

Lecture 16: Factories and Frameworks

Factory Pattern

- One definition:
 - Involves a class that is used to create instances of one or more other classes. e.g. it will contain methods like:
A: getInstanceofA()
which will create and return an instance of type A

Factory Motivation 1

Why not just use the constructor for A?

A is typically an interface or abstract class

- ii) Actual concrete class may not be known until runtime, via a run time property, and will be determined by the getInstanceA() method
- iii) Concrete class known during class reuse
- iv) Except for the factory, the code does not have to know what the concrete class is

Creator Pattern (Review)

- Give class B the responsibility of creating instances of class A if:
 - B aggregates object instances of A
 - B contains objects of class A
 - B records instances of objects of class A
 - B “closely uses” objects of class A
 - B has the initialization data that will be passed to an object of class A when it is created

Factory Motivation 2

- Why not just put the `getInstanceA()` code in the places where the Creator pattern suggests such an object should be created?
 - This code will have the details of what the concrete class looks like or how you get it. Could have poor cohesion since the code is built around using an object with interface/abstract supertype A, and the details do not belong here

Factories vs Singleton

- Singleton pattern: returns the single instance of a class
 - Is this an example of a factory? Sort of, but not really
 - the class of the returned instance is known in the code requesting it as a concrete class, not an interface or abstract class
 - the class of the returned instance is the same class as the place where the getInstance() method is located, it is not an instance of a different class

DS Factory Example

- Involves the use of a proxy class for the DataBase subsystem interface
- First a review of some Java features

Java Review – Properties Objects

- A Property object holds a set of string pairs where the first is a key and the second is a property associated with that key. (Subclass of Hashtable class.)
- A.getProperty(s) will return the string that is associated with the key s in the Properties object A
- A.setProperty(s, t) will set create a property s with value t in the Property object A

Java Review – System Class

- the System class
 - special class with static methods
 - e.g. `System.out` will return the standard output device, as used in `System.out.println(x)` to print out the string value of `x`
 - `System.getProperties()` will return a properties object that contains configuration information.
 - `System.getProperty(s)` will return a particular system property whose string name is `s`

Java Review – Changing System Properties

```
Properties sysProps=System.getProperties();  
sysProps.setProperty(xxx,yyy);
```

Java Review – Class Object

- For every class A in an application, the JRE provides an immutable object of type Class with information about A. Suppose that the fully qualified name for A is App.A.
- Getting the Class object for A. Suppose x is an instance of A. Three alternatives:
 - i) `Class class = x.getClass();`
 - ii) `Class class = A.class`
 - iii) `Class class = Class.forName("App.A")` //will also load class A

Java Review – Creating An Object from its Class Object

- Suppose `x` is an instance of the Class `class`, containing information about a class `A`
- `x.newInstance()` creates an object of the type `Object`, having the properties of class `A`. Suppose `IA` is an interface for `A`.
- (IA) `x.newInstance` will cast the new object of type `Object` as an object of type `IA`.

DS Factory Example

- We will assume the use of a DataBase proxy class that interfaces with a vendor supplied data base product
- Suppose that there are several different choices of data bases, so we want to create an instance of the correct proxy class at run time.
- The rest of the system just has to know the interface IDataBase for the DataBase proxy object

DS Object Factory Strategy

- Assume that the System properties object contains the fully qualified name of the DB Proxy class to be used
- Have a DBFactory Class with an IDBProxy variable, and a getProxy() method that returns the value of the variable if it is not null, or if it is null creates an instance of the current DB Proxy class
- Use the Class.forName method to load the desired DBProxy class and create an instance of its class object

DBProxy Factory

Class DBProxyFactory

```
{  
IDBProxy dbProxy;  
    ...  
IDBProxy getDBProxy()  
{  
    if dbProxy ==null  
    {  
        String className = System.getProperty("dbProxy");  
        dbProxy = (IDBProxy) Class.forName(className).newInstance();  
    }  
    return dbProxy;  
}  
}
```

Type 1 Factory

- Problem: Want to have a mechanism for creating an instance of a set of related objects without specifying its concrete classes
- Solution: Construct a method that returns an object that is only characterized by an interface or parent class, not the actual concrete class that is returned
- Above DS example is a type 1 factory

Type 2 Factory

- Problem: have a method `getInstance()` in class A that returns an instance of a class B. Want subclasses of A to be able to return different kinds of objects.
- Solution: Subtype class A with classes that have methods that override `getInstance()` and return the desired types of objects. The new types should be subclasses of B.
- In the following the DSFramework is a type 2 factory

Code Re-use - Frameworks

- Functional programming re-use: subroutines
 - re-usable low level layers, high-level structure and flow added by user programmer, *used by* higher levels
- O/O programming re-use: classes
 - re-usable low level layers, high-level structure and flow added by user programmer, *used by* higher levels
- O/O re-use: frameworks
 - re-usable high-level layers containing “major flow”, details are added by user programmer by subclassing framework classes, *uses* lower levels

Framework Examples - AWT

- Java AWT contains classes for frame, button, text boxes etc.
- To build a GUI you
 - subclass and define new constructors
 - implement required event handling routines that are specified in interfaces
- Does not quite fit the idea of a framework's containing the major flow, since you also implement the presentation logic

DS DB Review

- DB constructor is passed a file name
- File is assumed to be in a predetermined format, and to contain MemberData records
- DB reads in the records and stores them in a MemberData vector
- All data base operations (isMember(name), getMemberData(name), getNext(), etc.) are performed on this vector
- When system terminates, the vector is written back out to the file

DB ReUse Opportunity

- Basic idea is quite general
 - read in a set of records and store them in a vector. Write back out on termination.
 - access records sequentially or using a key
 - the only problem specific details are the contents of the record

DB Framework Strategy – Hiding the Details

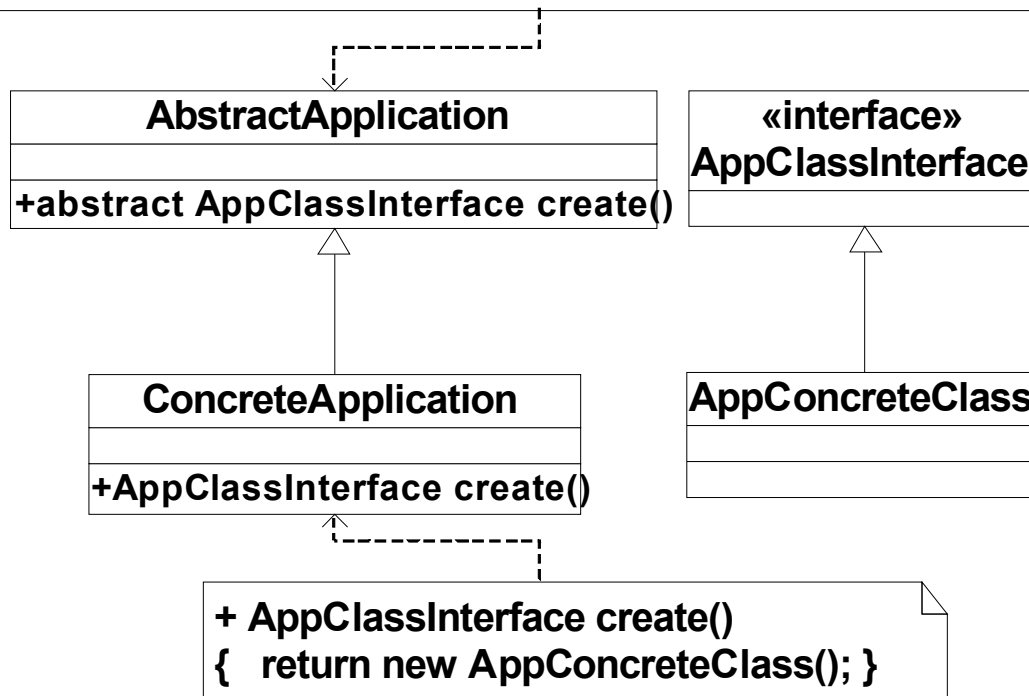
- Reading and writing records from the file into the DB vector will require knowledge of the details of the record
 - use expert pattern, object animation and tell the records to read and write themselves to the file. They know what they look like.

DB Framework Strategy – Overall Structure

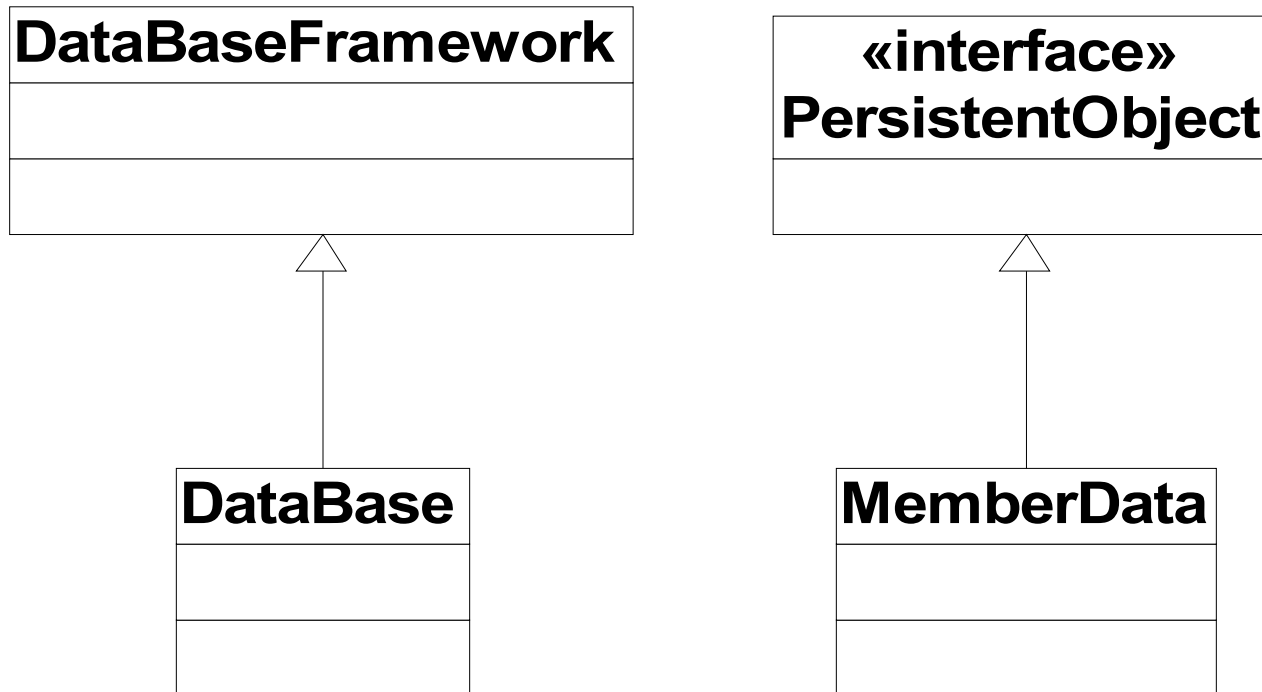
- DataBaseFramework is a data base subsystem, complete except for some details
- Framework abstractions
 - DataBaseFramework, an abstract class, containing:
 - createPersistentObject(), an abstract method
 - PersistentObject, an Interface, used to specify the type of object returned by createPersistentObject()
- User will subclass DataBaseFramework and PersistentObject

Factory Based Framework Pattern

Code for methods in **AbstractApplication** written using **abstract create()** method as though it returned a concrete object, satisfying the **AppClassInterface**



Framework Subclasses for DS



DataBase Framework Class

```
class DataBaseFramework
```

```
PersistentObject [ ] persistentObjects
```

```
int numberObjects
```

```
int maxObjects = 1500
```

```
int accessCounter
```

```
File objectDataFile
```

```
public DataBaseFramework(File file) // reads in file and stores records in PersistentObjects[]
```

```
public void closeDB() // writes objects in PersistentObjects[] back out to file
```

```
public PersistentObject getFirstObject()
```

```
public PersistentObject getNextObject()
```

```
public boolean update(PersistentObject pObj)
```

```
public boolean belongs(String key)
```

```
public persistentObject getObject(String key)
```

```
public boolean add(PersistentObject pObj)
```

```
public boolean delete(String name)
```

```
public abstract PersistentObject createPersistentObject() // to be defined
```

DataBaseFramework Notes 1

- The constructor in the framework has the logic to read in the instances of PersistentObject from the file and store them in a vector. It uses create() to make the objects and expects the objects to know how to read themselves with their read() method
- PersistentObject is an interface, and in the framework, create() is abstract. The framework will be subclassed and a definition for create() will be given that constructs instances of a concrete class that implements PersistentObject.

DataBaseFramework Notes 2

- In the subclass for the framework, the database methods that return records from the database will be refined so that the object that is returned is cast to the concrete implementation of PersistentObject

DataBase Implementation of DataBase Framework

```
public class DataBase extends DataBaseFramework
```

```
public DataBase(File file){super(file)}  
public PersistentObject createPersistentObject()  
    {return new MemberData();}  
public MemberData getMemberData(String name)  
    { return (MemberData) getObject(name); }  
public MemberData getFirst()  
    { return (MemberData) getFirstObject();}  
public MemberData getNext()  
    { return (MemberData) getNextObject();}
```

PersistentObject Interface

```
public interface PersistentObject
```

```
public void read(File file) throws IOException
```

```
public void write(File file) throws IOException
```

```
public String key()
```

MemberData Implementation of PersistentObject

```
public class MemberData implements PersistentObject
```

```
public String name
```

```
public DateeData dateeData
```

```
public AdminData adminData
```

```
public MemberData()
```

```
public String key() {return name;}
```

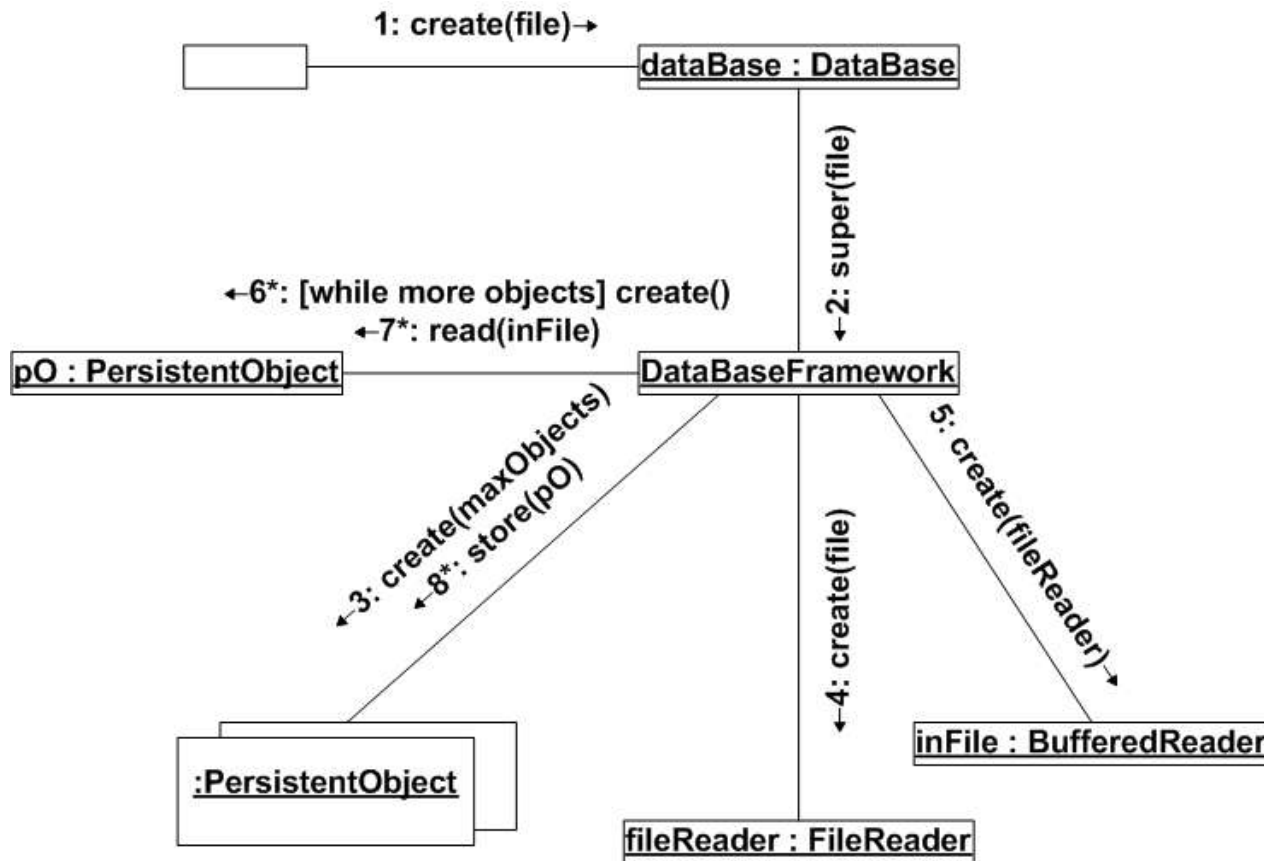
```
public void read(File file) throws IOException
```

```
public void write(File file) throws IOException
```

Technical Underpinnings

- Cannot create an instance of an abstract class or interface
- Can declare a variable of type interface or abstract class
 - of course any value of this variable will have to have a concrete type, but it will need to be a subtype of the declared type

Sample DataBase Method Design – DataBase Constructor



DS Framework and Type 2 Factories

- Type 1 contains a create() method that is declared to return instances of an interface or abstract class, but actually returns instances of a concrete class. Why? class is unknown until run time, and the rest of the code does not care
- Type 2 contains a create() method in a class that is abstract, and will be defined when the class is subclassed. This will coincide with the subclassing of the interface or subclass the abstract create() is declared to return. Why? So we can re-use the code in different applications which will have a different concrete subclass

DS Framework and Template Classes

- Template classes – have a parameter for which a class name will be substituted when the template class is used. Preprocessor macro substitution.
- E.g.

```
class GeneralDataBase<T>
{
    <T> [ ] persistentObjects
    <T> persistentObject
    ....
    // reading in objects from file
    persistentObject = new <T>;
    persistentObject.read(file);
    ...
}
```

Template Classes and Factories

- Type 1
 - may not know class until run time so is not applicable
- Type 2
 - What if we substitute a class T that does not have the required interface?
 - e.g. it has no read method
 - Does not allow us to model the idea that all classes that should be used implement an abstract concept that is an important aspect of the domain