ArgusConnect Pty Ltd
ABN 63107558387
Suite 9, GreenHill Enterprise Centre
University Drive, Ballarat Victoria 3353
Tel: 0415 645 291
Email: andrew.s@argusconnect.com.au
Website: http://www.argusconnect.com.au/

JAF Developers Handbook

Version: **4**

Table Of Contents

1 Introduction 3

1.1 Why do we need JAF? 3
1.2 Where did the idea for JAF originate? 3
1.3 So, what is it? 3
1.4 Version 4 upgrade 4
1.5 Terms and Abbreviations 4
1.6 References 5

2 Program Architecture 6

2.1 Overview 6
2.2 Modularity and decoupling 6
2.3 The Application Domain 8
2.4 The Factory Domain 8
2.4.1 Registering a class 8
2.4.2 Retrieving a class 9
2.5 The User Interface Domain (UI-Domain) 10
2.5.1 Class JPanelDirector 11
2.5.2 Class JPanelAbstract 11
2.5.3 Adding new classes to the UI-Domain 12
2.6 The Problem Domain (P-Domain) 13
2.6.1 Class JArguments 14
2.6.2 Class JStoredObject 14
2.6.3 Adding a new JStoredObject to the P-Domain 16
2.6.4 Class JStoredCollection 19
2.6.5 Adding a new JStoredCollection to the P-Domain 20
2.6.6 Class JService 23
2.6.7 Adding a new JService to the P-Domain 24
2.6.8 Executing Services on a separate thread 24
2.6.9 Class JAssertion 26
2.6.10 Class JCommandExec 26
2.6.11 Implementation of the Observer Pattern 27
2.7 The Data Management Domain (DM-Domain) 28
2.7.1 The Data Management Layer 28
2.7.2 The Database Implementation Layer 29
2.7.3 Class JConnections 29
2.7.4 Class JDatabase 30

3 Generating an execution trace 30

3.1 To activate tracing from the command line 30
3.2 To see all JTracer messages 30
3.3 To see JTracer messages from a specific domain 30
3.4 To see JTracer messages from a specific class 31

1 Introduction

1.1 Why do we need JAF?

JAF is our "**J**ava **A**pplication **F**ramework". It encourages all our Java development to follow a specific development methodology.

The adoption of a particular methodology is critical to the success of any large-scale development projects. This applies especially to situations where there are many developers working together to produce a system.

In any programming team, it is inevitable that the programmers will each have their own preferred approach to design and coding style. Without an adequately defined methodology, each developer is given the freedom to adopt their own methods, which may be variable, inappropriate, and not easily understood by others. Such inconsistencies hinder productivity where each member of the team needs to be easily able to pick up and maintain the work of others.

Where a methodology is clearly defined, projects can proceed with a direction and a focus, with all team members working the same way. Having a clearly defined methodology also facilitates better communication, giving team members a 'common 'language' with which to discuss design issues. JAF has its own lexicon, and has been documented using UML.

Like all methodologies, JAF must be expected to change and evolve. A changing methodology will also provide the opportunities for a development team to evolve. Without a methodology, there is no defined expectation or requirement of developers to meet a standard, and therefore no facility or direction for individual self-improvement.

1.2 Where did the idea for JAF originate?

It is not strictly true for me (Andrew Shrosbree) to claim that I "designed" JAF. I merely built it. Working for 8 years as a programmer I found that the brightest programmers I met all had something in common: they understood the need to build an abstract object framework upon which all other objects in a system should be based.

In 2000, I worked in Geelong with a software architect called Conal McClure. He had done more research into framework design than I, and was trying to encourage our employer to use a framework he had written (in Delphi). Within that company, his wisdom fell like pearls before swine, but I took his ideas to heart. We both left the company, and I pursued a greater understanding of patterns by stripping down Conal's code and by reading the book 'Design Patterns' by Gamma, Helm, et al (better known on the Internet as the "GoF"; Gang of Four).

Upon joining the CCeH in 2001, I built JAF with a strict eye on the patterns outlined by the GoF. As such, JAF constitutes an implementation of at least 9 of their patterns. In some cases, I was fortunate to be spared the implementation of a pattern, such as the Observer Pattern because it is already part of the Java language.

1.3 So, what is it?

The simplest description of JAF is that it is a library of about 30 java classes. These classes are organised into separate sub-folders (called 'domains'). When writing any Java class, the programmer decides what the basic behaviour is of

that class. The new class is subclassed from a JAF class that contains an abstract structure for the new class being created. For example, if the class is required to handle data retrieval, the subclass will implement a *JDataClass* because *JDataClass* already contains behaviour common to **all** classes that handle data retrieval.

What makes JAF important is that it represents the 'glue' that holds a system together. It contains a number of classes that perform two essential tasks:

1. JAF implements certain paths of execution (flows)
2. JAF allows certain classes to communicate with (subclasses of) each other.

As an example for 1), if a class needs to perform a SELECT on the database, the actual process does not need to be defined by a programmer. Instead, the programmer merely subclasses a *JDataClass*, overriding 3 or 4 methods which tell the framework the information it needs to perform a SELECT (such as the primary key and the table name). JAF will construct a SELECT statement and actually read the data, by calling the *readData()* method. The programmer does not need to implement the *readData()* method– it has been already defined in JAF.

I must stress that JAF will always be work in progress. Version 1 took four weeks to build; version 2 was completed three months later and version 3, yet another 4 months later. The need for enhancements became apparent as JAF was used to build 3 commercial systems.

1.4 Version 4 upgrade

By the end of 2004 JAF formed the foundation of 4 commercial systems. A major weakness was the need to accommodate the use of JDBC for Java to communicate with databases such as Interbase. Another weakness was JAF's usage of Borland data classes that were proprietary in nature.

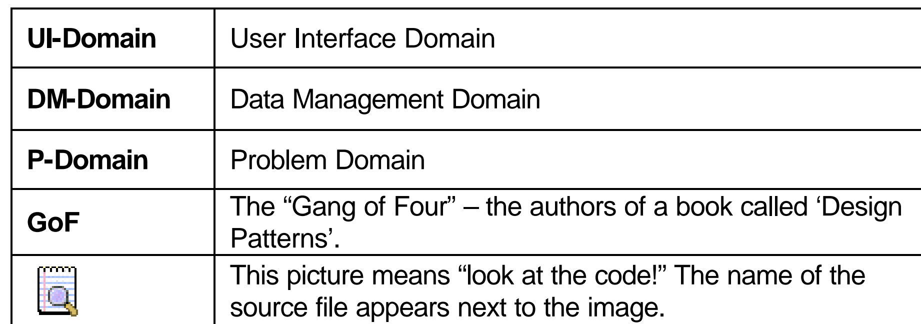
Advances in open source development in 2003/2004 made it possible for Java programs to communicate directly with databases, thereby eliminating the need for JDBC. The ***Jaybird*** project provides a number of classes that allow such native calls to the database.

I engaged in a huge upgrade of JAF's data management domain to allow the use of Jaybird classes. These changes simplified every subclass of *JDataClass* while also reducing the code in every *JDataClass* by a third.

Further changes were made by Phillip Cameron to eliminate JAF's dependency on Borland data classes, and to reduce dependencies that would make it difficult for JAF to be truly database independent.

1.5 Terms and Abbreviations

Term	Definition
JAF	Java Application Framework
UML	Unified Modelling Language – a notation used for the design and documentation of processes and workflow.
CCeH	**C**ollaborative Centre for **eH**ealth



UI-Domain	User Interface Domain
DM-Domain	Data Management Domain
P-Domain	Problem Domain
GoF	The “Gang of Four” – the authors of a book called ‘Design Patterns’.
	This picture means “look at the code!” The name of the source file appears next to the image.

1.6 References

“Design Patterns” – Gamma, Helm, Johnson, Vlissides (The “GoF”).
This document should be read with reference to “**Class Heirarchy.pdf**”. The dark grey classes in this diagram are the classes that are subclassed when JAF is used to build a system. Programmers will never subclass the other classes.

2 Program Architecture

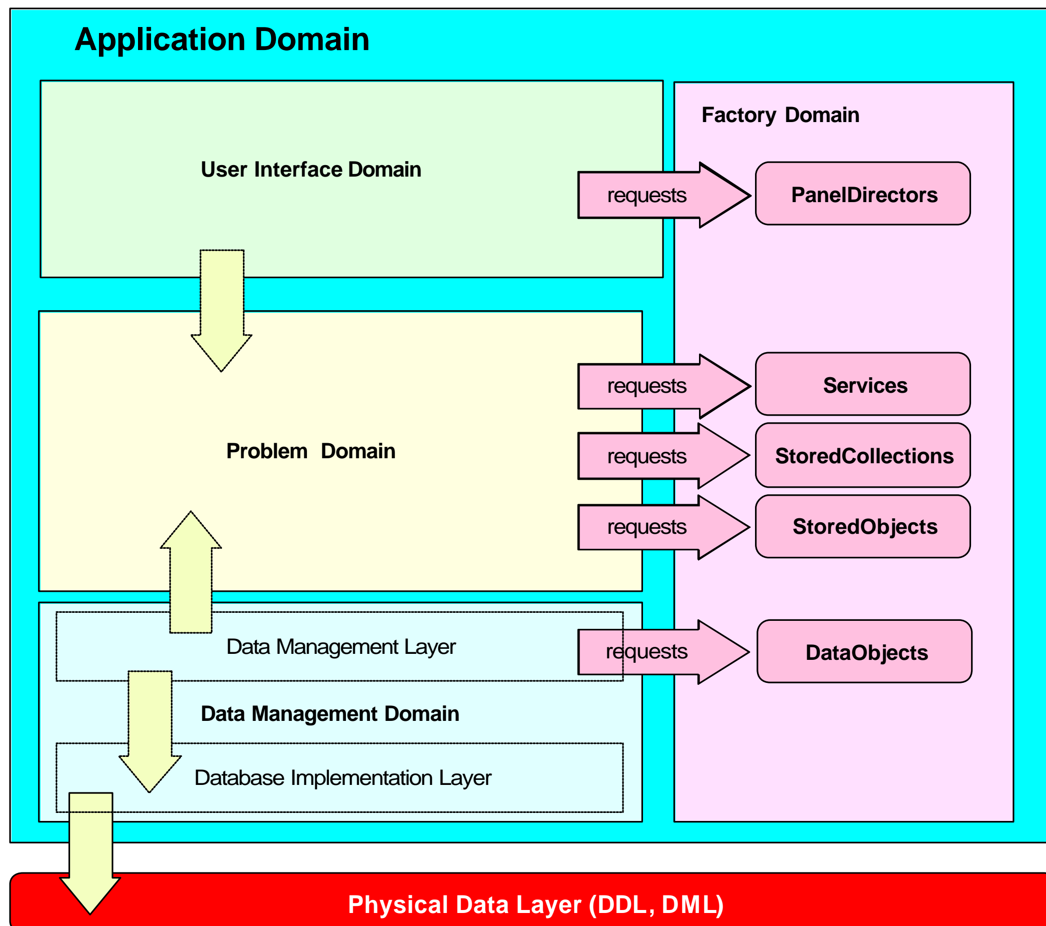


Figure 1: High Level Conceptual Architecture

2.1 Overview

The architecture adopted in JAF's methodology is based on a layered, object domain model. This model focuses heavily on the relationship between object responsibilities, and organises classes into clearly defined layers or domains. Each domain is represented by the classes in a single package.

A domain is best defined as a layer within the object space of the application. In other words, a domain contains code and business logic that is specific to the application.

There are 4 major object domains that make up this model, the Factory Domain, the User Interface Domain, the Problem Domain and the Data Management Domain.

2.2 Modularity and decoupling

The primary premise of JAF is that by using factories, any application based on JAF is totally decoupled. This means that when a particular class is required to do some work, its definition is resolved at run time. This allows an application to be compiled even before all the classes it references have been written. In theory, since there is no communication between User Interface Domain (UI-Domain) classes and Data Management Domain (DM-Domain) classes, it should be

possible to totally replace all classes in one of these layers. For example, a system may need to be converted to being web-based, rather than GUI-based. All screens in the UI-Domain can be replaced with scripted pages, without impacting the Problem Domain (P-Domain) or the DM-Domain.

2.3 The Application Domain

IApplicationDomain.java

This is an abstract domain without any classes. It exists merely to create a logical 'starting point' in the definition of JAF's class hierarchy. The 'domain' is represented by the (empty) interface *IApplicationDomain.* All other domains extend this interface for the purpose of creating an ordered heirarchy.

2.4 The Factory Domain

IFactoryDomain.java

Refer to GoF for a description of the factory and abstract factory patterns. The factory concept helps to de-couple the relationship between two objects by using a mapping association (similar to a hash-key), rather than a direct knowledge relationship.

This domain contains *JRegisterClasses*, *JController* and five factories. *JController* is the most important class in JAF. Each factory maintains a repository of subclasses (of the same base class) in a hash table. Every class in the repository has a friendly name, or "moniker". For example, *class JPanelDirectorArchive* would simply be referred to as "Archive".

JController.java

JController is a singleton (refer to GoF for a description of the singleton pattern) that controls the storage and retrieval of classes in 5 different factories.

Note: The programmer never directly references any factory - *JController* handles the details of class storage and retrieval.

Maintaining domain integrity imposes the following rules on class usage:

- All classes must be retrieved via *JController*, not independently instantiated.
- To enforce this, all subclasses of JAF must have private constructors.
- Classes in the same domain may reference each other.
- UI-Domain classes may reference classes from the P-Domain.
- UI-Domain classes may **not** reference classes from the DM-Domain.
- DM-Domain classes may reference classes from the P-Domain.
- DM-Domain classes may **not** reference classes from the UI-Domain.

2.4.1 Registering a class

Every class in an application must be registered by means of a call to *JController.*

For example, for a Stored Object called "Person":

```
JController.registerStoredObject("Person", "_ProblemDomain.JStoredObjectPerson");
```

In theory, classes can be registered from anywhere in an application. When a class is requested by a call to *JController,* it is important to ensure that the class has already been registered in a factory. The registration of classes should be done in one place, by a subclass of *JRegisterClasses.*

For example, a program called ArgusMail that uses JAF will have a class "*JRegisterClassesArgusMail extends JRegisterClasses".*

JRegisterClassesArgusMail overrides the four methods registerServices, registerStoredObjects, registerStoredCollections, registerPanelDirectors.

In the main class of ArgusMail, all four of the registration methods can be called via a single call to registerAll():

```
JRegisterClassesArgusMail oRegister = new JRegisterClassesArgusMail();

oRegister.registerAll();
```

2.4.2 Retrieving a class

Whenever an application needs a class it asks *JController* for the class, by the moniker under which the class was registered (a string). *JController* resolves the name and retrieves the class.

For example, if a program needs the class "JStoredObjectPerson", the correct call is:

```
JStoredObjectPerson oPerson;

oPerson = (JStoredObjectPerson)JController.getStoredObject("Person");
```

Each request made of JController is directed to the relevant factory. For instance, when a request is made for a panel, JController defers the request to the panel director factory JFactoryPanelDirector .

JFactoryPanelDirector.java -> getPanelByName()

JFactoryPanelDirector locates the requested class, returning an instance of it to JController.

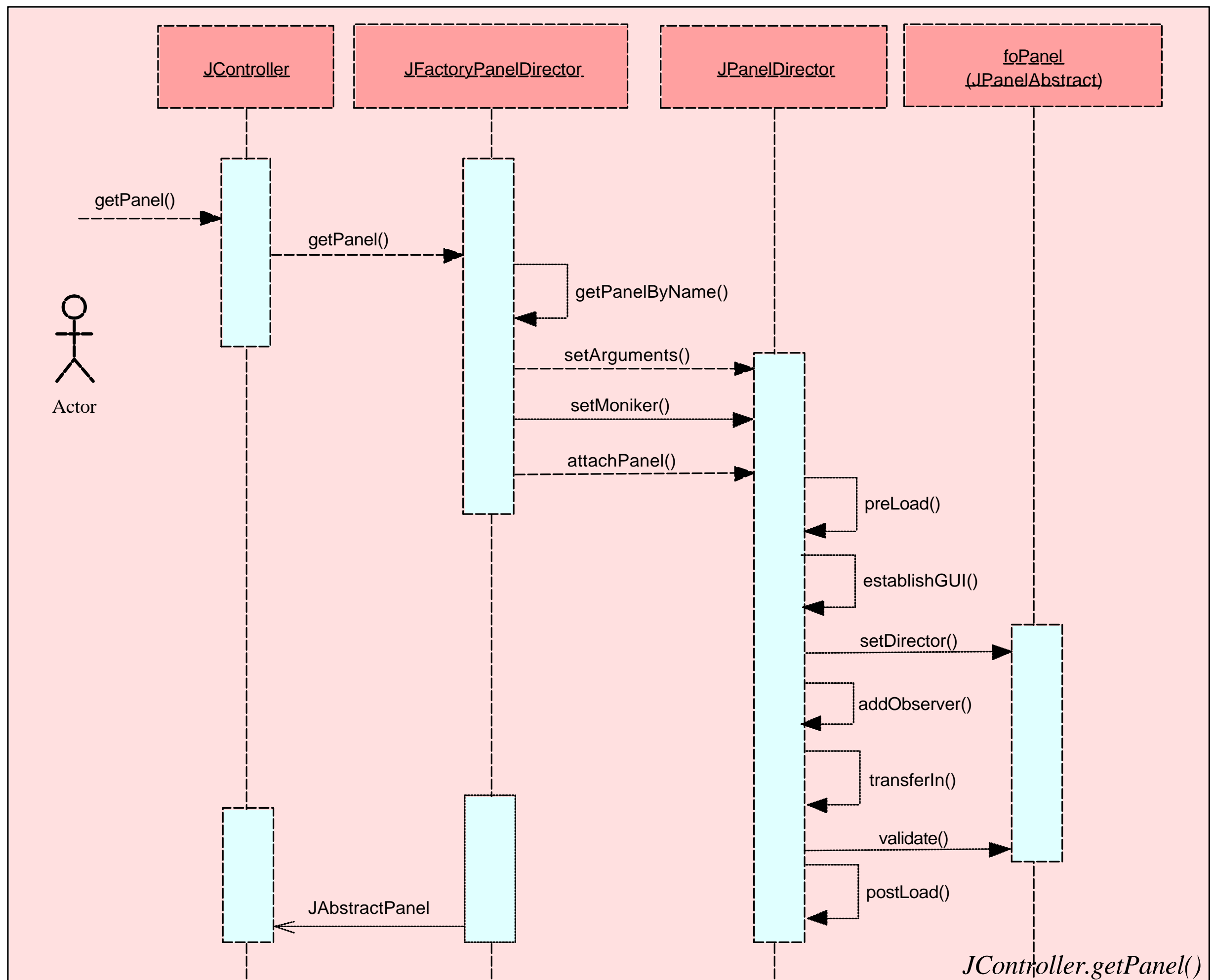


Figure 2: an example of how JController responds to a request for a class

2.5 The User Interface Domain (UI-Domain)

From an application design perspective, the user interface components are responsible for launching windows, dialogs, responding to menu clicks etc. At another level, it is the responsibility of the user interface to manipulate the various objects in the problem domain, to provide the desired functionality. For example, suppose a screen has been designed to initiate some calculations. The user interface is responsible for collecting the input values from the user, and instantiating the appropriate business object that will service the calculation. The user interface passes the values to the business object and extracts the results, displaying them back on a form.

IUserInterfaceDomain.java

The UI-domain contains all the classes that make up the user interface of the application. The user interface has knowledge about the Problem Domain, as indicated by the arrow direction joining the two domain regions in figure 1, the High Level Conceptual Architecture shown above.

All subclasses in the user interface domain should have no ability to process business logic, but focus on implementing a particular interface within a specific environment.

The user interface has knowledge about the various business objects available. This can potentially lead to GUI's becoming more intelligent than we would ideally like them to be, as they fire up a number of business objects and control the interactions between them. To mitigate this problem, the user interface model introduces two abstract classes: the Panel (*JPanelAbstract)* and the Director (*JPanelDirector)*, discussed below.

Together, *JPanelAbstract* and *JPanelDirector* constitute a very typical *Model-View Controller.*

2.5.1 Class **JPanelDirector**

JPanelDirector.java

The Director is a non-GUI class which encapsulates as much of the intelligence and decision making services required by a GUI as possible, thereby keeping the GUI classes themselves clean and *dumb*. The Director is designed to work primarily with the Panel class.

The Director is responsible for instantiating the GUI and passing information to it. If the GUI requires the services of a business object, or needs to instantiate another GUI object, this activity should be delegated back to the associated Director. If a Panel needs to perform some non-trivial validation the code should be kept out of the GUI and embedded in the Director instead.

The Director is very much dependent on the GUI and vice-versa; Director classes would be required to support changes to the user interface.

2.5.2 *Class* JPanelAbstract

JPanelAbstract.java

All GUI's that manipulate or work with the Problem Domain should be based on the abstract Panel class *JPanelAbstract*. The Panel is designed to work with the Director class discussed above.

JPanelAbstract has a 'model' property (foPanelDirector) that references a *JPanelDirector*. The model is the Director that is responsible for this form. The model property is automatically set by the Director, when the method *attachPanel()* is called. Take a close look at *attachPanel().* It calls a number of abstract methods, many of which must be defined in the subclass.

To recap: in the context of our Java applications, every 'screen' is represented by a JPanel subclass called *JPanelAbstract*. Subclasses of this class should contain no decision-making ability – that should be handled by a subclass of *JPanelDirector.* In other words, every 'screen' in a system is represented by:

- *A subclass of JPanelAbstract*
- *A subclass of JPanelDirector*

JPanelLogin.java + JPanelDirectorLogin.java

A good example of this implementation is the classes *JPanelLogin* and *JPanelDirectorLogin*. These two classes are the only subclasses of *JPanelAbstract* and *JPanelDirector* that are part of JAF. Together they constitute an enhanced login screen.

The Problem Domain and the Data Management Domain should have **no** knowledge of the existence of the User Interface classes. It should be possible to redeploy the application on a completely different user interface, or remove the user interface entirely, without disturbing any of the existing Problem Domain classes.

This approach is totally at odds with any kind of RAD (Rapid Application Development) design. JAF's user interface is just a wrapper around the business logic, and has no knowledge of the implementation of business rules or data structures.

2.5.3 Adding new classes to the UI-Domain

When adding a new 'screen' to a project, these are the steps to follow to add the two required subclasses:

1. Create a new class in _UserInterfaceDomain (referred to here as "[the panel]"), by subclassing *JPanelAbstract* from package UserInterfaceDomain.

2. Design the visual layout and components of [the panel]

3. Create a new class in _UserInterfaceDomain (referred to here as "[the director]"), by subclassing *JPanelDirector* from package UserInterfaceDomain. Make the constructor private.

4. To allow [the director] to be instantiated correctly (by a factory), define the *CreateMe()* method as follows:

```
static public Object CreateMe() {

        validateCaller();

        return new [the director]();
};
```

5. In [the director], define the contents of the e*stablishGUI()* method (to link this class to [the panel]):

```
public class [the director] extends JPanelDirector {

  protected void establishGUI() {

    foPanel = new [the panel]();
  }
  …
}
```

6. In [the director], define the contents of the *transferIn()*, *transferOut()* and *process()* methods. Method *process* must return a value, even if it's null.

7. In your application, register [the director]:

```
JController.registerPanelDirector("[a name]", "_UserInterfaceDomain.[the director]");

e.g.

JController.registerPanelDirector("System", "_UserInterfaceDomain.JPanelDirectorSystem")
```

8. Test [the director] by calling

```
JController.retrievePanel( [panel director name] )

or

JController.getPanel( [panel director name] ) for a panel without loaded data.
```

2.6 The Problem Domain (P-Domain)

IProblemDomain.java

The P-Domain encapsulates all of the business logic of the application. It knows nothing about the user interface, or where data is stored, organised or comes from.

The business logic includes business objects and interactions between multiple business objects (processes), to solve the business problems of the application.

Within the P-Domain, there are three major high-level abstract classes, *JService*, *JStoredObject* and *JStoredCollection*.

It is important to understand the purpose of these abstract classes within the context of the architecture, and the different roles that they provide. Both *JStoredObject* and *JStoredCollection* have a common ancestor (*JStored*) because they are both responsible for the storage of application-specific data (such as records from a database table). The fundamental difference between *JStoredObject* and *JStoredCollection* is that whereas *JStoredObject* can be thought of as storing a single item (such as a single record), *JStoredCollection* is a set of the same *JStoredObject* subclasses. Subclasses of these two classes are named accordingly. If for instance, a program has a “Persons” Stored Collection, it will have a matching “Person” Stored Object. The **Persons** object would constitute a collection of **Person** objects.

While a Stored Object can exist without an associated Stored Collection (**Person** without **Persons**), a Stored Collection <u>cannot</u> exist without an associated Stored Object (**Persons** without **Person**).

By contrast, a *JService* subclass is used to perform a specific task. Thus, any flow of decision-making execution must be encapsulated within a *JService* subclass.

2.6.1 *Class* JArguments

JArguments.java

This is one of the most important classes in JAF. *JArguments* stores a collection of objects in a hash table. Each object represents a parameter relevant to the execution of whatever method it is received by. *JArguments* is used frequently to pass a key (*JKey*) to an object, so there are two 'default' arguments (hash table entries) in *JArguments*: "KEY" and "ASSOCIATEDKEY".

Since *JBaseClass* has an arguments field, all its subclasses can access their argument list with calls to *getArguments()* and *setArguments()*. As an example, this creates a new key and adds it to the arguments:

```
JKey oKey = new JKey(234);

getArguments().setKey( oKey );
```

This can be retrieved like this:

```
JKey oKey = new JKey();

oKey = (JKey)getArguments.getKey();
```

For other arguments (not keys), this is the code:

```
JStoredObjectPerson  oPerson;

oPerson = (JStoredObjectPerson)JController.getStoredObject("Person");

oPerson.Name = "Andrew";

getArguments.addArgument( "PERSON", oPerson );
```

This can be retrieved like this:

```
JStoredObjectPerson  oPerson;

oPerson = (JStoredObjectPerson)getArguments().get( "PERSON");
```

Using *JArguments* ensures that you never need to use formal parameters – just bundle them up as arguments in a *JArguments* object and they can be easily retrieved and passed between objects.

2.6.2 *Class* JStoredObject

JStoredObject.java

A Stored Object represents a persistent object within the business model, which is typically a single record in the database. A Stored Object must be uniquely

identifiable in the database, via some kind of unique key. A base class has been set up within the infrastructure called *JKey*, which can store integer and string keys. Every *JStoredObject* has two 'key' fields: the first (foKey) stores its primary key and foAssociatedKey stores a foreign key.

For example, a 'Patient' object may be linked to many 'LabTest' objects. Each 'LabTest' object would store (in its foAssociatedKey) the foKey value of the 'Patient' to which it belongs. This association would make it possible to iterate through a collection of LabTests for a particular patient.

Stored objects exhibit the following foundational behaviour, which is defined in the superclass JStored:

Retrieve … Load the object with field values from the database.
Save … Copy the object's fields to the database.
Delete … Remove the database record which the object represents.

Stored Objects already existing in the database should generally be instantiated by calling *JController.retrieveStoredObject()*,a static method which creates and returns an instance of the Stored Object, loaded with data from the underlying database record.

Stored Objects that have not yet been saved on the database should generally be instantiated by calling *JController.getStoredObject()*,a static method which creates and returns an empty instance of the Stored Object. Once the object's fields have been loaded with data, the *JStoredObject's Save()* method will commit the new objects' contents to the database

For each Stored Object class in the Problem Domain, there should be a corresponding subclass of *JDataClass* in the Data Management Domain. The role of the *JDataClass* is to:

- populate the Stored Object from the database.
- save the Stored Object's fields into a database table.
- delete the Stored Object from the database.

These operations occur at a much lower level than in the *JStoredObject,* usually involving specific SQL calls to the database. More about *JDataClass* later.

2.6.3 *Adding a new* **JStoredObject** *to the P-Domain*

1. Create a new class in _ProblemDomain (referred to here as "[the Stored Object]"), by subclassing *JStoredObject* from package ProblemDomain. Make the constructor private.

2. To allow the object to be instantiated correctly (by a factory), define the *CreateMe()* method as follows:

```
static public Object CreateMe() {

  validateCaller();

  return new [the StoredObject];
};
```

Attach [the Stored Object] to a data object ONLY IF NEEDS TO LOAD ITSELF with data, by adding this call into the (private) constructor, but leave it commented out:

```
private JStoredObject[name]() {

   //JDataObject[name].RegisterMe(this.getClass().getName());
}
```

3. In your application, register [the Stored Object]:

```
JController.registerStoredObject("[a name]", "_ProblemDomain.[the StoredObject]");
```

e.g.

```
JController.registerStoredObject("System", "_ProblemDomain.JStoredObjectSystem");
```

4. Create the application-specific (public) fields that will be encapsulated by [the Stored Object].

Add these imports to the top of the source file:

```
import ProblemDomain.JStoredObject;
import _DataManagementDomain.*;
```

Now, to create the data class (described later in this document) that will be used by this stored Object:

5. Create a new Data Object (referred to here as "[the Data Object]") by subclassing *JDataClass* from package DataManagementDomain. Make the constructor private.

6. In [the Data Object], define the *RegisterMe()* method (to register the class in the factory).

```
static public void RegisterMe( String sClassName ) {

  [the Data Object] oTemp = new [the Data Object]();

  JController.registerDataObject( sClassName, oTemp.getClass() );
};
```

7. In the [the Data Object], define the *CreateMe()* method like this:

```
    static public Object CreateMe() {

      validateCaller();

      return new [the Data Object]();
    }
```

8. Define the behaviour of the r*eadData()* and *saveData()* methods in [the Data Object]. *SaveData()* calls *saveBlobs()* and *setParameterValues*(), so these need to be defined.

e.g.

```
protected void setParameterValues(JStored oStored) {

    JStoredObjectPerson oPerson = (JStoredObjectPerson)oStored;

    if (oPerson.getKey() == null) {
      foCallableStatement.setInt(1, -1 );
    }else{
      foCallableStatement.setInt(1, oPerson.getKey().getAsInt());
    }

    foCallableStatement.setString(2, oPerson.fsName);
    foCallableStatement.setString(3, oPerson.fsEmail_address);
}
```

9. Define the *loadStoredObject()* method in [the Data Object]..

e.g.

```
protected void loadStoredObject(JStored oStored) {

   JStoredObjectPerson oPerson = (JStoredObjectPerson)oStored;

    oPerson.setKey( new JKey( foResultSet.getInt( primaryKeyName() )) );
    oPerson.fsName = foResultSet.getString("Name");
    oPerson.fsEmail_address = foResultSet.getString("Email_address");
  }
```

10. Ensure that these are at the top of the source file:

```
import DataManagementDomain.*;
import ProblemDomain.*;

import _ProblemDomain.*;

import java.util.*;
import java.sql.*;
```

11. Define the behaviour of the *getGeneratorName(), storedProcedure(), storedProcedureReturnParam(), numInsertParameters(), primaryTable(), primaryKeyName()* methods in [the Data Object].

12. Remove the comment from the constructor of [the Stored Object]:

```
 private JStoredObject[name]() {

   [the Data Object].RegisterMe(this.getClass().getName());

 }
```

13. All Stored Objects must be cloneable. Override the *getEmptyInstance()* method, to initialise all the fields in such a way that an empty copy of the object can be constructed.

e.g.

```
 public Object getEmptyInstance() throws CloneNotSupportedException {

JStoredObjectPerson oPerson = (JStoredObjectPerson) super.clone();

   oPerson.fsDeleted = new String("");
   oPerson.fsEmail_address = new String("");
   oPerson.fsLogin_name = new String("");
 }
```

14. Add the method *showItem()* to [the Stored Object] thereby allowing programmers to easily interrogate the object to discern the contents of its fields. The contents of every stored object in a collection can be output by calling *JStoredCollection.showAll().*

e.g.

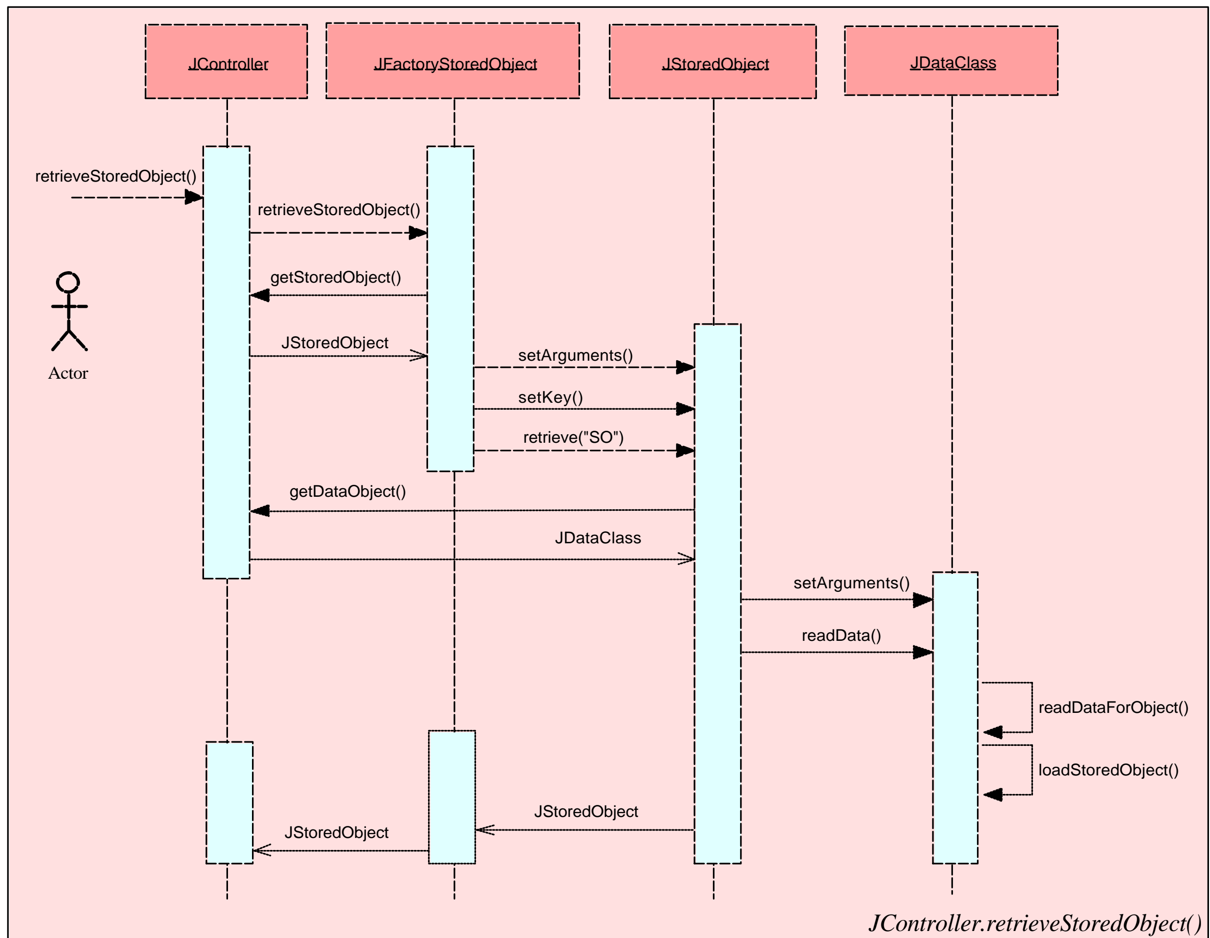
```
 public String showItem() {
   return
     fsName + ", " +
     fsEmail_address;
 }
```

15. Test [the Stored Object] by calling

JController.retrieveStoredObject([Stored Object name] {, params})

or

JController.getStoredObject([Stored Object name]{, params}) for a new Stored Object without loaded data.



2.6.4 *Class* JStoredCollection

JStoredCollection.java

A Stored Collection represents a persistent Collection within the business model, such as the result of a SELECT query. It contains a vector (fvItems) into which Stored Objects are added. *JStoredCollection* is JAF's equivalent of a record set.

Stored Collections exhibit the following foundational behaviour, which is defined in the superclass *JStored*:

Retrieve … Load the vector with Stored Objects, which have been loaded with field values from the database.
SaveAll … For each Stored Object in the vector, copy the field values to the database.
Delete[ByKey|ByIndex] … Remove the database record which a Stored Object represents.

Stored Collections already existing in the database should generally be instantiated by calling *JController.retrieveStoredCollection()*,a static method

which creates and returns an instance of the Stored Collection, containing Stored Objects that have been loaded with data from the underlying database query.

Stored Collections that have not yet been saved on the database should generally be instantiated by calling *JController.getStoredCollection()*,a static method which creates and returns an empty instance of the Stored Collection. Once the Stored Objects have been loaded with data and added to the Stored Collection, the *JStoredCollection's SaveAll()* method will commit the new objects' contents to the database. *JStoredCollection* is rarely used this way.

For each Stored Collection class in the Problem Domain, there should be a corresponding subclass of *JDataClass* in the Data Management Domain. The role of the *JDataClass* is to:

- populate Stored Objects that are in the Stored Collection from the database.
- save the Stored Objects that are in the Stored Collection into a database table.
- delete Stored Objects that are in the Stored Collection from the database.

These operations occur at a much lower level than in the *JStoredCollection,* usually involving specific SQL calls to the database. More about these classes later.

JStoredCollection.java

Another important purpose of the *JStoredCollection* relates to Java's use of 'models'. Java uses models to store the data that is displayed in visual components such as lists, combo boxes and tables. Implementing a model for a set of data is easy with *JStoredCollection:* all that is needed is for one or two abstract methods of the *JStoredCollection* to be defined. The *JStoredCollection* can then be attached to a visual component very easily, by calling the *JStoredCollection.*get[List|Combo|Table]Model*()* method.

For example, this code sets the list model for a list box (jListBoxConditions) to be the Stored Collection 'foConditions':

```
jListBoxConditions.setModel( foConditions.getListModel() );
```

2.6.5 *Adding a new* **JStoredCollection** *to the P-Domain*

1. Create a new class in _ProblemDomain (referred to here as "[the Stored Collection]"), by subclassing *JStoredCollection* from package ProblemDomain. Make the constructor private.

2. To allow the object to be instantiated correctly (by a factory), define the *CreateMe()* method as follows:

```
static public Object CreateMe() {

  validateCaller();

  return new [the Stored Collection]();
};
```

Attach [the Stored Collection] to a data object IF IT NEEDS TO LOAD ITSELF with data, by adding this call into the (private) constructor, but leave it commented out:

```
//JDataObject[name].RegisterMe(this.getClass().getName());
```

3. In your application, register [the Stored Collection]:

```
JController.registerStoredCollection("[a name]", "_ProblemDomain.[the Stored Collection]");
```

e.g.

```
JController.registerStoredCollection("Persons","_ProblemDomain.JStoredCollectionPersons
)
```

4. Create the application-specific (public) fields for [the Stored Collection]. These should be minimal because the actual data is stored in the fields of Stored Objects, which in turn are stored in the vector field of *JStoredCollection* (fvItems).

Add these imports to the top of the source file:

```
import ProblemDomain.*;
import FactoryDomain.*;
import _DataManagementDomain.*;
```

<u>Now, to create the data class (described later in this document) that will be used by this Stored Collection**:**</u>

5. Create a new Data Object (referred to here as "[the Data Object]") by subclassing *JDataClass* from package DataManagementDomain. Make the constructor private.

6. In the [the Data Object], define the *RegisterMe()* method (to register the class in the factory).

```
static public void RegisterMe( String sClassName ){

    [the Data Object] oTemp = new [the Data Object]();

    JController.registerDataObject( sClassName, oTemp.getClass() );
};
```

Ensure that these are at the top of the source file:

```
import DataManagementDomain.*;
import ProblemDomain.*;
import FactoryDomain.*;
import _ProblemDomain.*;
```

7. In the [the Data Object], define the *CreateMe()* method like this:

```
static public Object CreateMe() {
  validateCaller();
  return new [the Data Object]();
}
```

8. Remove the comment from the constructor of [the Stored Collection]:

```
private [the Stored Collection]() {
  [the Data Object].RegisterMe(this.getClass().getName());
}
```

Add:

```
import _DataManagementDomain.*;
```

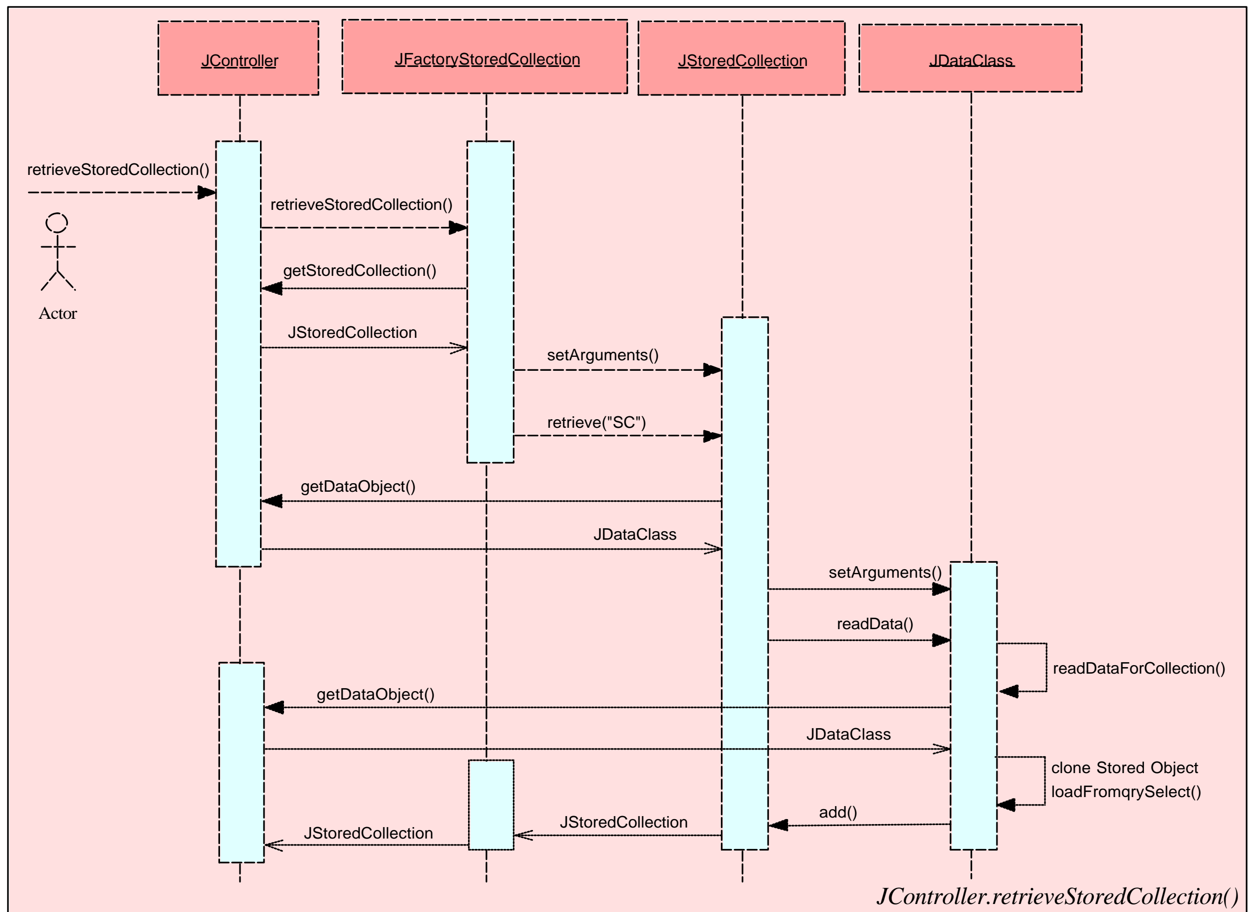
9. Test [the Stored Collection] by calling

```
JController.retrieveStoredCollection( [Stored Collection name] {, params} )
```

or

```
JController.getStoredCollection(  [Stored Collection name]{, params} )
```

for a Stored Collection without loaded data.



2.6.6 *Class* JService

JService.java

A Service is the implementation of some business activity, which may be achieved through the interaction of one or many other Services, Stored Objects and Stored Collections.

Business processes do not have direct corresponding data management classes in the Data Management Domain.

So, when would a programmer create a service? This is best illustrated with an example:

Lets assume that a program needs to download all the messages from somebody's mailbox, on an ordinary mail server. Each message must be saved locally, after it has been scanned for the presence of specific keywords. Based on this description of the problem, two Service classes could be used.

The first Service (let's call it *JServiceGetMail ()*) connects to the mail server, opens the relevant mailbox and one by one, iterates through all the messages. On each iteration, *JServiceGetMail()* calls another Service (let's call it *JServiceScanMessage()*), passing it one message at a time for scanning. This shows how one 'trivial' service can be initiated by another more complex Service to achieve an objective.

Now obviously, programmers could go to the extreme, creating a multitude of tiny Services to control all processes within a system. Every loop or block of code could be replaced with a dedicated Service. This is a perfect example of the conflict between theory and practice. In theory, a proliferation of 'modularity' is desirable, but in practice it would make the code difficult to work with. Discretion is advised.

As a rule of thumb, a *JService* subclass should be an independent process (most likely fired off by a GUI event, such a s a button click) and be a clearly discernable task. With reference to the example above, the two tasks can be defined as:

JServiceGetMail () - downloads messages
JServiceScanMessage () - scans a message for keywords.

2.6.7 Adding a new **JService** *to the P-Domain*

1.Create a new class in _ProblemDomain (referred to here as "[the service]"), by subclassing *JService* from package ProblemDomain. Make the constructor private.

2. To allow the object to be instantiated correctly (by a factory), define the *CreateMe()* method as follows:

```
static public Object CreateMe() {

  validateCaller();

  return new [the service]();
};
```

3. In your application, register [the service]:

```
JController.registerService("[a name]", "_ProblemDomain.[the service]");
```

e.g.

```
JController.registerService("DoSomething", "_ProblemDomain.JServiceDoSomething");
```

4. Define the behaviour of the *Execute*() method in [the service].

6. Test [the service] by calling

```
JController.executeService( [service name] {, arguments} );
```

2.6.8 Executing Services on a separate thread

Often, data must be loaded into internal objects while a screen is being painted. It can be desirable to load such objects in the background, by threading the process separately. *JProblemClass* contains three methods relevant to this function:

loadStoredCollectionOnThread(), *loadStoredObjectOnThread()* and *afterThread()*.

JProblemClass.java

JProblemClass establishes a callback loop so that when the service that loads the Stored Object/Collection is finished, it calls the *afterThread()* method in the object that initiated the thread loading.

As an example, consider this method of a Panel Director, which instructs a service to load the Stored Collection that has been registered under the name "AddressBooks".

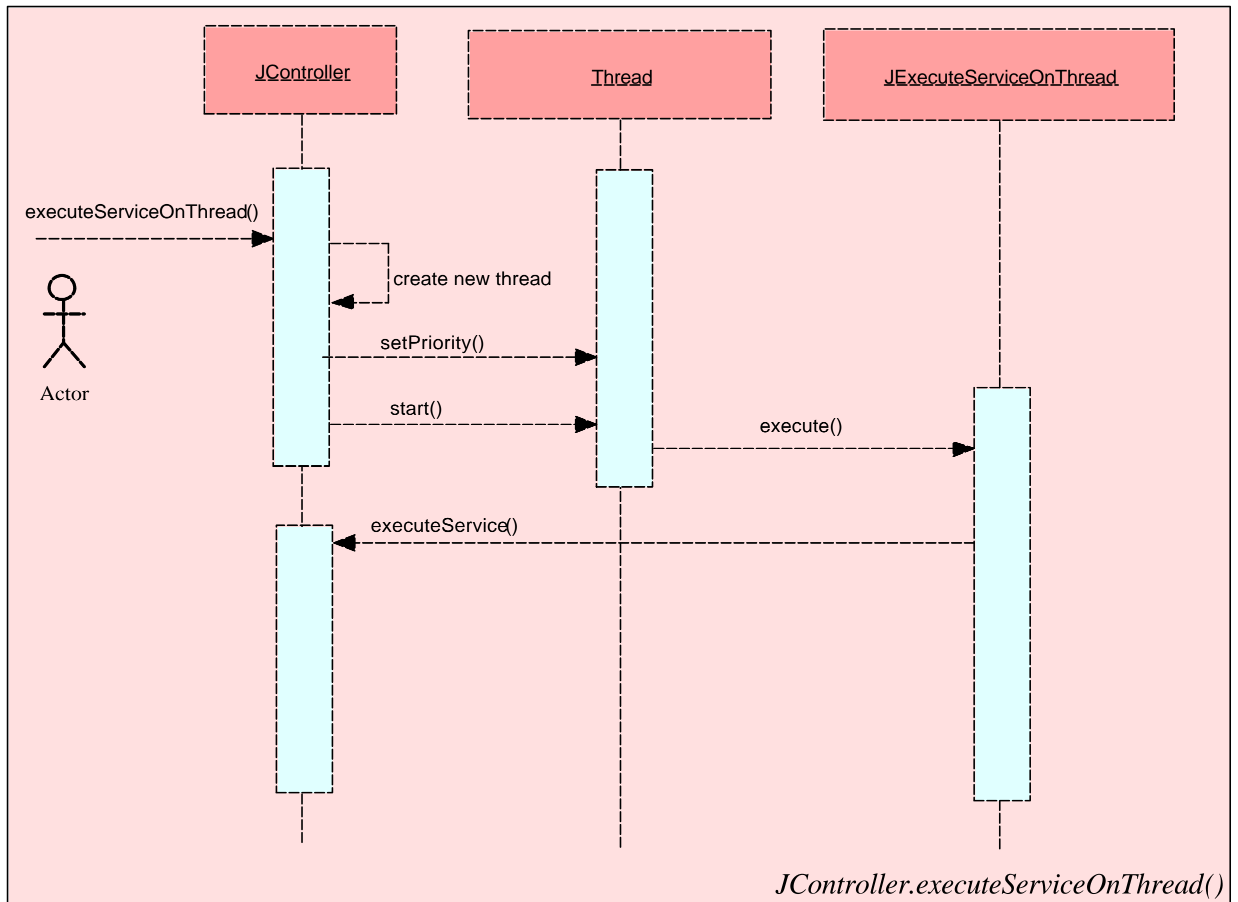
```
protected void postLoad() {

       loadStoredCollectionOnThread("AddressBooks");

}
```

If the Panel Director with this *postLoad()* method has also defined the *afterThread()* method, upon completion *loadStoredCollectionOnThreadI()* will call *afterThread()*.The method *afterThread()* received a copy of the Stored Object/Collection into which data has been loaded.

```
public void afterThread(JStored pStored) {

    if (pStored.getMoniker()=="AddressBooks") {

      foAddresses = (JStoredCollectionAddressBooks)pStored;

    }
}
```

The example given above relates to the specific implementation of Services on threads for the sake of loading *JStored* subclasses. In fact, any Service can be executed on a thread. This is appropriate for asynchronous operations.

```
JController.executeServiceOnThread("Get Mail", oArguments );
```



2.6.9 *Class JAssertion*

JAssertion.java

Filling your code with methods that throw exceptions can be very messy. If you need to raise an exception when a condition fails, simply call the static *assert()* method:

```
JAssertion.assert( iSalary<95000, "This programmer is underpaid!" );
```

2.6.10 *Class JCommandExec*

JCommandExec.java

This class allows the command-line execution of statements from within Java applications. This is handy when working with Interbase (gsec & gbak) under Linux.

2.6.11 Implementation of the Observer Pattern

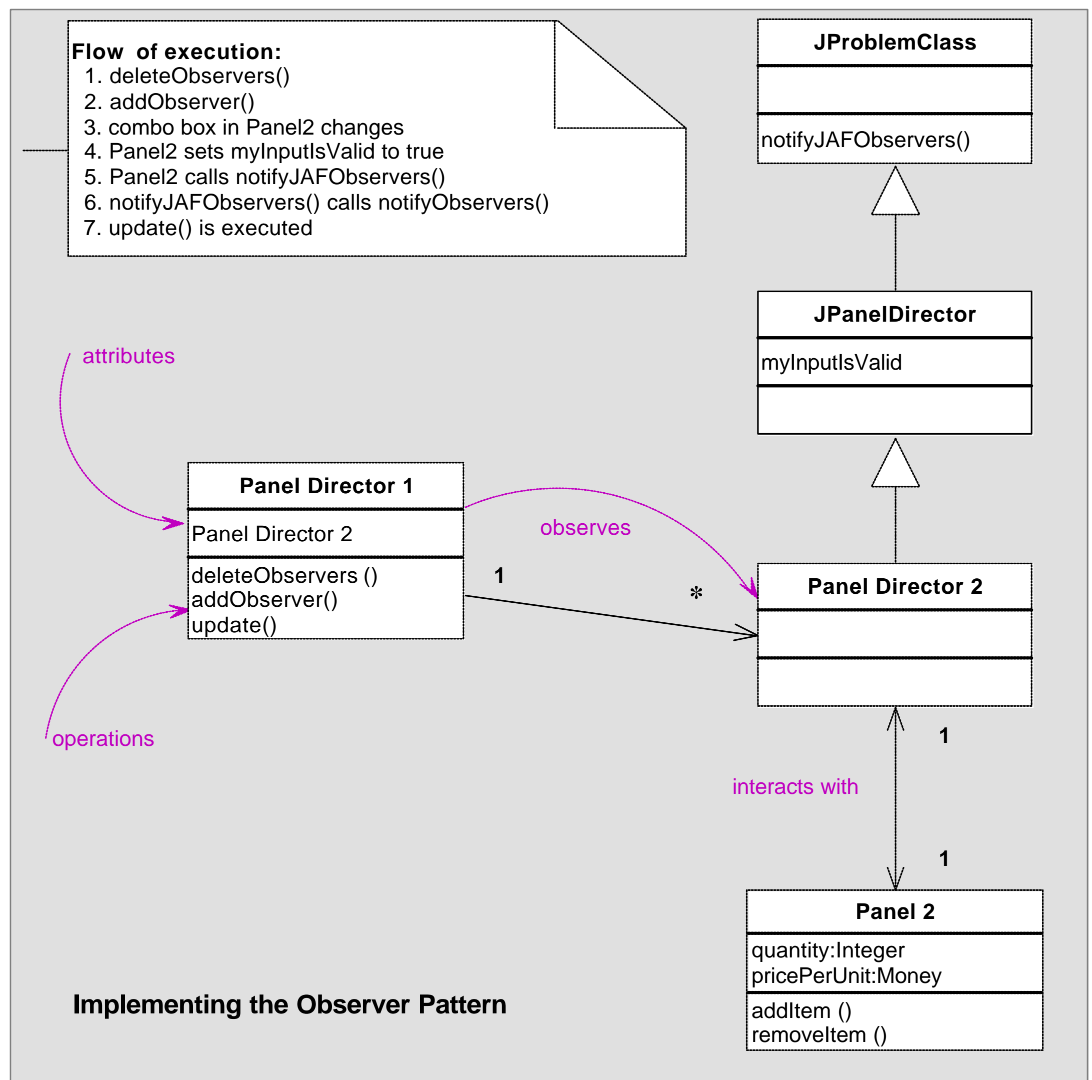
For an explanation of the observer pattern, refer to the GoF.

Some objects need to be notified automatically when the state of other objects changes. A good example is the relationship between a screen and the object containing the data being shown on that screen. If the data in said object changes, the screen must be repainted to reflect the underlying changes.

The most appropriate classes to be observed are *JPanelDirector, JService, JStoredObject* and *JStoredCollection*. They may also wish to observe each other. These classes are all able to observe, and be observed by each other, by virtue of their inheritance from *JProblemClass*. *JProblemClass* inherits the Java superclass **Observable** and it also implements the **Observer** interface.

This pattern can be implemented by:

Overriding void *update()* in a *JProblemClass* subclass that will observe another object.
Calling *notifyJAFObservers()* when an observed object changes



Implementing the Observer Pattern

For example, this code is taken from a *JPanelAbstract* subclass. The Director that controls this Panel is being observed by another object. When the value in the combo box *jcbExportTemplate* is changed, calling the *notifyJAFObservers()* method in the Director will initiate the execution of the *update()* method in the observing object. The Director is also informed that the data entry for the Panel is valid, if the selected value in the combo box is not null.

```
void jcbExportTemplate_itemStateChanged(ItemEvent e) {

  if ( getDirector()!=null ) {

    String s = (String)jcbExportTemplate.getSelectedItem();

    getDirector().myInputIsValid = ( s.toString().length() > 0 );

    getDirector().notifyJAFObservers();
  }
}
```

2.7 The Data Management Domain (DM-Domain)

IDataManagementDomain.java

The data management domain contains the knowledge required by the application to interact with a database.

Classes in this domain have knowledge of the physical data organisation within a database. Some also have intimate knowledge of the Stored Objects in the problem domain that have been stored in the database.

2.7.1 The Data Management Layer

The Data Management layer is an abstract infrastructure that models the physical database system. It allows objects in the application domain to be built as generically as possible, without having to make design decisions based on limitations or extensions in the database system.

The Data Management Layer has knowledge of the Problem Domain. This allows a Stored Object/Collection to be passed to its associated Data Object, which will initiate the load/save/deletion of the data in the underlying database.

JDataClass.java

The core of this layer is *JDataClass,* the base class for all data operations. It contains a JKey, since single-record data operations usually rely on a single primary key. *JDataClass* subclasses can service either Stored Objects or Stored Collections. In the former case, the *JDataClass* subclass will read a single database record into its corresponding Stored Object. In the case of the latter, the *JDataClass* subclass reads the contents of a dataset into a stored collection of stored objects.

In general, an application will have four particular subclasses for each database table, though this is not necessarily the case. For instance, if a table “Person” exists, the application will probably require:

JStoredObjectPerson + JDataObjectPerson
JStoredCollectionPersons + JDataObjectPersons

Data Classes are thus of two types: one to service the needs of a Stored Object (such as representing one database record) and another to service the needs of a Stored Collection (such as multiple records, part of a data set). Pay attention to the strict naming convention applied to JAF subclasses.

An explanation of how to create a new *JDataClass* subclass is given above, as part of the explanation of how to subclass a Stored Object (2.6.3).

2.7.2 The Database Implementation Layer

The database implementation layer protects applications from having to make design decisions that may be affected by the physical database set up/ location/ limitations etc.

For example, an Interbase database may be set up on an Interbase Server. Ideally, the business logic handling the data management by the application should not be dependant either upon the underlying database type, server or client engine. This way, if it becomes necessary to run the application on Microsoft SQL Server, only this layer needs to change. In fact, the application may even be downsized to run on a non-server based system such as Paradox.

The database implementation layer is concerned with how the application views, gains and uses connections to the database. This is facilitated by two important classes: *JConnections* and *JDatabase*.

2.7.3 *Class JConnections*

JConnections.java

This is a singleton class containing a list of database connections. The various database connections can be established at any time. Each database connection must have a unique moniker.

Basically, the first connection to be set in *JConnections* becomes the ‘Active’ connection. The Active connection should be set manually, despite the default. All instructions relating to “the” database are directed by JAF to the active database. To swap databases, simply call *setActiveConnection()* to denote the Active connection.

This code shows:

The use of 2 database connections
the connection "ORACLE_DATA" uses the class *JOracle*
the connection "SYBASE_DATA" uses the class *JSybase*
the active connection is to the Oracle database
both the Oracle and the Sybase connections are open.

```
JConnections.addConnection("ORACLE_DATA", "JOracle", … etc.);
JConnections.addConnection("SYBASE_DATA", "JSybase", etc.);
JConnections.setActiveConnection("ORACLE_DATA");
JConnections.openActiveConnection();
JConnections.openConnection("SYBASE_DATA");
```

2.7.4 *Class JDatabase*

JDatabase.java

In the example code shown above (2.7.3), *JOracle* and *JSybase* are subclasses of *JDatabase*. This class defines the manner in which a connection is established between the application and a database using JDBC (refer to the *openDatabase()* method).

JInterbase.java

To create a class for opening a previously undefined database, subclass *JDatabase* and override the methods *getDriver()* and *getJDBCPrefix()*. This has already been done for Interbase connections in JInterbase.

3 Generating an execution trace

ProblemDomain.JTracer.java

JAF is capable of generating a trace of every operation it performs, by using the *JTracer* class. This class is a singleton that has static methods accessible to users of JAF. If the programmer wants to print a trace, using *JTracer.showTrace()* instead of System.out.println() allows all tracing to be switched on and off for the entire application.

The trace appears in the trace window by default, but can also be streamed out to a text file called “trace.txt”. To activate this, the programmer simply calls *JTracer.directTraceToFile()* at the start of the main application. It is at this point that tracing can be activated, by calling *JTracer.setShowTrace([argument]).* To find out if tracing is enabled, *JTracer.isTracing()* can be called.

3.1 To activate tracing from the command line

Specify the parameters “Trace=ON” or “Trace=FILE”

If you wish to specify a domain or class, make sure this specification does not conflict with project tracing settings, eg. If the project includes command line arguments to switch on the tracing ('Trace=ON'), this will overwrite the *JTracer* specifics if this command executes after the *JTracer* specifics have been set.

3.2 To see all JTracer messages

Near beginning of application execution add line 'JTracer.setShowTrace("");'.

3.3 To see JTracer messages from a specific domain

Near beginning of application execution add line 'JTracer.setShowTrace("<domain>");'

eg. 'JTracer.setShowTrace("UserInterfaceDomain");'

3.4 To see JTracer messages from a specific class

Near beginning of application execution add line
'JTracer.setShowTrace("<required class>");'

eg. 'JTracer.setShowTrace("JStoredObjectMessage");'

These can be mixed and matched eg. 'JTracer.setShowTrace("UserInterfaceDomain FactoryDomain JStoredObjectMessage");' (separated by a space).

Andrew N. Shrosbree
Ballarat, 21 January 2005.

--- fin ---