instances of the abstract types in the family
  - Concrete classes realize the families abstract types
  - Concrete factory realizes the operations of the abstract factory interface, instantiating concrete implementations of the family

Abstract Factory Solution

- Discussion
  - Want the factory to create additional abstract types
    - Define the new `Product` interface
    - Add an operation to the `AbstractFactory` interface
    - Define a concrete subclass of new product
    - Add new create product method to the concrete factories
  - Client class is unaffected by the change

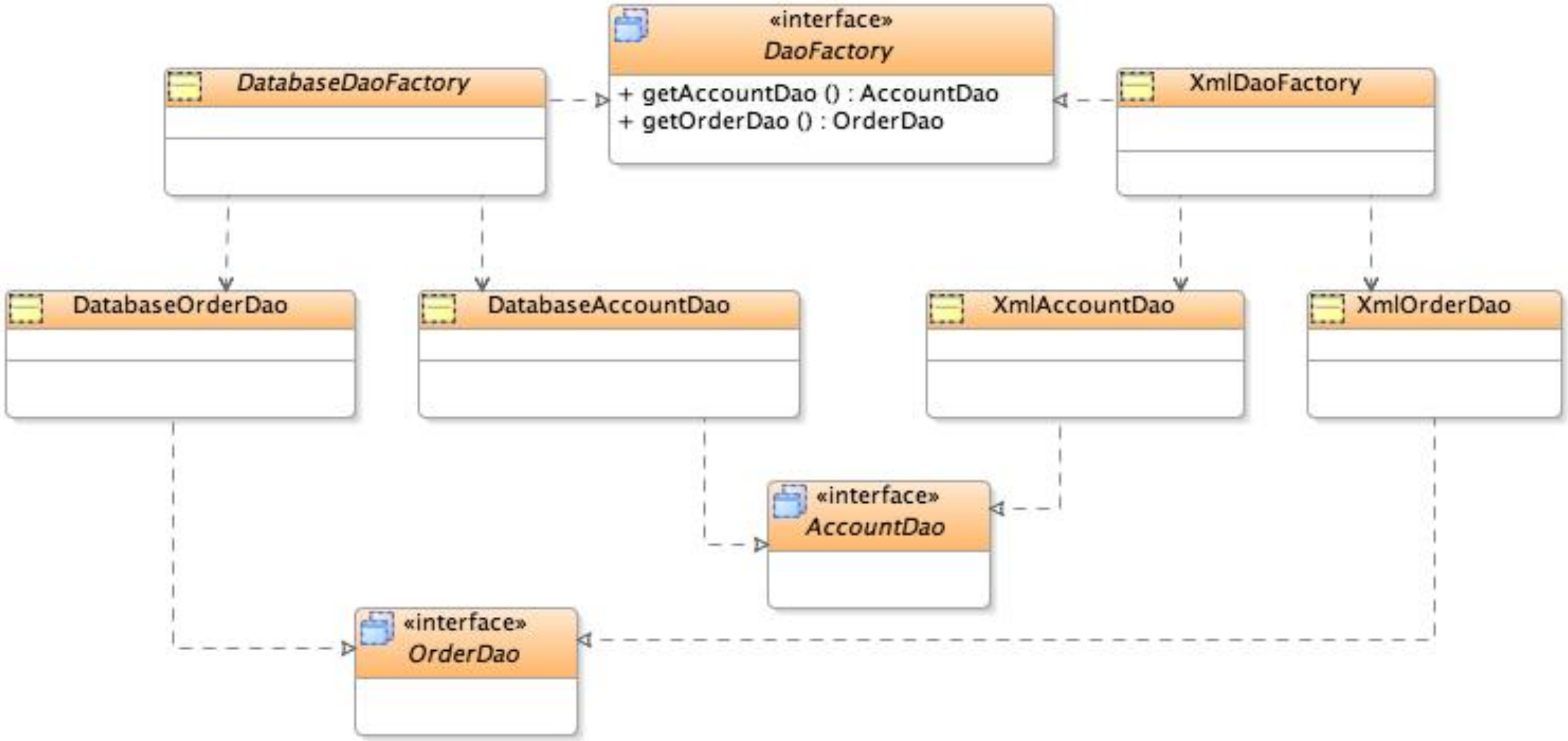
Abstract Factory



Abstract Factory Consequences

- Restrict the client code to using only the calls of the abstract base classes
- Implementing whole new families of product is simplified
  - Implement the new classes in the hierarchy
  - Derive a new factory
  - Install it
- Adding new abstract types to be created by the factory is difficult
  - All factories implementing the abstract factory interface require change

DAOFactory (AbstractFactory)



Time for a break…

JNDI

- Java Naming and Directory Interface
  - Standard interface to naming/directory services
    - LDAP
    - COS Naming
    - RMI Registry
    - NIS
    - DNS
    - File System
    - Windows Registry

Context

- **Principle interface**
  - `void bind(String name, Object obj)`
  - `Object lookup(String name)`
  - `void close()`

`InitialContext`

- Implementation of `Context`, provides a “bootstrap” context
- Initialization mechanisms
  - `Hashtable`
  - Properties file

Context Properties

- Constants
  - `INITIAL_CONTEXT_FACTORY`
    - `java.naming.factory.initial`
    - Name of the `InitialContext` factory class
  - `PROVIDER_URL`
    - `java.naming.provider.url`
      - Location of configuration information for the service provider
  - Others…

Initializing Context

- Using Hashtable

```
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY, "naming.factory.class.name");
  ht.put( Context.PROVIDER_URL, "provider.config.file.url" );
  Context ctx = new InitialContext( ht );
```

- Using jndi.properties

jndi.properties file

```
java.naming.factory.initial= naming.factory.class.name
java.naming.provider.url= provider.config.file.url
...
```

Source code

```
   Context ctx = new InitialContext();
```

Context Initialization Example

```
Hashtable ht = new Hashtable();
ht.put( Context.INITIAL_CONTEXT_FACTORY,
    "edu.washington.ext.cp130.naming.LocalInMemoryContextFactory" );
ht.put( Context.PROVIDER_URL, "namespace.xml" );
Context ctx = new InitialContext(ht);
```

Recall - Loading JDBC Drivers

- Three approaches:
  1. Load driver class explicitly
     - For quick and dirty development
     - Least desirable
  2. Identify driver classes using system properties
     - When naming service isn't available
  3. Using `DataSource` class and JNDI
     - Preferred method

JNDI JDBC Namespace

- JDBC namespaces
  - J2EE
    - `java:comp/env/jdbc/`
  - Otherwise
    - `jdbc/`

DataSource

- Basic implementation
  - Standard Connection object
- Connection pooling implementation
  - Connection object participates in connection pooling
- Distributed transaction implementation
  - Connection object that may be used for distributed transactions

Connecting

- Two methods
  - `Connection getConnection()`
    - Username and password configured in provider
  - `Connection getConnection(String user,`
    `                                String pass)`
    - Username and password provided by application

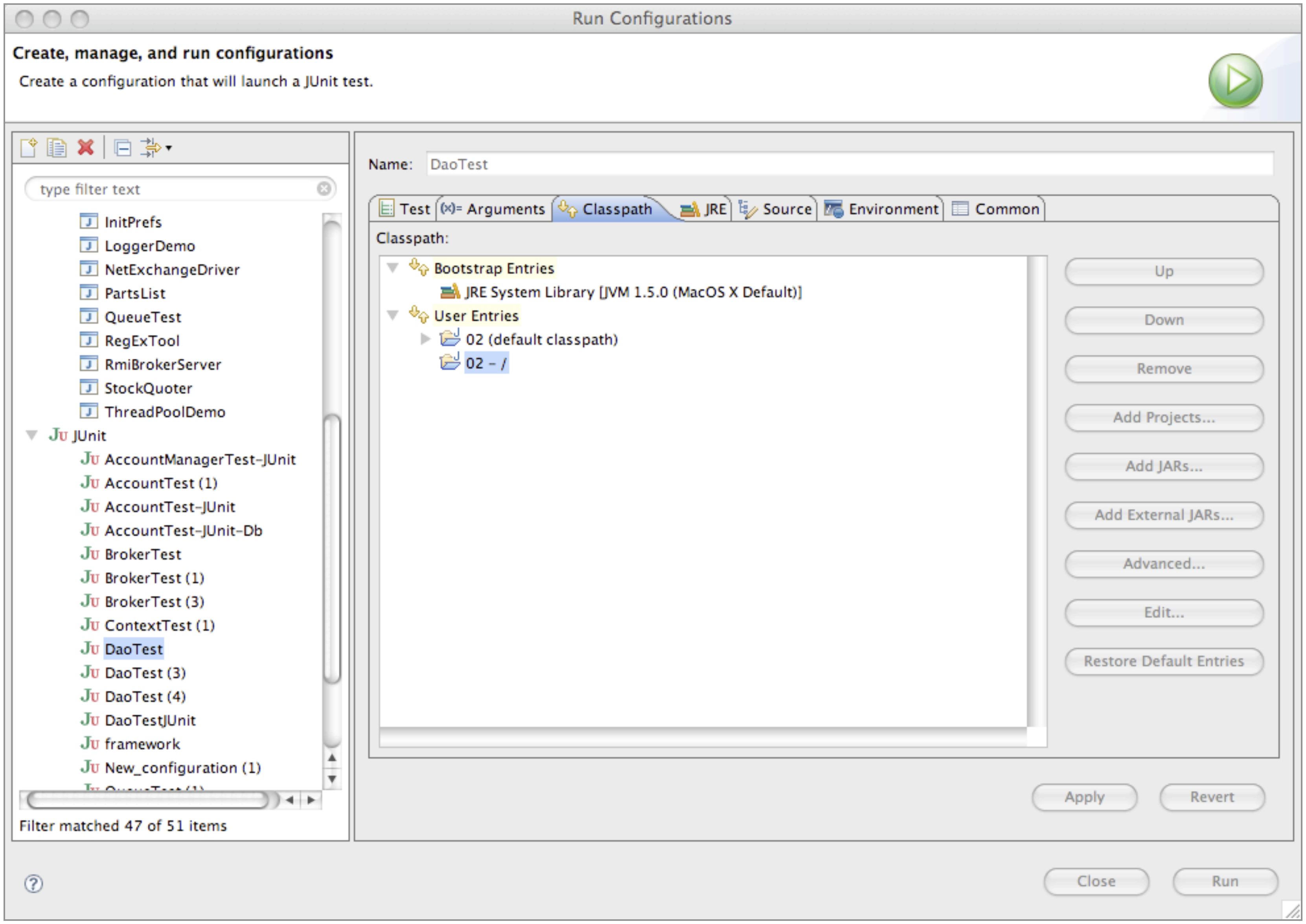
Connecting Using JNDI

- Configure `DataSource` in JNDI, then
  - Uses JNDI and `javax.sql.DataSource`

```
Connection conn = null;
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup( "jdbc/"+"mySrc" );
ctx.close();
try {
   conn = ds.getConnection();
} catch( SQLException ex ) {
   ...
}
```

Using `jndi.properties`

- Need to add the project directory to the runtime classpath:
  1. Open the “Run Configurations” dialog
  2. Select the desired run configuration
  3. Select the “Classpath” tab
  4. Select “User Entries”
  5. Click the “Advanced…” button
  6. Select “Add Folders” radio button and click the “OK” button
  7. Select the desired project folder and click the “OK” button



Run Configurations

Create, manage, and run configurations

Create a configuration that will launch a JUnit test.

type filter text

InitPrefs
LoggerDemo
NetExchangeDriver
PartsList
QueueTest
RegExTool
RmiBrokerServer
StockQuoter
ThreadPoolDemo
JUnit
AccountManagerTest-JUnit
AccountTest (1)
AccountTest-JUnit
AccountTest-JUnit-Db
BrokerTest
BrokerTest (1)
BrokerTest (3)
ContextTest (1)
DaoTest
DaoTest (3)
DaoTest (4)
DaoTestJUnit
framework
New_configuration (1)

Filter matched 47 of 51 items

Name: DaoTest

Test | Arguments | Classpath | JRE | Source | Environment | Common

Classpath:

```
Bootstrap Entries
    JRE System Library [JVM 1.5.0 (MacOS X Default)]
User Entries
    02 (default classpath)
    02 - /
```

Message Digest

java.security.MessageDigest

- A MessageDigest is a hash algorithm
- If contents are altered digest will change
  - MD5 from MIT (16 bytes)
  - SHA-1 from NIST (20 bytes)
- Hash algorithms are
  - Collision-resistant
  - One-way functions

Using MessageDigest

```
MessageDigest md =
    MessageDigest.getInstance( "SHA1" );
md.update( dataBytes );
byte[] digestBytes = md.digest();
```