RITIS Development Framework

Dan Bucatanschi
Department of Computer Science
University of Maryland, College Park
Email: dangb@umd.edu

Abstract: Several goals of Intelligent Transportation Systems are to improve productivity, ease traffic congestion and minimize road risks through the use of advanced communications technologies. The RITIS software system developed at University of Maryland inside the Center for Advanced Transportation Technology Laboratory aims to improve communication among transportation agencies through data integration and sharing. We present the RITIS Development Framework, a software framework that aims to improve the design and maintainability of agency-to-RITIS data loaders and translators, while also minimizing the training and learning required to develop loaders for future data sources.

1 Introduction

1.1 What is ITS?

According to the Research and Innovative Technology Administration (RITA), Intelligent Transportation Systems (ITS) "improve transportation safety and mobility and enhance American productivity through the integration of advanced communications technologies into the transportation infrastructure and in vehicles. Intelligent Transportation Systems (ITS) encompass a broad range of wireless and wire line communications-based information and electronics technologies" [12].

In other words, Intelligent Transportation Systems try to alleviate some of the major transportation and traffic problems of the 21st century by using advanced communications technology, electronic equipment and computers. Numerous governmental and private agencies, researchers, companies and even individual citizens are involved in using and creating the necessary technology that helps contribute to solving the increasing traffic problem.

1.2 The goal of the Center for Advanced Transportation Technology Laboratory (CATT Lab)

The Center for Advanced Transportation Technology Laboratory (CATT Lab) is part of the Civil Engineering Department at the University of Maryland, and is one of the many participating agencies in ITS research [2]. The laboratory was created with the goal of researching innovative ways in gathering transportation data from around the DC metropolitan area (Northern Virginia, District of Columbia and Maryland) and presenting it in a useful manner to participating transportation agencies, university researchers and the general public.

Since its inception in 2001, the CATT Lab produced several research projects, the staple of which is the Regional Integrated Transportation Information System (RITIS). RITIS gathers and integrates various types of traffic information from various transportation agencies by presenting a multitude of interactive tools, which allow users to keep up to date with the state of the whole DC metropolitan area or explore historical information [3]. RITIS is composed of several software components. However, in this paper we shall focus on the way RITIS gathers and integrates data, and our contribution, the RITIS Development Framework, which helps developers in writing new components for integration with future data sources.

1.3 Outline

We start off by presenting related work to our field in Section 2 and follow with the RITIS overview in Section 3. We then move on to describing our problem (Section 4) and proposed solution (Section 5). Finally, we provide an evaluation of the solution in Section 6, possible future directions where the project can go next (Section 7), and our conclusions in Section 8. In Appendix A we provide a UML diagram depicting the RITIS Development Framework.

2 Related Work

The field of Intelligent Transportation Systems (ITS) is flourishing with many research projects as can be seen on the RITA website [12]. Ning and Li describe a “Comprehensive Information System” for railway networks. They address the issue of a common data format, but do not go into details about how data already gathered by legacy systems can be integrated into their new tools [9]. Kelly, et al. propose and implement a project for security management of vehicle fleets. Their approach is using an Object-Oriented Framework with various components programmed off of it [6]. Zavattero and Wu created a “Traveler Information System” that integrates data from across various agencies and presents it to the public and interested entities. Their work presents a monolithic system that uses CORBA for inter-agency communication [18]. Finally, Scherer presents “an overview of their vision” of the Northern Virginia (NOVA) system, which integrates data from various agencies and “fuses” it in a web-based format, but without providing an implementation and an evaluation [13].

The problem of integrating data from various legacy systems has been studied by [7], [8], [13], [17] and [18]. Lussier and Wu [8], and Kiel [7] dealt with the problem of integrating legacy geo-spatial data into ArcView, while Scherer [13], Yang, et al. [17], and Zavattero and Wu [18] discussed integration of traffic information data across various systems.

On the software engineering side, Heistracher, et al. present a common framework for ITS message passing based on the publisher-subscriber model using intermediate brokers and filters [5]. Choi, Choi and Yeom discuss guidelines for distributed software development using components, especially addressing issues such as distribution, performance, fault tolerance, security and distributed transactions. They use CORBA in their work, but also mention Enterprise Java Beans (EJB) [4]. Regarding integration of various software frameworks for message passing and remote method invocation, Bocquet and Gransart present a unified system for both synchronous (CORBA, RMI, RPC) and asynchronous (JMS, JavaSpaces) messages. However, they assume a totally homogenous system, since they do not deal with the issue of message translation [1].

3 RITIS Overview

The Regional Integrated Transportation Information System (RITIS) is a software system developed by CATT Lab that gathers and integrates various types of traffic information from various data sources by presenting a multitude of interactive tools, which allow users to keep up to date with the state of the system or explore historical information [3].

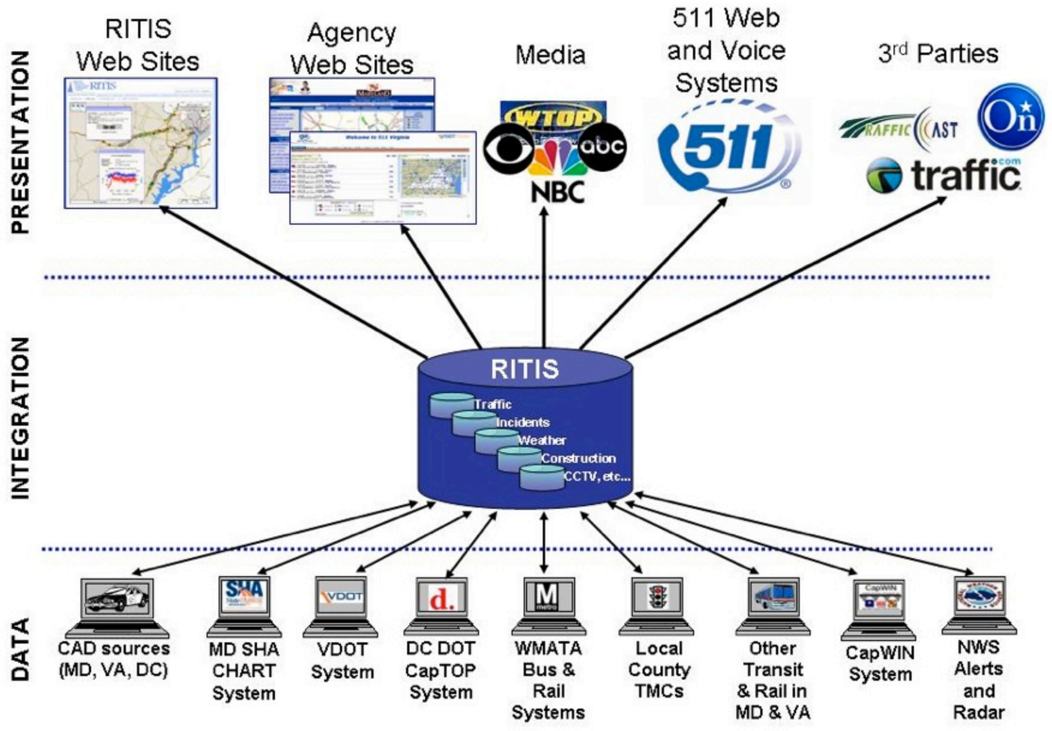
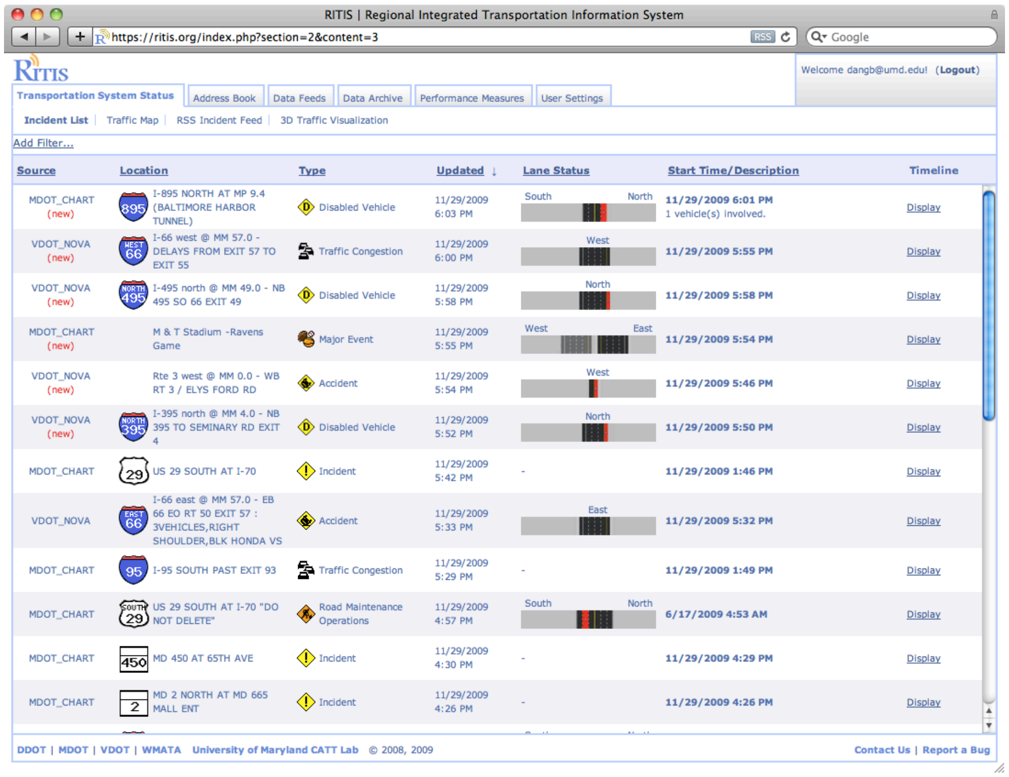


Figure 1: RITIS Logical Overview: The role of RITIS is to both integrate and present data from various agencies

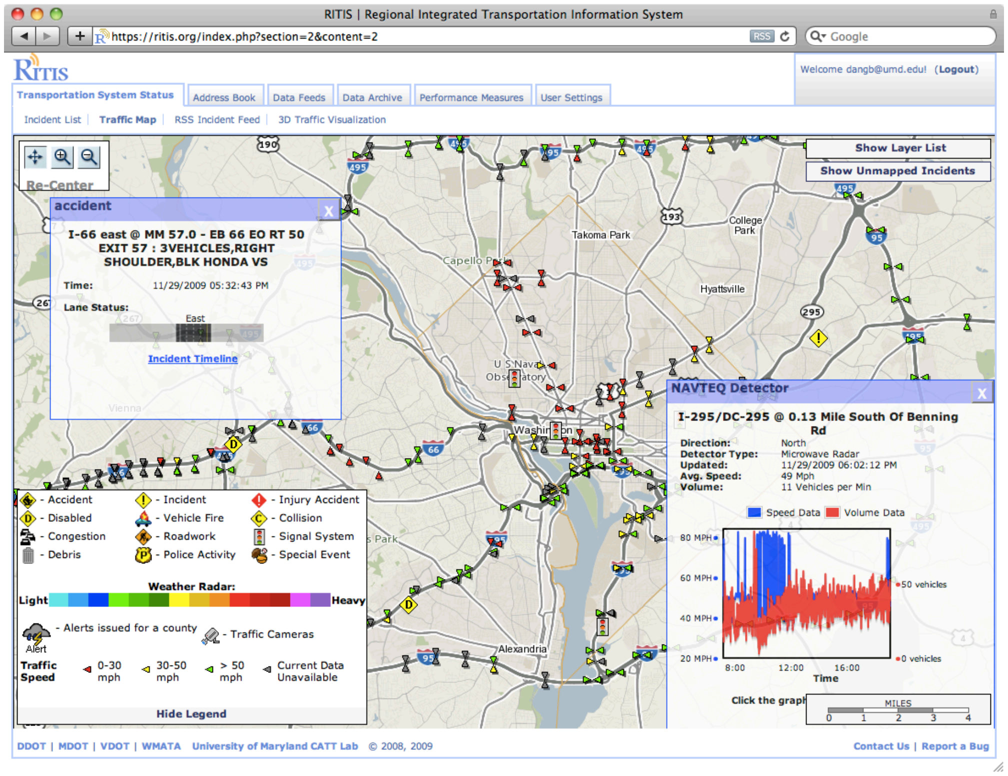
Figure 1 represents the logical design of RITIS. Various public and private agencies send data streams to RITIS and in return they receive updates with the globally integrated information back from the system. RITIS acts as both a tool for instant, real-time updates regarding unexpected incidents, planned events and traffic information, as well as an archival database with powerful interfaces tweaked specifically for visualizing various trends and anomalies.


RITIS | Regional Integrated Transportation Information System
https://ritis.org/index.php?section=2&content=3
Google
Welcome dangb@umd.edu! (Logout)
Transportation System Status
Address Book
Data Feeds
Data Archive
Performance Measures
User Settings
Incident List
Traffic Map
RSS Incident Feed
3D Traffic Visualization
Add Filter...

Source	Location	Type	Updated ↓	Lane Status	Start Time/Description	Timeline
MDOT_CHART (new)	I-895 NORTH AT MP 9.4 (BALTIMORE HARBOR TUNNEL)	Disabled Vehicle	11/29/2009 6:03 PM	South North	11/29/2009 6:01 PM 1 vehicle(s) involved.	Display
VDOT_NOVA (new)	I-66 west @ MM 57.0 - DELAYS FROM EXIT 57 TO EXIT 55	Traffic Congestion	11/29/2009 6:00 PM	West	11/29/2009 5:55 PM	Display
VDOT_NOVA (new)	I-495 north @ MM 49.0 - NB 495 SO 66 EXIT 49	Disabled Vehicle	11/29/2009 5:58 PM	North	11/29/2009 5:58 PM	Display
MDOT_CHART (new)	M & T Stadium -Ravens Game	Major Event	11/29/2009 5:55 PM	West East	11/29/2009 5:54 PM	Display
VDOT_NOVA (new)	Rte 3 west @ MM 0.0 - WB RT 3 / ELYS FORD RD	Accident	11/29/2009 5:54 PM	West	11/29/2009 5:46 PM	Display
VDOT_NOVA (new)	I-395 north @ MM 4.0 - NB 395 TO SEMINARY RD EXIT 4	Disabled Vehicle	11/29/2009 5:52 PM	North	11/29/2009 5:50 PM	Display
MDOT_CHART	US 29 SOUTH AT I-70	Incident	11/29/2009 5:42 PM	-	11/29/2009 1:46 PM	Display
VDOT_NOVA	I-66 east @ MM 57.0 - EB 66 EO RT 50 EXIT 57 : 3VEHICLES,RIGHT SHOULDER,BLK HONDA VS	Accident	11/29/2009 5:33 PM	East	11/29/2009 5:32 PM	Display
MDOT_CHART	I-95 SOUTH PAST EXIT 93	Traffic Congestion	11/29/2009 5:29 PM	-	11/29/2009 1:49 PM	Display
MDOT_CHART	US 29 SOUTH AT I-70 "DO NOT DELETE"	Road Maintenance Operations	11/29/2009 4:57 PM	South North	6/17/2009 4:53 AM	Display
MDOT_CHART	MD 450 AT 65TH AVE	Incident	11/29/2009 4:30 PM	-	11/29/2009 4:29 PM	Display
MDOT_CHART	MD 2 NORTH AT MD 665 MALL ENT	Incident	11/29/2009 4:26 PM	-	11/29/2009 4:26 PM	Display

DDOT | MDOT | VDOT | WMATA University of Maryland CATT Lab © 2008, 2009
Contact Us | Report a Bug

Figure 2: RITIS Event View



accident

I-66 east @ MM 57.0 - EB 66 EO RT 50 EXIT 57 : 3VEHICLES,RIGHT SHOULDER,BLK HONDA VS

Time: 11/29/2009 05:32:43 PM

Lane Status: East

Incident Timeline

- Accident	- Incident	- Injury Accident
- Disabled	- Vehicle Fire	- Collision
- Congestion	- Roadwork	- Signal System
- Debris	- Police Activity	- Special Event

Weather Radar: Light … Heavy

Alert - Alerts issued for a county; - Traffic Cameras

Traffic Speed: 0-30 mph; 30-50 mph; > 50 mph; Current Data Unavailable

Hide Legend

NAVTEQ Detector

I-295/DC-295 @ 0.13 Mile South Of Benning Rd

Direction:	North
Detector Type:	Microwave Radar
Updated:	11/29/2009 06:02:12 PM
Avg. Speed:	49 Mph
Volume:	11 Vehicles per Min

Figure 3: RITIS Map View

Figure 2 shows an example of the Event View in RITIS. Events and Incidents are listed in a tabular fashion and the most relevant information is presented. The user can view more details about any event by clicking on the “Display” link in the “Timeline” column.

Figure 3 depicts the Map View. This view overlays several layers of information on top of a map: traffic speed/volume information, incidents and planned events, weather conditions, etc. The map elements are interactive and they open more details when clicked. In the figure one can see windows open about an accident on I-66 East and a Navteq Traffic Detector’s data.

3.1 System Components

Figure 4 depicts the various System Components that make up RITIS. Data is received in various forms from the agencies providing RITIS with real-time information. For each custom data feed, a loader interprets the data, converts it into a common, ITS standards compliant format (CATT XML) and finally it loads the data into a shared database. Then CATT XML updates are sent to agencies that request them. In the meantime, the visualization tools query the database to present either real-time or historical views of the collected data.

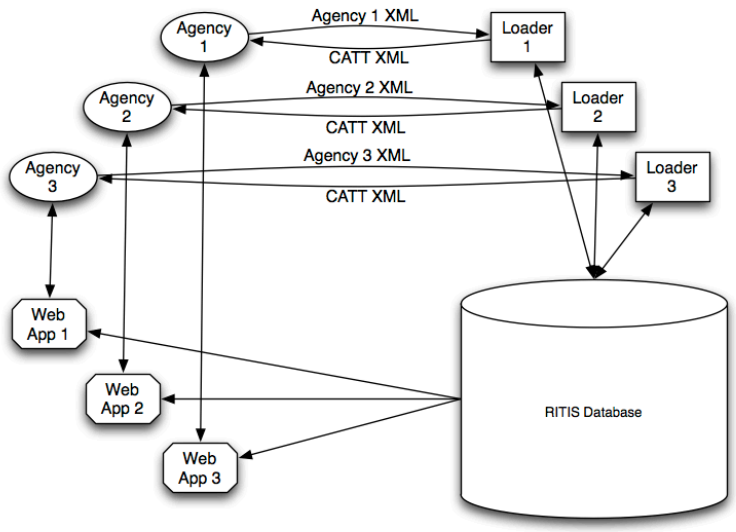


Figure 4: RITIS System Components Overview

3.2 Data Feeds

The data feeds that RITIS receives from the agencies are very heterogeneous. For example, traffic information from road traffic sensors can contain several hundreds of data elements per second, each element being composed of two data values: speed and volume. On the other hand, road incident updates come on the order of minutes, but contain various pieces of information, such as the number of cars involved in an incident, the type of cars, the number of responders at the scene, the time of responder arrival and departure, contact information of notified people, road conditions, lane configuration at the scene of the incident, lane closure information, various responder chat logs, and many more.

Moreover, the heterogeneity of data feeds extends across different agencies for the same type of data. For example, in the case of incident data, one agency might provide only the number of cars involved, while another might actually give a description of each car, including car make and model, color, license plate numbers, type of vehicle, number of persons in each vehicle, etc.

Because most of sensor data is relatively simple and repetitive, various easy to parse formats are being used by the different agencies providing data feeds to RITIS. However, these formats do not usually have automatic parsers already built for them. One has to write a data feed parser from scratch for every new sensor data format, but because of the format's simplicity, it is usually not difficult to do so.

On the other hand, incident and planned event data feeds, contain various pieces of information and not all types of information are usually available at any given point in time. Indeed, some of the types of information that can be recorded may not be available at all during the whole lifetime of an incident. For example, when an incident is reported, an operator working for an agency responsible with the incident gets notified. The operator might create a new incident in the agency's system, and the new incident might contain just the number of vehicles involved and the location of the incident. Lane configuration might not be known at this time, and certainly lane closure information will not be known until a responder arrives on scene. Later, during the progress of the incident, updates regarding these pieces of information may become available. Some types of information may never reach into the final incident data feed. For example, if a Medevac helicopter was never needed at the scene of the incident, then the information about the Medevac will never appear in the respective incident.

Because of the various possible types of information that an agency might keep track regarding incidents and planned events, most agencies opt for an XML representation of the incident and planned event data feeds. Nowadays, most governmental agencies also provide RITIS with an XML schema containing a description of the possible values in the data feed. These XML schemas vary wildly both in terms of content and amount of detail from agency system to agency system.

Thus, in order to accommodate the various forms of data, RITIS employs a PostgreSQL database that contains a common RITIS database schema, as well as agency/system specific extended schemas. The common RITIS database schema captures information that is common to all systems. All the common information from incidents and planned events is stored in the common database schema and is thus consistent from system to system. The system specific extended schemas are designed in such a way that they inherit every relation from the RITIS schema and add new relations and/or extend the inherited relations with agency specific columns. The system is easily extensible with new agencies and schemas.

3.3 Incident Loader

A loader is a piece of software that receives information from an agency and loads it into the RITIS database in the proper schemas. A generic loader has several tasks to perform depending on the kind of data it is supposed to load in the database. Because traffic information loaders receive simple types of very repetitive data, they do not make the focus of this paper. From now on, whenever we talk about a loader we refer to incident and planned event loaders that deal exclusively with XML data. Thus, the main tasks of a loader, in the order in which they should be executed, are:

1. Connect to the XML data feed from a given agency system. Download or receive latest data,
2. Convert agency specific schema compliant XML to RITIS specific schema compliant XML (called CATT XML),
3. Load the data into the agency specific database schema,
4. Send converted CATT XML message update back to listening agencies.

As we shall see in the following Section, Task 2 is the most time consuming task for a software developer who is already familiar with the RITIS loader development. However, Tasks 1, 3 and 4 take an equal, if not longer amount of time than Task 2 for a software developer who is not familiar with the technologies involved and who has never designed a RITIS loader in the past. Since most of the developers at CATT Lab are newly hired undergraduate or graduate students, more often than not, they fall into the latter category of developer and thus there is much time being spent on getting accustomed with the technologies and the design required for programming such a piece of software, instead of solving the intricate problem of translation from one system schema to the RITIS common schema, i.e. solving Task 2.

4 Problem

As we have seen in Section 3.3, there are four tasks that need to be accomplished by a RITIS loader. The loader is composed of several components to help it finish the tasks.

4.1 Typical Components of a classic RITIS Loader

Figure 5 depicts the typical components of a classic RITIS Loader.

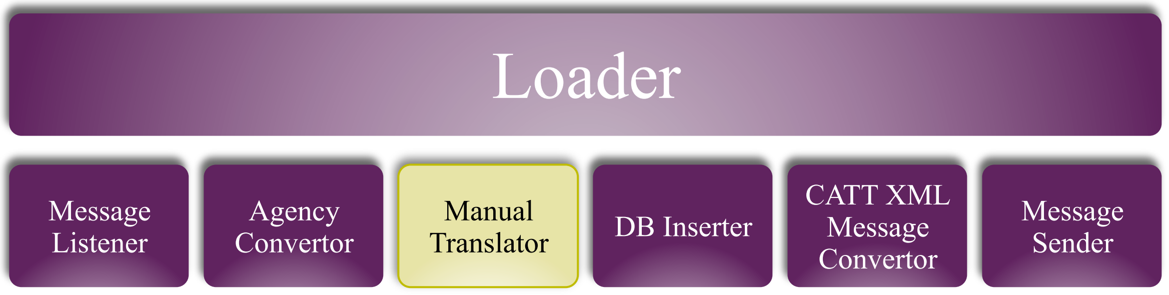


Figure 5: Components of a Classic RITIS Loader

Each component contributes to one of the tasks. For example, the Message Listener is responsible for Task 1, connecting to and receiving or downloading the data from an agency. The Agency Convertor is responsible with the first part of Task 2, converting the data from the agency XML schema to an internal object-oriented format understandable by the loader. The Manual Translator, responsible with the second part of Task 2, transforms the agency data into the required RITIS format. At this point in time the data is still in an object-oriented format inside the loader's memory. The DB Inserter takes the RITIS formatted data and inserts it into the RITIS database, thus contributing to Task 3. The CATT XML Message Convertor converts the object-oriented representation of the RITIS data into the CATT XML format, and, finally, the Message Sender is responsible with executing Task 4, sending the CATT XML as a message update to listening agencies. Out of all these components, the highlighted component, the Manual Translator, is the only component that is fundamentally different from agency to agency. The rest of the components are relatively similar.

This is a functional overview of a loader. However, we point out that we have presented this particular division of functional components after we have done the research to identify these components in the first place. Before the RITIS Development Framework was created, a software developer would mostly program the loader as one monolithic application, with little thought for such functional components.

4.2 Traditional RITIS Loader Development Process

Since most of the classic RITIS Loader applications were developed from scratch, with little chance of identifying any common functional components among the early loaders, the software developers were forced to re-invent functionalities that already existed in previous loaders. It was very difficult for any developer to go back to a finished and deployed loader and identify possible pieces of code or components that could have been reused in their own application.

Thus, we identified several problems with the traditional development process (Figure 6):

- Wasted development time and effort: Developing each loader would require an investment of time and effort in reinventing several very similar functional components.
- Extra maintainability effort: Having several monolithic and independent projects to maintain makes it difficult to identify and fix potential and real problems in an environment that demands 24/7 availability (the RITIS website aims to provide its services non-stop around the clock).
- Extra learning/training time: Because the CATT Lab's software is mostly developed by students who work part time and because the workforce changes very often, the developers need time to learn the technologies and APIs involved in developing a loader, or to understand the used technologies in an older loader that they have to maintain. Having no consistency among loaders increases training time as there are likely few people familiar with old loader technologies, and also learning time, since when one is done working on a loader, one has to learn new parts of a different loader that they have to work on.

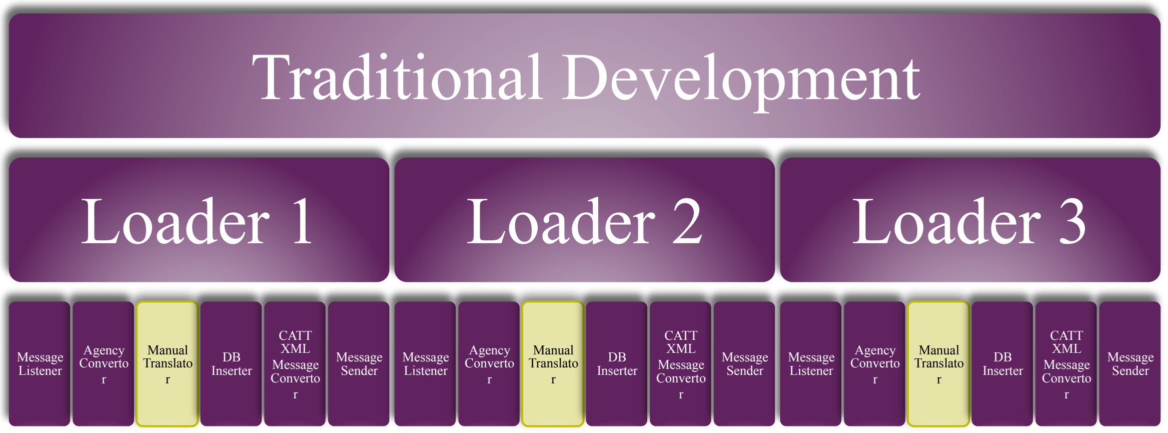


Figure 6: Traditional Development Process. Every loader is developed independently of one another.

5 Solution

The RITIS Development Framework was conceived to alleviate the problems identified in Section 4.2.

5.1 Revised Development Process

Figure 7 contains the overview of the RITIS Development Framework. As can be seen from the figure, the Framework provides predefined components for most of the parts that make up a loader. With little amounts of coding and configuration, a developer can create customized versions of the Message Listener, Agency Convertor, DB Inserter, CATT XML Message Convertor and Message Sender, and afterwards the developer can concentrate their efforts on the translator.

Moreover, even the translator has an abstract template defined for it in the Framework, the Manual Translator Template. The Translator Template provides Java classes and interfaces as a guideline for the developer to brake down the translation process into several components. Please refer to Appendix A for a Java based UML diagram that describes in detail the various classes that make up the Framework.

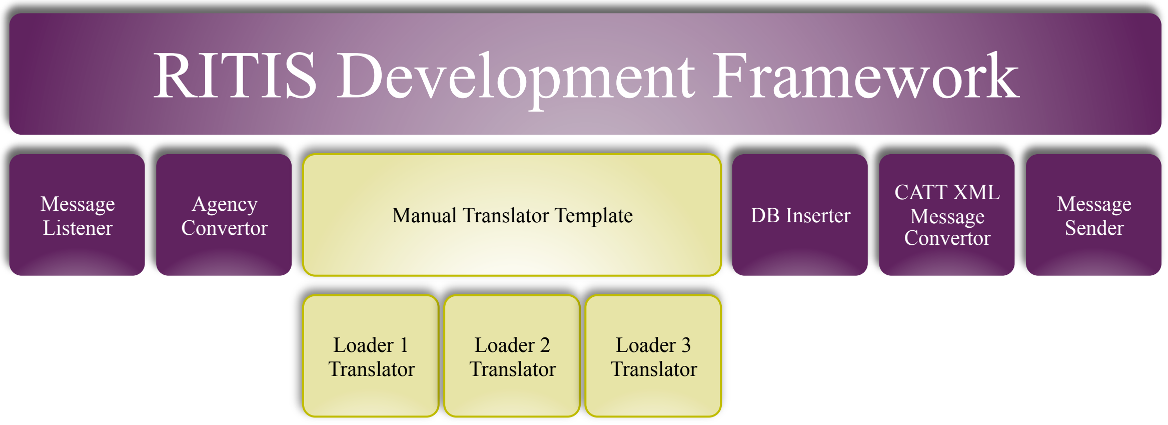


Figure 7: Revised Development Process. Developers concentrate their efforts on the Manual Translator component; the other components are reused from the framework.

This design tries to address the problems mentioned earlier in the following ways:

- Development time and effort: By providing a framework and reusable and configurable code and classes, the developers do not have to spend time on architecting their loader and major components, and thus have more time to develop, test and deploy the translator part component.
- Maintainability: Every loader follows the same design principles and components, thus the framework makes it easy for anyone familiar with it to look into any loader and fix problems or upgrade functionality.
- Learning/training time: The time spent on understanding the framework can be made up for by offloading the developer's attention from all the technologies used behind the scenes by every classic loader. Moreover, once a developer is used to the framework, the knowledge gained can be used in any loader within RITIS.

5.2 About the Implementation

The RITIS Development Framework, as well as all RITIS loaders, is programmed in Java 1.6. Some of the consistency improvements that we were able to achieve were also a result of the underlying technologies the framework uses, as well as the particular way the framework and the loaders using it are deployed at runtime.

At the base of the configuration of every RITIS loader stands an Inversion of Control framework, called Spring Framework [14]. This framework allows all developers to use a common and consistent, XML based method to write the configuration and instantiation of classes that make up a loader. This mechanism makes it very easy to reuse the Framework's components and classes with minimal programming effort from the developer. We shall now address the details of implementation of the various components.

The Message Listener and the Message Sender components are using the Java Message System (JMS) to respectively receive and send messages containing XML from and to the agencies [15]. To support this communication one or more ActiveMQ message brokers is used as an intermediary between the loader and the agencies [16]. The framework contains all the classes required to communicate with ActiveMQ brokers and a developer would just have to instantiate and configure these classes in the XML configuration using the Spring Framework.

Both the Agency Convertor and the CATT XML Message Convertor components make use of the Java Architecture for XML Binding (JAXB). JAXB is a DOM-based technology used for converting data back and forth between W3C compliant XML and a Java object model inside the loader's memory, thus making it painless for a Java developer to work with XML data just by using familiar Java objects instead of parsing the XML manually inside the program [10].

The DB Inserter component makes use of the Hibernate framework [11]. Hibernate allows programs to connect to and work with data in relational databases by using an object-oriented abstraction for tables, rows and columns. It makes use of so-called Hibernate mapping files, which describe mappings between various Java classes and the corresponding tables in a database. Moreover, the mapping is fully portable from one supported DBMS vendor to any other, with minimal or no changes.

All the Java classes of the RITIS Development Framework are packaged in a JAR file, which is distributed to every loader developer and is also included in the runtime environment of the RITIS loader. Every loader also has its own development environment set up, where the source files and the configuration files reside. This structured environment allows for consistency among the locations and internal organization of source files and configuration files across all loaders.

6 Evaluation

We worked on the RITIS Development Framework starting in the summer of 2008 and ending towards December of the same year. In parallel, various software developers at CATT Lab started using the Framework (or parts of it) for the development of their RITIS loaders in August 2008, and continued using it until the time of this writing.

During this period, the CATT Lab's Senior Developer, Jason Ellison, pointed out various positive aspects about the Framework:

- A guideline for RITIS loader development: The RITIS Development Framework is at the very least a guideline to follow in terms of RITIS loader design. It provides development consistency among various loaders, allowing different developers to work on different loaders more easily than was possible before the Framework existed.
- Time spent on design is cut down to a minimum: Developers do not have to architecture their loaders from scratch. The major components are already laid out for them. They just need to use them, maybe modify them and configure them.
- Smaller retraining/relearning time: A developer who is familiar with the framework and loader design, can more easily move on to a different loader and start working on improving it much faster than before the Framework existed. This is achieved on two fronts: the internal structure of a loader and the technologies being used. By using a common set of underlying technologies, one has to learn them only once as they move from loader to loader.

On the other hand, several negative aspects were identified:

- High RITIS loader dependency on Framework revisions: If a bug exists in the Framework, then it makes it into all loaders dependent on the Framework. After the Framework is fixed, all loaders using it have to be retested with the new revision of the framework. Many times, a particular revision of the framework (which contains the identified bug) together with several loaders are already in production systems, thus it can take significant time until the productions systems, together with the Framework and the loaders, get fixed. Moreover, by modifying the framework, one can introduce new bugs in otherwise functioning loaders that depend on it. On the other hand there is also the potential trade-off that if a bug is fixed in the Framework, then it gets fixed in all loaders. However, in practice, this potential advantage does not seem to overcome the former problem of inter-dependency. In conclusion, maintainability is compromised, however we try to address this issue in the Future Work section.

- Coordination problems: During the common period of Framework and RITIS loader development, we identified some coordination problems with having such an approach. Whenever a new revision of the Framework was ready, it had to be included in the development environment of the RITIS loaders, in order for them to take advantage of the latest bug fixes and new features presents. This created some queuing problems that did not exist before the Framework was developed. Before the Framework existed, every loader was completely independent of each other and thus every developer would take ownership of their loader and build it in parallel with other developers' loaders. When we started using the Framework, there were times when developers identified certain feature requests or bugs in the Framework. Thus, they were forced to wait until these requests were fulfilled before they were able to continue working on the particular component that required the improvements made. Moreover, as new revisions of the Framework were released, developers who did not even have a problem with using older revisions were forced to integrate their code with the new revisions, thus cutting down on their productivity.

Even though the two problems mentioned here seem to have a lot of space devoted to them in comparison to the positive aspects, overall the RITIS Development Framework was considered a success. It allowed the CATT Lab to standardize the loader development process and spend less time training and teaching new employees about the development process, the underlying technologies and loader design principles, while spending more time getting new data sources from new agencies integrated with RITIS.

7 Future Work

We have already identified some areas that require future addressing. More specifically, we believe that more work needs to be done in tweaking the RITIS development process within CATT Lab in order to better accommodate loader software developers with respect to RITIS Framework changes. We suggest adopting a separate internal development cycle for the Framework in order to achieve that.

Furthermore, the RITIS Framework can be improved by including more preset classes that would help several other kinds of loaders. For example there can be further abstractions defined for connecting to HTTP data sources (as opposed to JMS), dealing with other types of data such as RSS Feeds or Comma Separated Value (CSV) files and allowing various ways of persistence in the database such as simple JDBC queries.

8 Conclusions

In the field of Intelligent Transportation Systems, CATT Lab is a pioneer at integrating data from various governmental and private agencies. During the integration process, the CATT Lab software developers face challenges of varying degrees of complexities in adapting the originating agency's data format to the internal RITIS format. These challenges differ in terms of complexity, technologies involved, amount of stability and redundancy required, and one's prior knowledge of similar problems and systems.

The RITIS Development Framework is an answer to an increasing amount of similar but slightly different challenges that RITIS loader developers were facing as CATT Lab was receiving more and more data sources to integrate. The Framework has succeeded in improving several aspects of loader development, but as it started being used more within the organization, we also identified several problems with it. In either case, the Framework has reached its minimum goal: to become a guideline for loader development in integrating legacy data from across different systems. While some aspects can use some improvement, we hope that future developers will take initiative in implementing our suggested changes and thus provide an even better product in an emerging field.

9 References

[1] Bocquet, Aurelien, and Christophe Gransart. "Multi-model architecture for ITS software design improvements." *7th International Conference on ITS Telecommunications.* Côte d'Azur, France: IEEE, 2007. 1-6.

[2] Center for Advanced Transportation Technology Laboratory, University of Maryland, College Park. *CATT Laboratory.* 2009. http://www.cattlab.umd.edu/ (accessed December 1, 2009).

[3] —. *Regional Integrated Transportation Information System.* 2009. https://www.ritis.org/ (accessed November 29, 2009).

[4] Choi, H, Y Choi, and K Yeom. "An Approach To Software Analysis and Design Based on Distributed Components for Intelligent Transportation Systems (ITS)." *ISIE.* South Korea: IEEE, 2001. 649-654.

[5] Heistracher, T, R Widmann, B Stadlmann, and J Zellinger. "An Event-based Communication Environment for Multi-Modal Traffic Information Systems." *Intelligent Transportation Systems Conference.* Toronto, Canada: IEEE, 2006. 1214-1219.

[6] Kelly, S M, J W Myre, Price M, E D Russell, and D W Scott. "Configurable, Object-Oriented, Transportation System Software Framework." (Sandia National Laboratories, Department of Energy) August 2000.

[7] Kiel, D E. "Design and Implementation of a Prototype Multi-Agency Transportation Information System in Pennsylvania." Edited by D D Moyer and T Ries. *Proceedings of the Ninth Symposium on Geographic Information Systems for Transportation (GIS-T)* (American Association of State Highway and Transportation Officials), March 1996: 111-122.

[8] Lussier, R, and J H Wu. "Development of a Data Exchange Protocol Between EMME/2 and ARCINFO." *Procedeeings of 1997 ESRI user Converence* (Environmental Systems Research Institute), July 1997.

[9] Ning, B, and X Li. "A Comprehensive Information System for Railway Networks." *Publication of: WIT Press* (WIT Press), 2002: 897-904.

[10] Ort, Ed, and Bhakti Mehta. "Java Architecture for XML Binding (JAXB)." *Sun Developer Network (SDN).* March 2003. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/ (accessed December 1, 2009).

[11] Red Hat Middleware, LLC. *Hibernate.org.* 2009. https://www.hibernate.org/ (accessed December 1, 2009).

[12] Research and Innovative Technology Administration. "RITA US Frequently Asked Questions." *Research and Innovative Technology Administration.* http://www.its.dot.gov/faqs.htm (accessed December 1, 2009).

[13] Scherer, W T. "The Development of Integrated Transportation Systems Management (ITSM) in Northern Virginia." *Proceedings of Merging the Transportation and*

Communications Revolutions. Abstracts for ITS America Seventh Annual Meeting and Exposition (ITS America), June 1997.

[14] SpringSource. *About Spring.* 2009. http://www.springsource.org/about (accessed December 1, 2009).

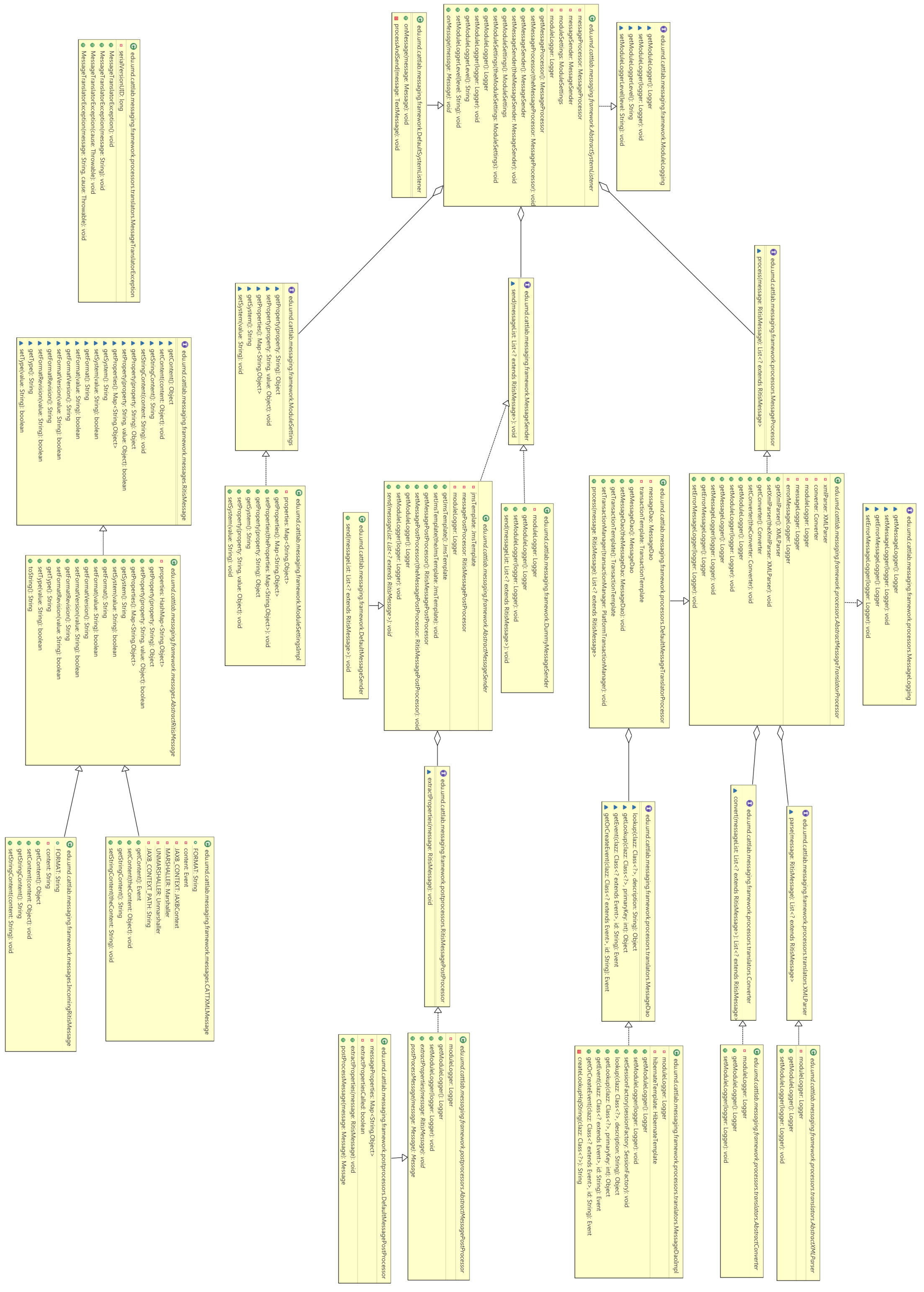
[15] Sun Microsystems, Inc. "Java Message Service (JMS)." *Sun Developer Network (SDN).* 2009. http://java.sun.com/products/jms/ (accessed December 1, 2009).

[16] The Apache Software Foundation. *Apache ActiveMQ.* 2008. http://activemq.apache.org (accessed December 1, 2009).

[17] Yang, Qingyan, Heng Wei, Jeffrey R Manring, and Preeti Agarwal. "XML, SOAP and Web Service: A New Option For ITS C2C Communication and Interoperability." *National Research Council (US) Transportation Research Board.* Berkeley CA: UC Berkeley Transportation Library, 2003. 20p.

[18] Zavattero, David, and Wei Wu. *Design of the Gary-Chicago-Milwaukee Gateway: A Multi-Modal Traveler Information System.* Reston, VA: American Society of Civil Engineers, 2004, 75-88.

Appendix A: RITIS Development Framework UML Diagram


edu.umd.cattlab.messaging.framework.processors.translators.MessageTranslatorException
serialVersionUID: long
MessageTranslatorException(): void
MessageTranslatorException(message: String): void
MessageTranslatorException(cause: Throwable): void
MessageTranslatorException(message: String, cause: Throwable): void
edu.umd.cattlab.messaging.framework.DefaultSystemListener
onMessage(message: Message): void
processAndSend(message: TextMessage): void
edu.umd.cattlab.messaging.framework.AbstractSystemListener
messageProcessor: MessageProcessor
messageSender: MessageSender
moduleSettings: ModuleSettings
moduleLogger: Logger
getMessageProcessor(): MessageProcessor
setMessageProcessor(theMessageProcessor: MessageProcessor): void
getMessageSender(): MessageSender
setMessageSender(theMessageSender: MessageSender): void
getModuleSettings(): ModuleSettings
setModuleSettings(theModuleSettings: ModuleSettings): void
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
getModuleLoggerLevel(): String
setModuleLoggerLevel(level: String): void
onMessage(message: Message): void
edu.umd.cattlab.messaging.framework.ModuleLogging
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
getModuleLoggerLevel(): String
setModuleLoggerLevel(level: String): void
edu.umd.cattlab.messaging.framework.ModuleSettings
getProperty(property: String): Object
setProperty(property: String, value: Object): void
getProperties(): Map<String,Object>
getSystem(): String
setSystem(value: String): void
edu.umd.cattlab.messaging.framework.MessageSender
send(messageList: List<? extends RitisMessage>): void
edu.umd.cattlab.messaging.framework.processors.MessageProcessor
process(message: RitisMessage): List<? extends RitisMessage>
edu.umd.cattlab.messaging.framework.messages.RitisMessage
getContent(): Object
setContent(content: Object): void
getStringContent(): String
setStringContent(content: String): void
getProperty(property: String): Object
setProperty(property: String, value: Object): boolean
getProperties(): Map<String,Object>
getSystem(): String
setSystem(value: String): boolean
getFormat(): String
setFormat(value: String): boolean
getFormatVersion(): String
setFormatVersion(value: String): boolean
getFormatRevision(): String
setFormatRevision(value: String): boolean
getType(): String
setType(value: String): boolean
edu.umd.cattlab.messaging.framework.ModuleSettingsImpl
properties: Map<String,Object>
getProperties(): Map<String,Object>
setProperties(theProperties: Map<String,Object>): void
getProperty(property: String): Object
getSystem(): String
setProperty(property: String, value: Object): void
setSystem(value: String): void
edu.umd.cattlab.messaging.framework.DefaultMessageSender
send(messageList: List<? extends RitisMessage>): void
edu.umd.cattlab.messaging.framework.AbstractMessageSender
jmsTemplate: JmsTemplate
messagePostProcessor: RitisMessagePostProcessor
moduleLogger: Logger
getJmsTemplate(): JmsTemplate
setJmsTemplate(theJmsTemplate: JmsTemplate): void
getMessagePostProcessor(): RitisMessagePostProcessor
setMessagePostProcessor(theMessagePostProcessor: RitisMessagePostProcessor): void
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
send(messageList: List<? extends RitisMessage>): void
edu.umd.cattlab.messaging.framework.DummyMessageSender
moduleLogger: Logger
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
send(messageList: List<? extends RitisMessage>): void
edu.umd.cattlab.messaging.framework.processors.DefaultMessageTranslationProcessor
messageDao: MessageDao
transactionTemplate: TransactionTemplate
getMessageDao(): MessageDao
setMessageDao(theMessageDao: MessageDao): void
getTransactionTemplate(): TransactionTemplate
setTransactionManager(transactionManager: PlatformTransactionManager): void
process(message: RitisMessage): List<? extends RitisMessage>
edu.umd.cattlab.messaging.framework.processors.AbstractMessageTranslatorProcessor
xmlParser: XMLParser
converter: Converter
moduleLogger: Logger
messageLogger: Logger
errorMessageLogger: Logger
getXmlParser(): XMLParser
setXmlParser(theXmlParser: XMLParser): void
getConverter(): Converter
setConverter(theConverter: Converter): void
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
getMessageLogger(): Logger
setMessageLogger(logger: Logger): void
getErrorMessageLogger(): Logger
setErrorMessageLogger(logger: Logger): void
edu.umd.cattlab.messaging.framework.processors.MessageLogging
getMessageLogger(): Logger
setMessageLogger(logger: Logger): void
getErrorMessageLogger(): Logger
setErrorMessageLogger(logger: Logger): void
edu.umd.cattlab.messaging.framework.messages.AbstractRitisMessage
properties: HashMap<String,Object>
getProperty(property: String): Object
setProperty(property: String, value: Object): boolean
getProperties(): Map<String,Object>
getSystem(): String
setSystem(value: String): boolean
getFormat(): String
setFormat(value: String): boolean
getFormatVersion(): String
setFormatVersion(value: String): boolean
getFormatRevision(): String
setFormatRevision(value: String): boolean
getType(): String
setType(value: String): boolean
toString(): String
edu.umd.cattlab.messaging.framework.postprocessors.RitisMessagePostProcessor
extractProperties(message: RitisMessage): void
edu.umd.cattlab.messaging.framework.processors.translators.MessageDao
lookup(clazz: Class<?>, description: String): Object
getLookup(clazz: Class<?>, primaryKey: int): Object
getEvent(clazz: Class<? extends Event>, id: String): Event
getOrCreateEvent(clazz: Class<? extends Event>, id: String): Event
edu.umd.cattlab.messaging.framework.processors.translators.Converter
convert(messageList: List<? extends RitisMessage>): List<? extends RitisMessage>
edu.umd.cattlab.messaging.framework.processors.translators.XMLParser
parse(message: RitisMessage): List<? extends RitisMessage>
edu.umd.cattlab.messaging.framework.messages.IncomingRitisMessage
FORMAT: String
content: String
getContent(): Object
setContent(content: Object): void
getStringContent(): String
setStringContent(content: String): void
edu.umd.cattlab.messaging.framework.messages.CATTXMLMessage
FORMAT: String
content: Event
JAXB_CONTEXT: JAXBContext
MARSHALLER: Marshaller
UNMARSHALLER: Unmarshaller
JAXB_CONTEXT_PATH: String
getContent(): Event
setContent(theContent: Object): void
getStringContent(): String
setStringContent(theContent: String): void
edu.umd.cattlab.messaging.framework.postprocessors.DefaultMessagePostProcessor
messageProperties: Map<String,Object>
extractPropertiesCalled: boolean
extractProperties(message: RitisMessage): void
postProcessMessage(message: Message): Message
edu.umd.cattlab.messaging.framework.postprocessors.AbstractMessagePostProcessor
moduleLogger: Logger
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
extractProperties(message: RitisMessage): void
postProcessMessage(message: Message): Message
edu.umd.cattlab.messaging.framework.processors.translators.MessageDaoImpl
moduleLogger: Logger
hibernateTemplate: HibernateTemplate
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
setSessionFactory(sessionFactory: SessionFactory): void
lookup(clazz: Class<?>, description: String): Object
getLookup(clazz: Class<?>, primaryKey: int): Object
getEvent(clazz: Class<? extends Event>, id: String): Event
getOrCreateEvent(clazz: Class<? extends Event>, id: String): Event
createLookupHql(clazz: Class<?>): String
edu.umd.cattlab.messaging.framework.processors.translators.AbstractConverter
moduleLogger: Logger
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void
edu.umd.cattlab.messaging.framework.processors.translators.AbstractXMLParser
moduleLogger: Logger
getModuleLogger(): Logger
setModuleLogger(logger: Logger): void