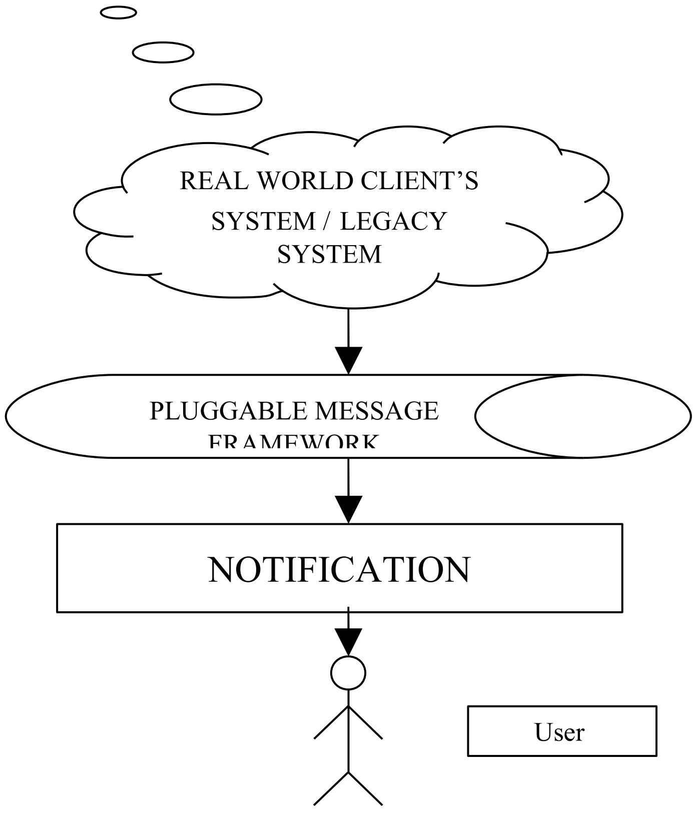
# A programmer's HandBook to GenericMessage application

A first draft Prepared By
Debadatta Mishra

## Contents

High Level Design Overview


REAL WORLD CLIENT'S SYSTEM / LEGACY SYSTEM
PLUGGABLE MESSAGE FRAMEWORK
NOTIFICATION
User

**Description**

The above diagram depicts the overall high level vision of this developed message module. This message notification framework acts as a pluggable application that can sit over any existing legacy system and sends message to the end user as per the requirements.

# ***Generic Message Notification Framework Technical Architecture***

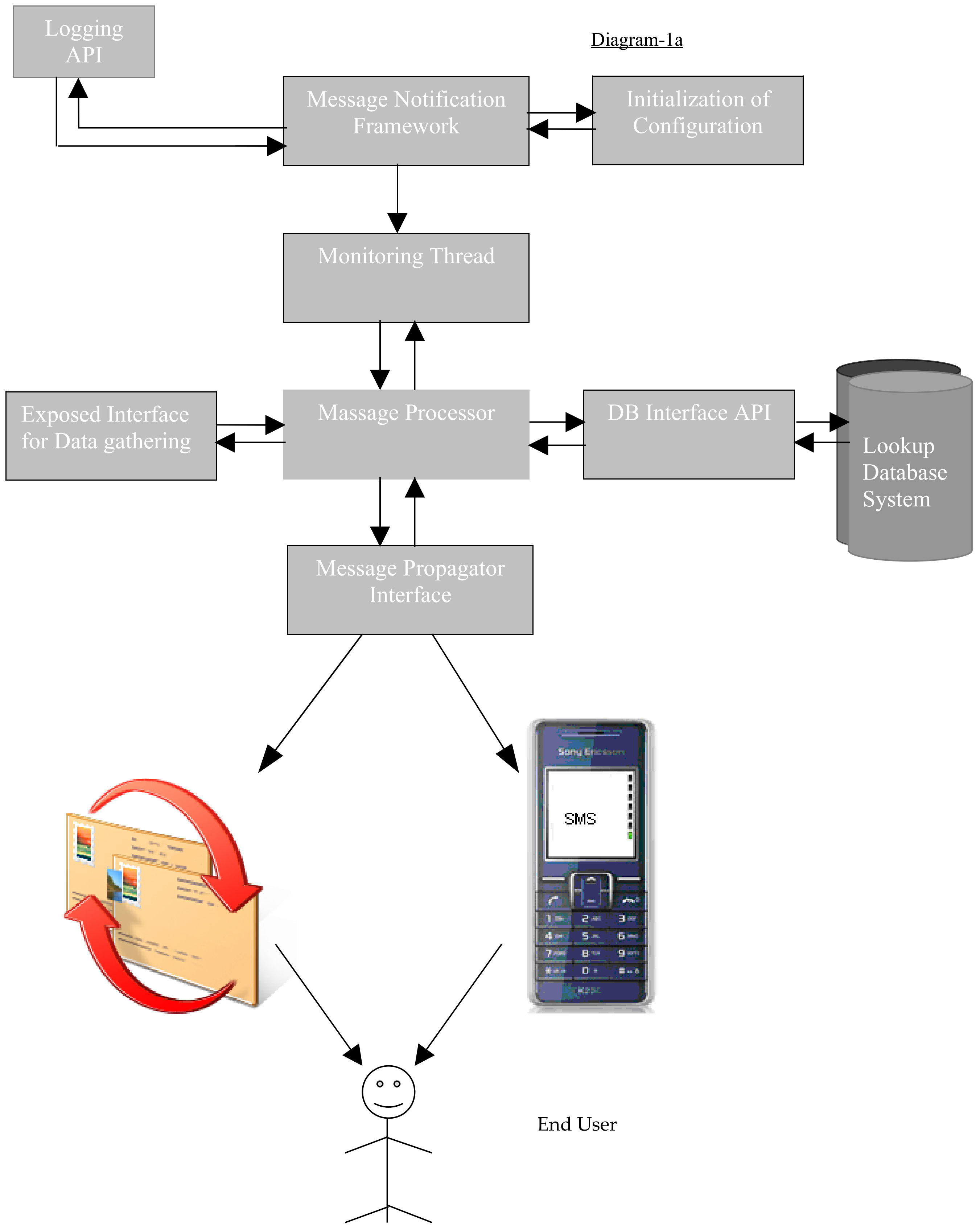


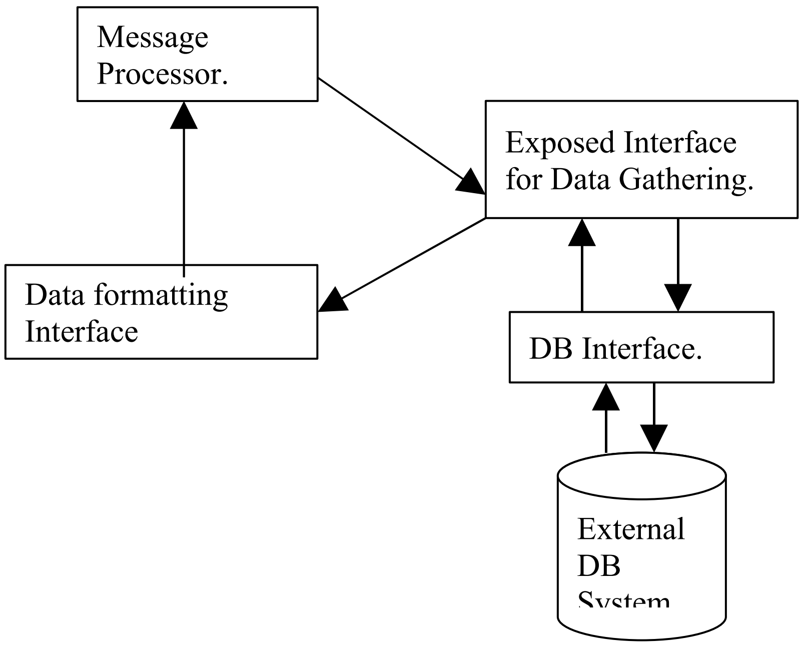
Diagram-1a
Logging API
Message Notification Framework
Initialization of Configuration
Monitoring Thread
Exposed Interface for Data gathering
Massage Processor
DB Interface API
Lookup Database System
Message Propagator Interface
Sony Ericsson
SMS
End User

**Description**

The above diagram ie. Diagram-1a describes the whole technical architecture of the developed application.

This diagram has been divided into two sections. One is core message notification system which provides the configurable api to interact with the existing legacy system. Another part describes the interaction with the external system.

Diagram-1b


Message Processor.
Exposed Interface for Data Gathering.
Data formatting Interface
DB Interface.
External DB System

**Description**

The above diagram ie.diagram-1b describes the core interaction part with the external system. This is the developed version of the underlying message notification system. However the above architecture may be altered depending upon the requirements of the external system, but core architecture will not be affected. So far current design and requirement is concerned, it has been developed with some precise concept. In the later version, the architecture will be changed to make it more generic.

## **Detailed Implementation Overview**

The following description gives an insight into the implementation details of the "**Generic Message Notification**" framework. This implementation incorporates some of the other developed framework like "**Email**" and "**CustomApplicationLogger**". These implementation details are not covered here as these are out of scope of this message module. To get the notion of the above framework, refer to their Design and Functional specifications. These frameworks are the supporting utilities for this application.

Here some of the Class diagrams will be provided here and the internal brief logic and implementation will be covered. It is always assumed that the developer has the knowledge of java application system.

### **Application structure details**

To know about the complete directory structure of the application , please go through the **"*ReadMe.doc*"** found in the root directory of the application. Also read the ***changelog.log*** file to know the change log history. Here only the relevant files in the source folder will be explained here. So far the application structure is concerned, the **src** folder contains all the .java files with the following package structure.

Package structure
com.iit.core.action
com.iit.core.bean
com.iit.core.common
com.iit.core.config
com.iit.core.data
com.iit.core.db
com.iit.core.loader
* com.iit.core.logger
* com.iit.core.message
com.iit.core.msg.data
com.iit.core.scheduler
com.iit.core.ui
com.iit.core.util

The .java files present inside the above red colored package structure are the files of the supporting framework of the current application.

Besides the directory **test** contains all the java files for unit testing. Right now manual unit testing has been performed without the use of any test case frameworks. In the later release all the test cases will be includes include the Junit test suite.

So far configuration files are concerned, this application needs some configuration files for its internal use. The configuration files are present inside the directory called **configuration**. Since these configuration files(except .bat) are the part of the supporting frameworks, explanation details will not be covered here.
All the .bat files are required for running the application in the client side.

**Source package description**

com.iit.core.action :- This package contains only one file called the "***ActionProcessor.java***". This file is the most significant file for this message application. All the operation begins from this java class.

com.iit.core.bean: This package contains all the java files called the bean classes for the application.

com.iit.core.common**:** This package contains all the files which are used as Common constants in the application to avoid hard coding.

com.iit.core.config : This package contains all the java files which are used to create and to read the runtime configuration files specific to this application. The runtime configuration files are generated at the start up of the application.

com.iit.core.data : This package contains only one file for formatting the data for the specific requirements.

com.iit.core.db : This package contains all the files used for database interaction.

com.iit.core.loader :This package contains all the java files used for dynamic class loading for this application. This application has a feature of loading of third party library file provided by the client.

com.iit.core.msg.data : This package contains only one file which exposes an interface to the external system.

com.iit.core.scheduler : This package contains all the files used to create a scheduler for this application.

com.iit.core.ui : This package contains all the User interface for this application.

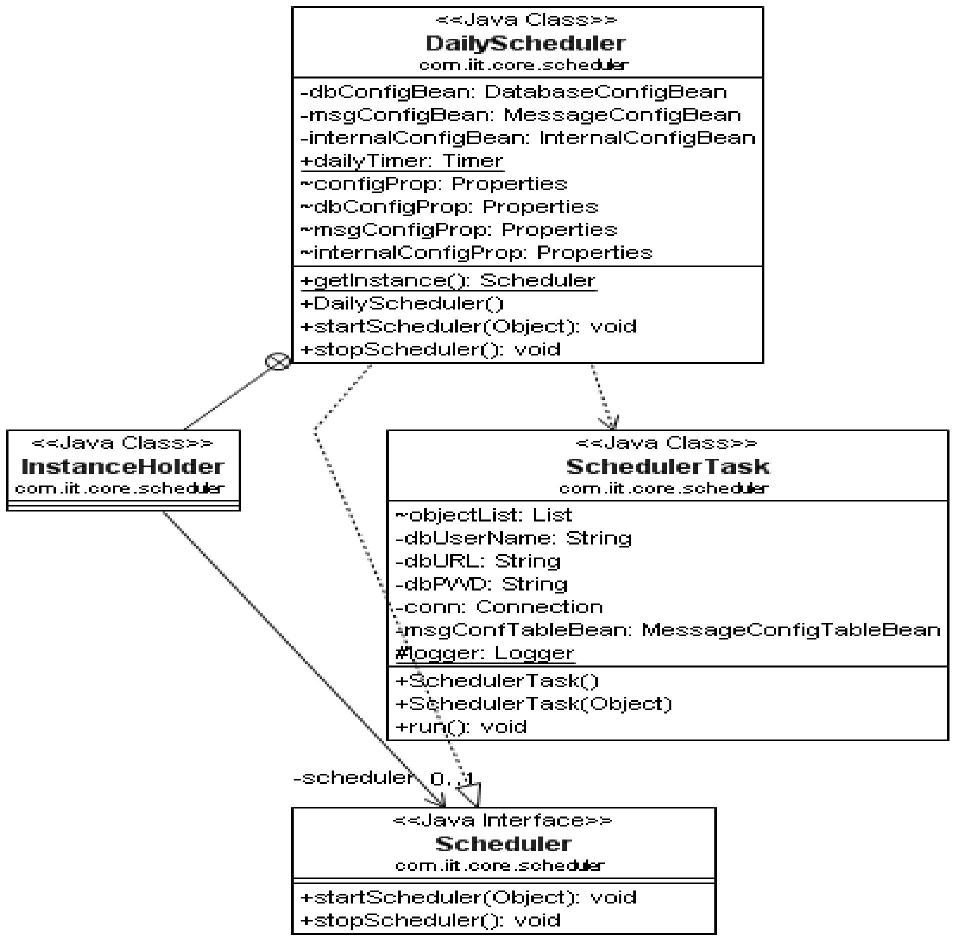
com.iit.core.util : This package contains all the utility classes used in this application.

**Implementation logic overview in brief**

As a high level superficial overview, this application has a thread which is running continuously to look up the current dbms system. If there is a change in data, it captures the changes, passes the requirements to the external system interface to gather the formatted data. Now the formatted data are sent to message propagation station . Now the propagation station takes care where to deliver the message. Once the message is delivered, a message feedback is propagated to an action which makes changes in the current dbms system , which is called the delivery status. So this process continuously goes on if there is a change in the current dbms system. Some configuration files are generated from the UI of the application. This is the one time configuration for a particular system.

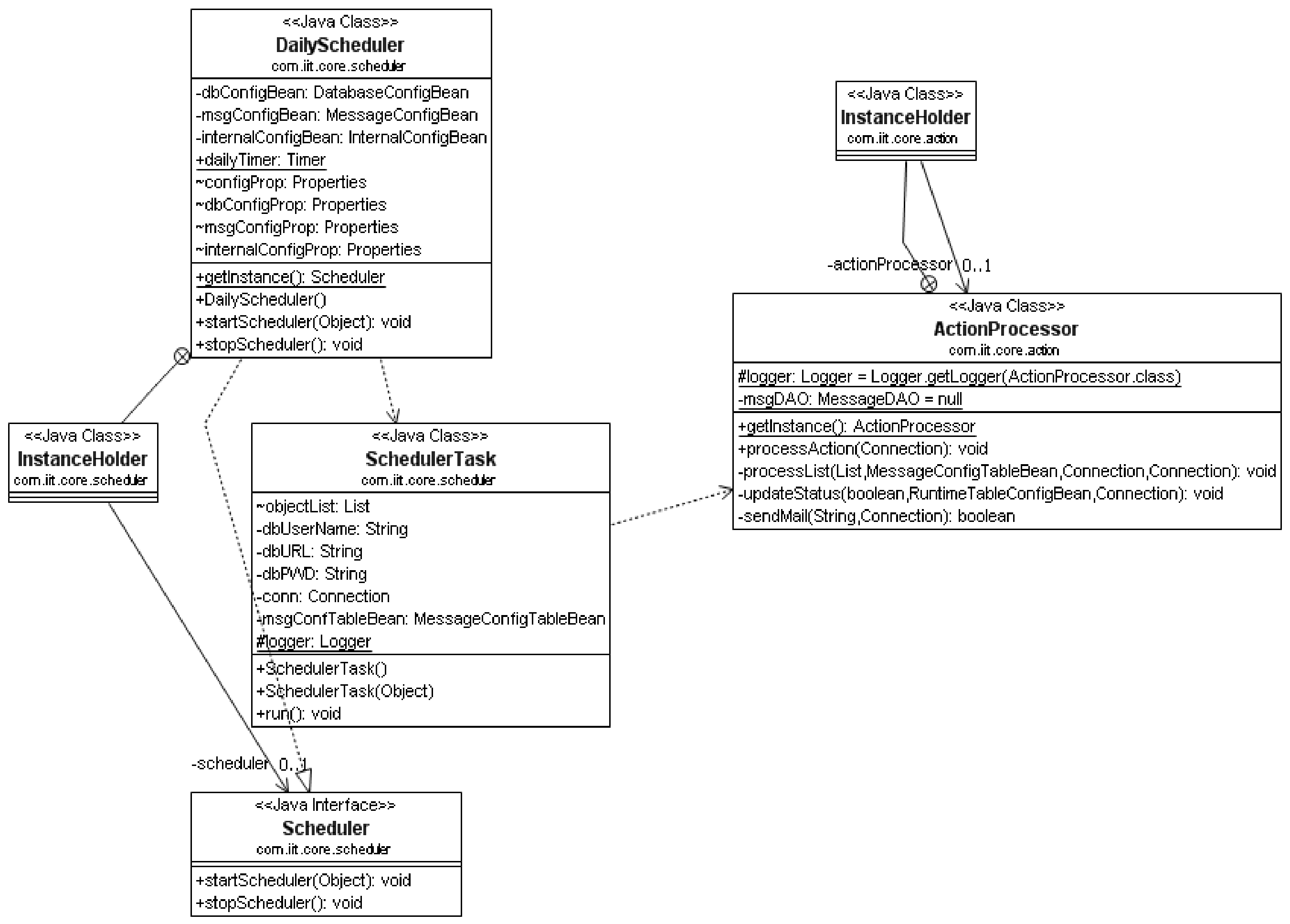
**Class level Implementation details**

The current application has a scheduler which keeps on running. This scheduler has been developed with the help of java.util.Timer. The complete class diagram is given below.



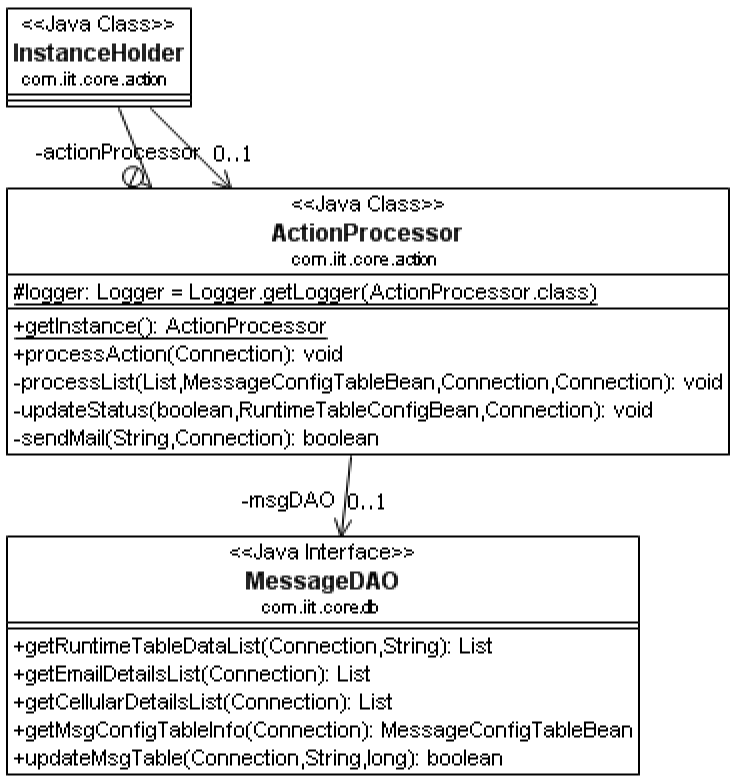
The class “**DailyScheduler**” is a singleton implementation of the interface “**Scheduler**”. This class at the startup performs some of the initialization process to gather the basic configuration details for the scheduler. The class “**SchedulerTask**” has a run method to start the scheduler. So the scheduler starts from this class. The class “**SchedulerTask**” interacts with the the class “**ActionProcessor**”.

The overall class diagram for both the Scheduler and Action is given below.


<<Java Class>>
DailyScheduler
com.iit.core.scheduler
-dbConfigBean: DatabaseConfigBean
-msgConfigBean: MessageConfigBean
-internalConfigBean: InternalConfigBean
+dailyTimer: Timer
~configProp: Properties
~dbConfigProp: Properties
~msgConfigProp: Properties
~internalConfigProp: Properties
+getInstance(): Scheduler
+DailyScheduler()
+startScheduler(Object): void
+stopScheduler(): void
<<Java Class>>
InstanceHolder
com.iit.core.action
-actionProcessor
0..1
<<Java Class>>
ActionProcessor
com.iit.core.action
#logger: Logger = Logger.getLogger(ActionProcessor.class)
-msgDAO: MessageDAO = null
+getInstance(): ActionProcessor
+processAction(Connection): void
-processList(List,MessageConfigTableBean,Connection,Connection): void
-updateStatus(boolean,RuntimeTableConfigBean,Connection): void
-sendMail(String,Connection): boolean
<<Java Class>>
InstanceHolder
com.iit.core.scheduler
<<Java Class>>
SchedulerTask
com.iit.core.scheduler
~objectList: List
-dbUserName: String
-dbURL: String
-dbPWD: String
-conn: Connection
-msgConfTableBean: MessageConfigTableBean
#logger: Logger
+SchedulerTask()
+SchedulerTask(Object)
+run(): void
-scheduler
0..1
<<Java Interface>>
Scheduler
com.iit.core.scheduler
+startScheduler(Object): void
+stopScheduler(): void

The above diagram displays the dependency between the scheduler and the action processor which processes the entire application logic.

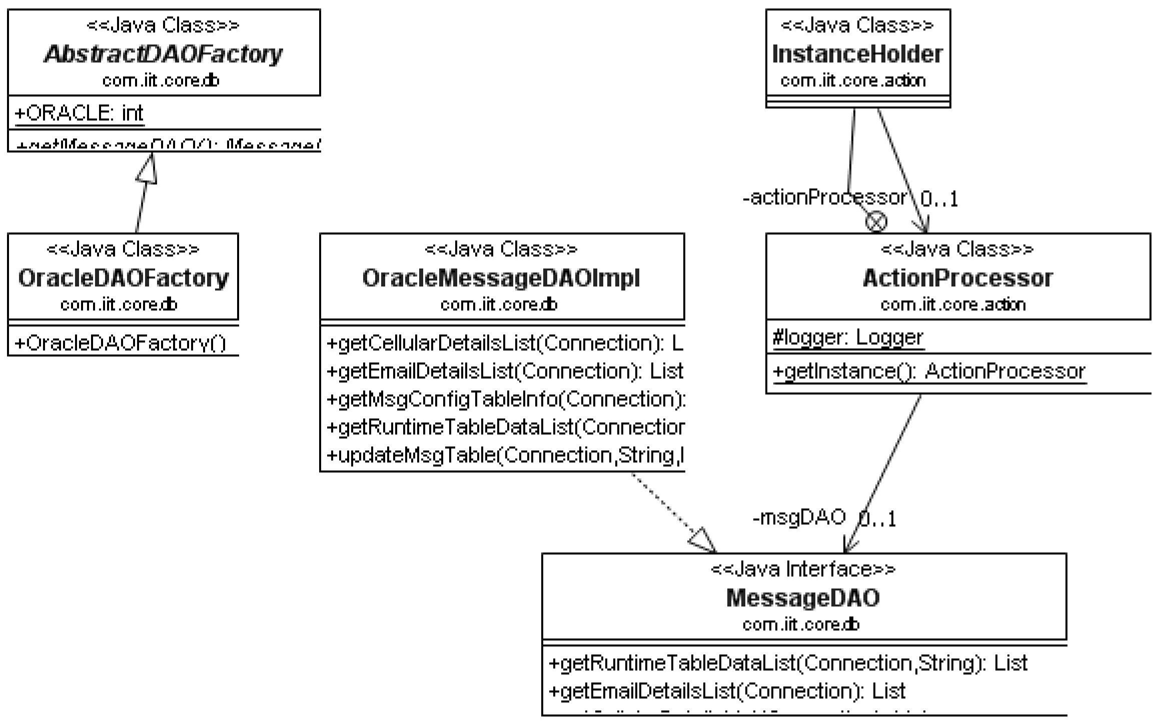
The following diagram only displays the class structure of the "**ActionProcessor**" class.



<<Java Class>>
InstanceHolder
com.iit.core.action
-actionProcessor
0..1
<<Java Class>>
ActionProcessor
com.iit.core.action
#logger: Logger = Logger.getLogger(ActionProcessor.class)
+getInstance(): ActionProcessor
+processAction(Connection): void
-processList(List,MessageConfigTableBean,Connection,Connection): void
-updateStatus(boolean,RuntimeTableConfigBean,Connection): void
-sendMail(String,Connection): boolean
-msgDAO
0..1
<<Java Interface>>
MessageDAO
com.iit.core.db
+getRuntimeTableDataList(Connection,String): List
+getEmailDetailsList(Connection): List
+getCellularDetailsList(Connection): List
+getMsgConfigTableInfo(Connection): MessageConfigTableBean
+updateMsgTable(Connection,String,long): boolean

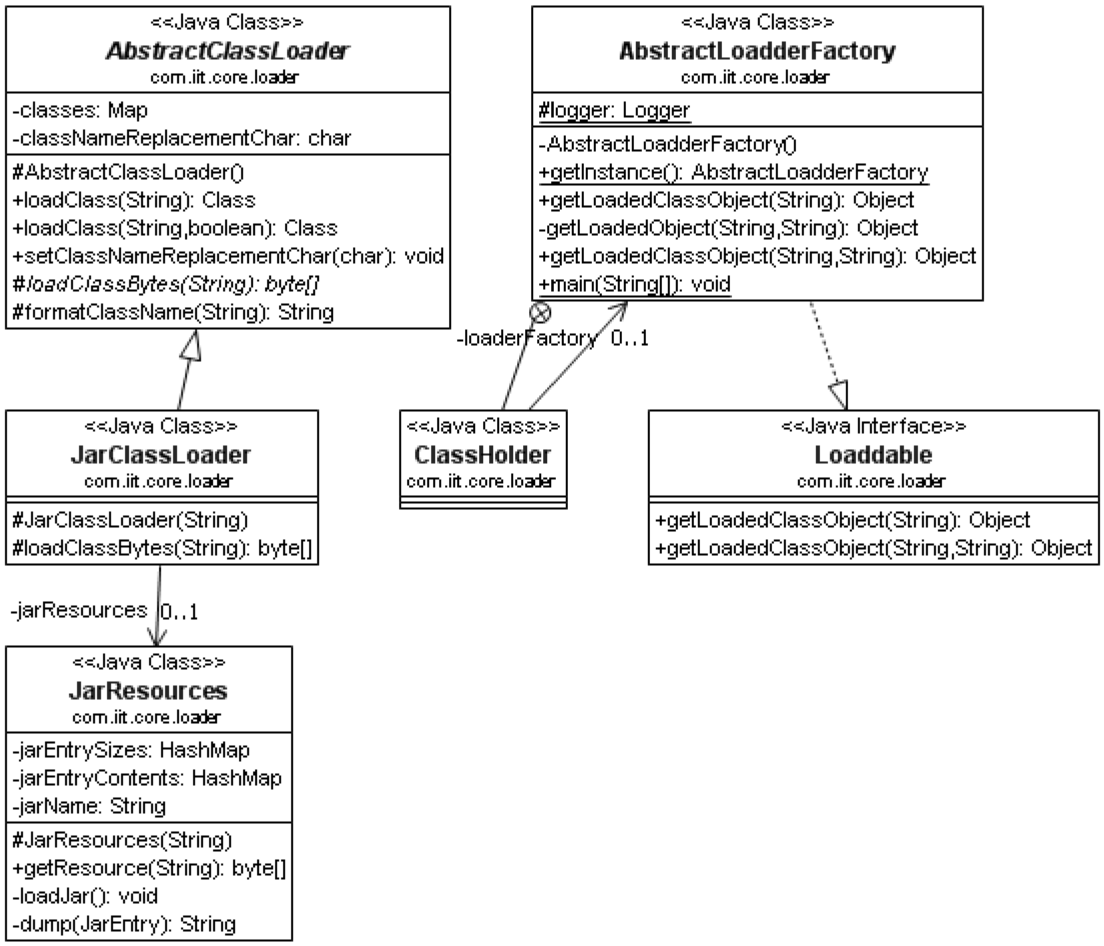
As per the diagram above, it is a singleton implementation to minimize the object creation from the view point of performance. This class first gathers the information from the Runtime table defined in the dbms system. It will obtain the details whose status is “U”( U – stands for unprocessed ). The Action processor interacts with the DAO for database interaction to gather database details.

The following diagram displays the interaction between the DAO for database and the Action.



The main DAO class is used here to obtain the loose coupling in the application. There is a flexibility here, if the underlying database gets changed, only one java file will be modified and rest will remain unaffected.

The next action of the action class is to interact with the external system to gather data so that it can send as SMS or email based upon the configuration. In order to get the data from the external system, there is a class called "**DataCollector.java**" which interacts with a class loader to get the data.. The class diagram for class loader is given below.


<<Java Class>>
AbstractClassLoader
com.iit.core.loader
-classes: Map
-classNameReplacementChar: char
#AbstractClassLoader()
+loadClass(String): Class
+loadClass(String,boolean): Class
+setClassNameReplacementChar(char): void
#loadClassBytes(String): byte[]
#formatClassName(String): String
<<Java Class>>
AbstractLoadderFactory
com.iit.core.loader
#logger: Logger
-AbstractLoadderFactory()
+getInstance(): AbstractLoadderFactory
+getLoadedClassObject(String): Object
-getLoadedObject(String,String): Object
+getLoadedClassObject(String,String): Object
+main(String[]): void
-loaderFactory 0..1
<<Java Class>>
JarClassLoader
com.iit.core.loader
#JarClassLoader(String)
#loadClassBytes(String): byte[]
<<Java Class>>
ClassHolder
com.iit.core.loader
<<Java Interface>>
Loaddable
com.iit.core.loader
+getLoadedClassObject(String): Object
+getLoadedClassObject(String,String): Object
-jarResources 0..1
<<Java Class>>
JarResources
com.iit.core.loader
-jarEntrySizes: HashMap
-jarEntryContents: HashMap
-jarName: String
#JarResources(String)
+getResource(String): byte[]
-loadJar(): void
-dump(JarEntry): String

Finally the class action processor sends the data to the email system. For the time there is no way to implement the message sending to the cellular system.

## Conclusion

The above describes the whole practical implementation of the Message framework. For debugging purpose you can provide the logger statement to get the deeper insight.