Recondo 1: PRS Project Rehearsal Final Report

By
Oliver Chase
Mitchell Wright

Table of Contents

Abstract 3
Introduction 3
Requirements 4
    Functional 4
    Nonfunctional 5
Scope and Project Progression 5
System Design 6
    Sender 7
    Analyzer 9
Implementation Details 17
Results 18
Conclusions and Future Directions 18
Glossary 19

Abstract

The process of testing the connectivity of a hospital's payers to the *Recondo Payer Resolution System (PRS)* is manual and slow. With recent sales and company growth, there are so many pending activations that the company's development resources would be tied up for days using the established manual approach. The existing process is also inaccurate as developers are visually scanning for errors.

Recondo needs a way to automate these tests in order to free up developer resources and improve the speed and accuracy of activation testing. The purpose of this project was to design and implement a tool to automate these tests and integrate the tool with Recondo's existing code base.

The presented JAVA solution is a two-tiered system consisting of a command line program and an always running service. The command line program packages information for test submission and then turns over control to the service. The service then finds and formats each request and posts the request to Recondo's existing system, PRS. It then receives responses from PRS and summarizes them into a simple report for the developers. Rehearsal allows the developer to simply provide the background information for the tests and then receive a report after all tests have finished. This solution is expected to dramatically improve the speed and accuracy of activation testing, reducing the cost and developer time spent on activation testing.

Introduction

Recondo provides real-time cost information to hospitals and patients nationwide. Recondo's goal is to tell patients the exact out-of-pocket cost they will need to pay to receive medical services, allowing the patient to make informed decisions regarding their medical care and helping hospitals receive payment for their services.

Before a hospital can use Recondo's services, Recondo must test connectivity to the list of *payers* that the hospital requires. A *payer* is an entity (such as an insurance company) that pays for some or all of a patient's medical services. The process of bringing a new hospital online and testing its connections is called *activation testing*.

Current activation testing requires that a developer perform a number (sometimes, a very large number) of manual tests. The developer must manually form a valid request from a real person's health information, known as *protected health information (PHI)*, manually make adjustments to the request, submit it to the Stage PRS system, wait for the response, and visually scan the response to see if the connection is viable or not. This process can be slow and error prone, especially when a very large number of transactions must be sent to the system.

Recondo would like a tool to help with activation testing. The provided tool has been named *Rehearsal*, as it will be used in one of the final stages of bringing new hospitals into the Recondo services. Compared with the current repetitive and error-

prone manual method, Rehearsal will reduce developer work hours and increase the accuracy of testing activations of hospitals by automating the activation test process.

Requirements

Rehearsal will be designed and implemented as two separate modules. The first, the **Sender**, will collect transaction cost data, validate the NPI, and submit necessary data to the second module. This second module, the **Analyzer**, will run as a service and send requests to Apache ActiveMQ for processing. It will then listen for completed test transactions, analyze them, and generate a report.

Functional Requirements

Configuration

1. There will be a system by which Rehearsal can be configured. The goal is to make everything that can be configured configurable. Good interfaces should be provided for future development.

Connections

1. Rehearsal will be able to connect to three different data sources: Stage PRS through ActiveMQ, the txLogger and PRS Config databases through PostGres, and to CSV files.
    a. *txLogger* holds a record of all transactions that go through Stage PRS and will be used for analyzing responses
    b. *PRS Config* will be used for cost estimation and NPI validation
2. When parsing CSV files, Rehearsal will remove duplication in the transactions it will send.

Analysis

1. Most transactions incur a fee each time Recondo connects and runs a transaction. Rehearsal will calculate the estimated cost of the test batch before sending any transactions and await approval from the developer before continuing.
2. Rehearsal will analyze and record failures according to the protocol provided by the client. Failing tests must indicate various levels of error, such as a color-coded system. A red (or similarly labeled) failure is one that indicates a critical problem that will not clear up with time, while an orange or yellow failure might be due to circumstances beyond Recondo's control. An orange failure will always notify the developer. A yellow failure will only notify the developer if a certain threshold of failures (percentage of failures for the total test) occurs.
3. Rehearsal will report the actual cost of the test in its report.

Sender-specific

1. Sender will accept either an input directory, output directory, *NPI* (National Provider Identifier) tuple, and a path to the PHI to use, or a tuple containing a UUID of a past transaction and an output directory for a regenerated report.
2. Sender will handle the calculating of expected costs and the validation of the given NPI.
3. Sender will assign each battery of tests a UUID upon runtime.
4. Sender will pass all relevant information to the tests to the Analyzer.
5. The Sender will have a command line interface for use.

Analyzer-*specific*

1. Analyzer will operate as a service listening for requests.
2. Analyzer will create a report of the tests when each transaction of a given UUID has completed.
3. Analyzer will run the tests for different Recondo products in the following order: SPH first, EligibilityPlus second, and AuthDP third.
4. Analyzer will change requests to include the correct NPI and payer before sending.
5. Analyzer will be able to post requests to Stage PRS through ActiveMQ.
6. Analyzer will be able to receive responses corresponding to the requests it sent to Stage PRS through ActiveMQ.
7. Analyzer will assign an error code to all results according to configurable rules provided by the client.
8. Analyzer will generate a report with metrics describing the results of the tests and any errors encountered.

Nonfunctional Requirements

1. The team will have six weeks in total to design and implement Rehearsal.
2. Development will be done on company issued laptops or virtual machines.
3. The program will be well documented and configured for customization so future developers can use the program effectively.
4. The program will be written in JAVA.

Scope and Project Progression

Rehearsal was developed, as with much of Recondo's codebase, with a loosely agile methodology. It was agreed that the design and implementation of the Sender component was the first priority and that the design of the Analyzer would come later. The first week of the term was reserved for meetings, gathering requirements, and learning technologies, so the first actual design of the Sender began during the second week. In addition to designing the Sender, it was largely implemented by the end of the

week. A rough design for the Analyzer was created, but only to satisfy the requirements of the Colorado School of Mines (CSM) course portion of the project, and subject to later change. For example, the Sender was initially designed to send messages all the way to Stage PRS, and then the Analyzer (then called Receiver) would process the results received from Stage PRS. The design changed to have the Analyzer handle both the sending and receiving of messages to Stage PRS. The project's supervisor also later reduced the scope of the project, only requiring us to implement the skeleton of cost verification and NPI validation, awaiting the actual functionality to be implemented by Recondo at a later date. This removed the requirement to connect to the PRS Config database.

During week three, the Sender was finished and the design and implementation was begun on the Analyzer. The biggest difficulty in this week was setting up the communication between the Sender and Analyzer via ActiveMQ. The design of the Analyzer was also simplified, allowing reports to be constructed any time a response is received from stage PRS. Generating incomplete reports after each transaction, instead of complex logic determining whether the entire test set was complete, dramatically simplified the Analyzer's design. Weeks four and five were used to complete the Analyzer.

System Design

Rehearsal has been designed as two modules. The first module, the Sender, is responsible for accepting user input, verifying the input to insure that it is well-formed and valid, and providing a handle into the Analyzer.

The second module, the Analyzer, is responsible for choosing an example patient to perform a test on, transforming this patient's file into the format needed by the existing Recondo architecture, receiving the corresponding response file, and analyzing it according to rules provided by the client. After analysis, the Analyzer will generate a report and save it to disk for the user to review.

The Sender and Analyzer will communicate with each other and the existing Recondo architecture using *Apache ActiveMQ*, which is an open source message brokering system. ActiveMQ is currently being used in Recondo's architecture, and Analyzer will interface directly with the existing framework.

Figure 1.1 illustrates a very high-level view of how Rehearsal will work and interface with existing Recondo architecture.

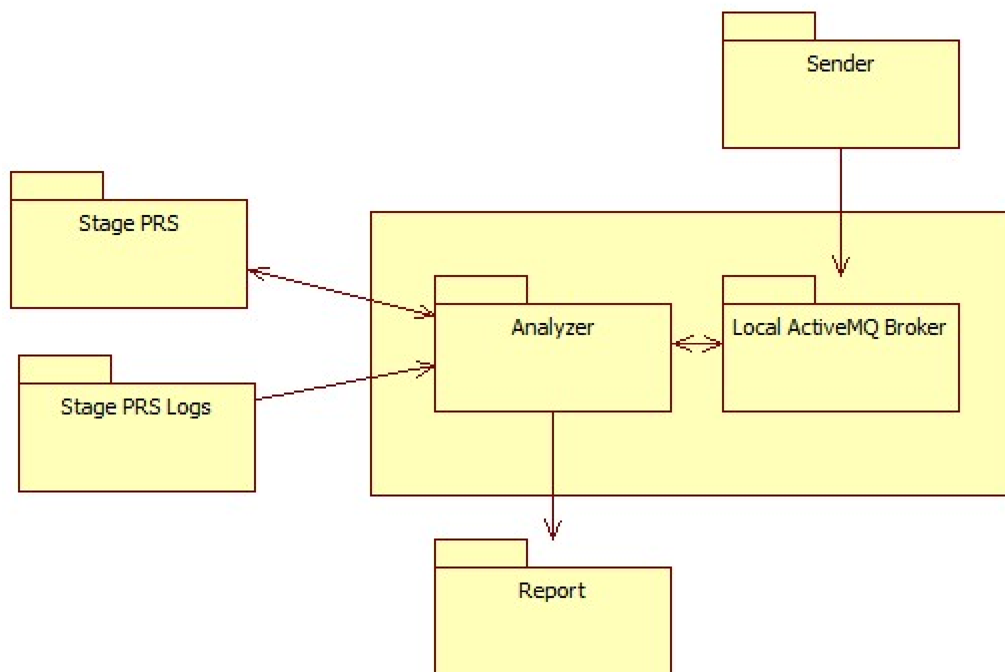

Figure 1.1: High-level Rehearsal Architecture

The Sender posts SenderMetadata objects to a message broker (the ActiveMQ Broker) running on the same machine as the Analyzer service. The SenderMetadata can be best described as an expression of the user's intent, containing information to both describe what the intent is and how the intent is to be satisfied. Analyzer will consume the SenderMetadata message and create a Thread, loading all the information that the intent requires into it. This thread is a self-contained environment in which it is possible to run the test. Throughout the course of the test's execution, the Thread will use a number of tools provided by Analyzer that allows it to interface with the existing Recondo system, sending and receiving data to complete its test. Stage PRS provides the responses that Analyzer bases its conclusions on through its txLogger database, which contains a record of all the transactions run through PRS. When Analyzer has completed a test, the Thread associated with the test writes a report and then is killed.

Sender

The Sender will operate as a command-line program that can accept two different sets of input. The first is a UUID (Universally unique identifier) and an output path that corresponds to a test batch that was run sometime in the past and recreates a report for that test batch. The second input set is a national provider identifier (NPI), an input file path for a comma separated (CSV) file containing a list of payers for that NPI, an input path for a directory containing *270* request files, which are the files that contain the patient information used in a test, and an output path for the report to be generated.

Sender is responsible for verifying either of these tuples passed in by the user and passing certain information to the Analyzer module. In the NPI-payer input-request input-output case, the Sender checks if the NPI is formed correctly and checks to see if the payer input file exists. There is also skeletal behavior for cost analysis and NPI verification. If all of these conditions are true, Sender sends the NPI, a generated UUID, a list of PayerAndProduct instances, the input path to the request files, and an output path down to the Analyzer packaged in a ReportMetadata instance. Analyzer will use this object to find and format the necessary input files (each Payer instance requires a particular input file).

If a UUID and output path are passed into Sender, the UUID and path will be packaged in a UuidMetadata instance that lacks a list of Payer instances. This object will be used in Analyzer to generate a report using transaction logs. Figure 2.1 below shows the structure of the Sender module.

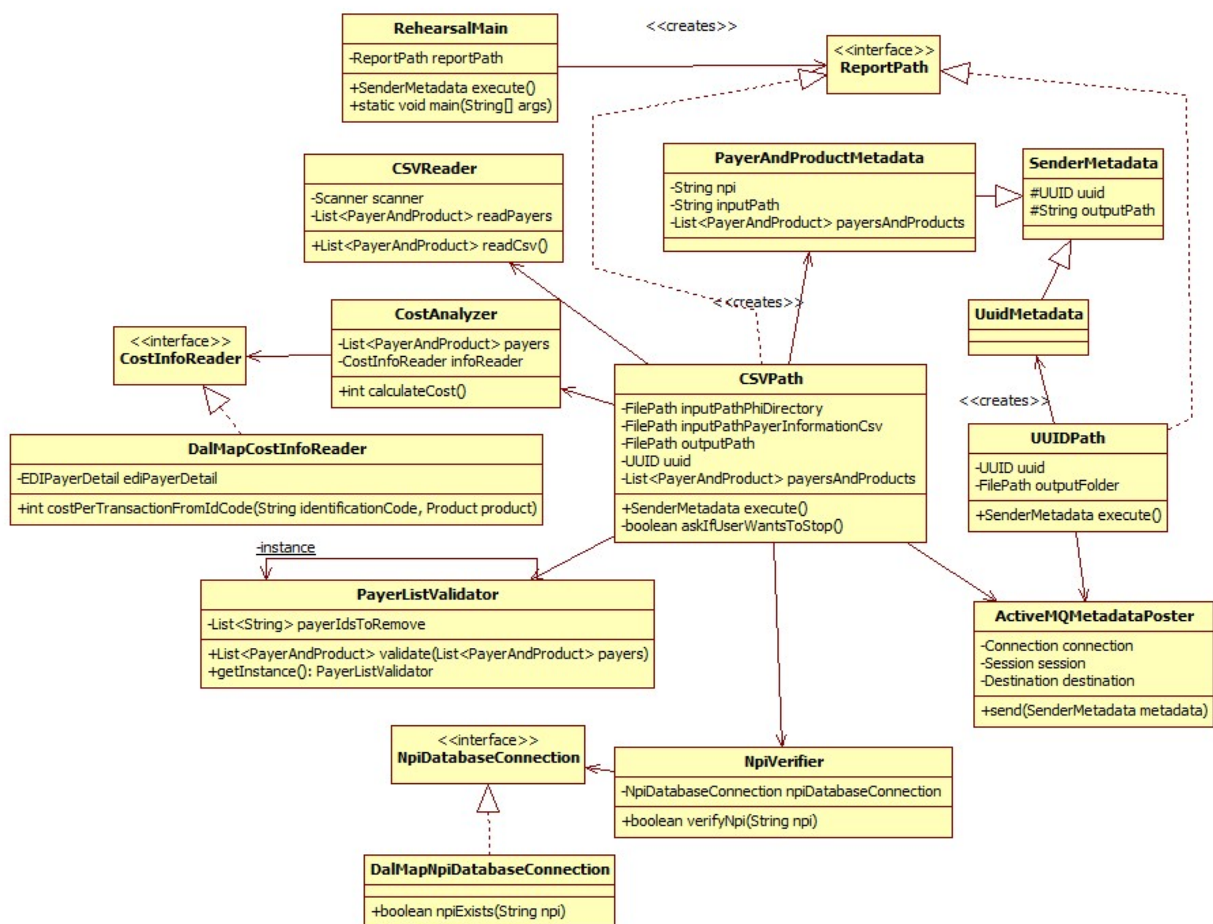

Figure 2.1: Sender Design

The entry point of the Sender module is the “RehearsalMain” class in Figure 2.1. It is this class that the user will enter his or her arguments in order to run an activation test or regenerate a previously run activation test.

RehearsalMain parses the input it receives into one of two categories. The first corresponds to an input tuple of an NPI, input path to a comma-seperated-value file (CSV) containing the payer identification codes for all the payers desired by the provider with that NPI, an input path to the directory containing the PHI request files, and a path that the user would like the report to be written to. This input tuple is parsed as a full activation test, and RehearsalMain will create a CSVPath object for this behavior. Likewise, an input tuple of a UUID and output path is the necessary input to regenerate a previously-run test’s report, and so creates a UuidPath instance. The CSVPath and UuidPath in the figure both realize, or implement, the ReportPath interface. The goal is to encapsulate the varying behaviors of the full activation test and the report regeneration so that RehearsalMain can remain flexible and simple. This is an example of the Strategy design pattern.

If a CSVPath is executed, it will construct a PayerAndProductMetadata instance and fill it with appropriate payer identification codes, then post it to ActiveMQ. If a UuidPath is executed, it will create a new UuidMetadata instance, add the UUID and output path, and likewise post it to ActiveMQ.

Analyzer

Analyzer’s responsibilities include receiving metadata from Sender via ActiveMQ, finding and formatting a relevant 270 request for each Payer in that metadata, sending the request to the existing Stage PRS system, receiving the corresponding response, analyzing it, and generating the report for each batch of responses.

In order to receive metadata from Sender, Analyzer contains two event-driven *listener* classes, MQInputListener and MQResponseListener. The former is composed with a handle to the queue that Sender posts its metadata objects to. When ActiveMQ receives a message on this queue, it will notify MQInputListener and call the listener’s *onMessage* method.

When MQInputListener receives a metadata object, it in turn notifies Analyzer and supplies the metadata so that Analyzer can parse it and generate a set of requests. In order to facilitate this behavior, MQInputListener and Analyzer use the *Observer* design pattern, with MQInputListener being the Observable and Analyzer being the Observer. (See the glossary for information on the Observer pattern.) When notified, Analyzer pulls the metadata from the MQInputListener and checks its type, dealing with different types of metadata in different manners.

Each set of input supplied to Sender creates one of two differing types of metadata, either PayerAndProductMetadata or UuidMetadata. If the user enters the information corresponding to a full test run to Sender, a PayerAndProductMetadata

instance is created, and if the user enters only a UUID and an output path, a UuidMetadata instance is the result. Each type of metadata contains all the information that Analyzer needs to generate a report based on the user's intention; for example, if MQInputListener receives a PayerAndProductMetadata, the user intends to run a full test that may incur costs. If MQInputListener receives a UuidMetadata instance, the user's intention is to regenerate a report without running transactions through Recondo's Stage PRS system. Figure 3.1 below shows how Analyzer uses thread pools to organize a multithreaded environment to allow multiple activation tests to proceed concurrently.

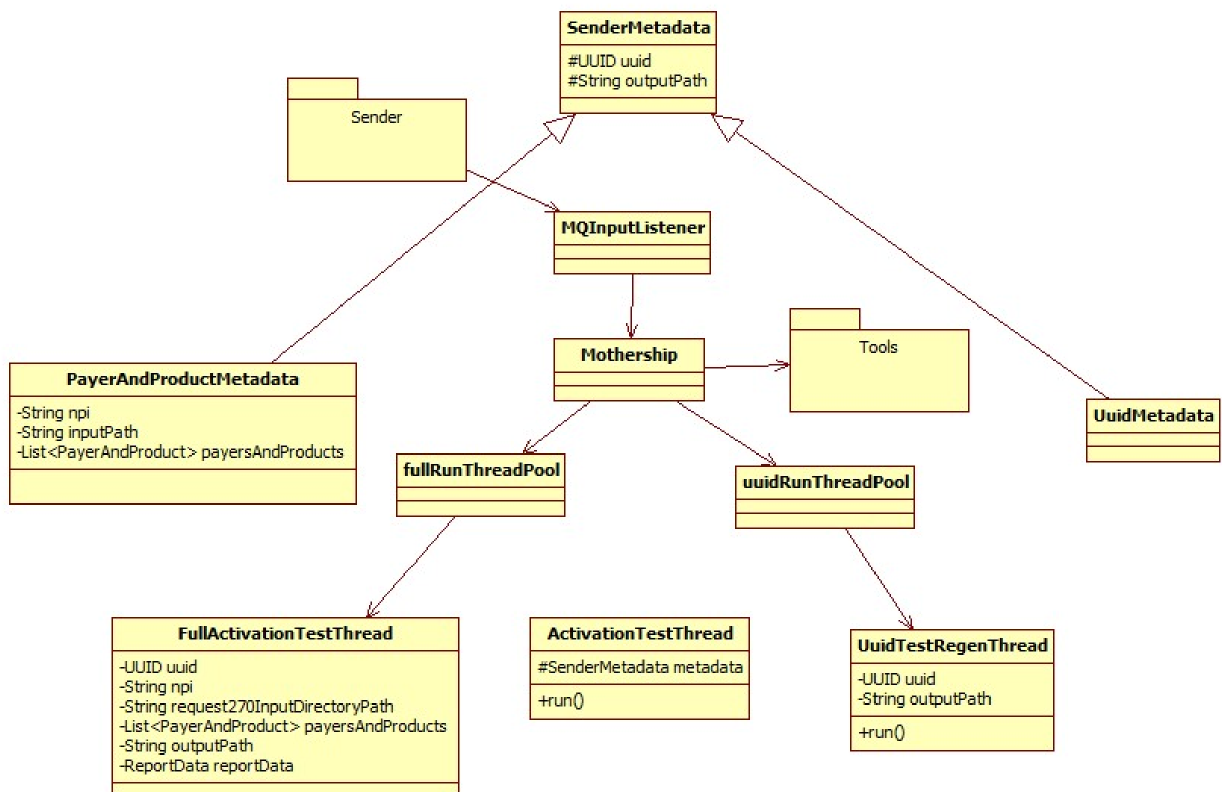

Figure 3.1: Thread pooling -

Analyzer uses a Java thread pool to allow multiple activation tests to be run at the same time. A thread pool is an object that maintains a queue of tasks and a pool of threads to perform those tasks. Thread pools help to organize a multithreaded environment and allow different types of tests to be processed in different ways.

Upon receipt of one of the two types of metadata, Analyzer creates a runnable command instance of one of two types, a FullRunTestThread for a full activation test, and a UuidRegenThread for a simple report generation using already-processed transactions. The FullRunTestThread command knows how to find and transform

requests, post those requests to Stage PRS, receive responses, analyze those responses, and generate a report, retrying requests when necessary. The UuidRegenThread does not need to interface with Stage PRS, so its class only knows how to generate a report. The following description of Analyzer's behavior assumes that the user wishes to run a full activation test.

A full test takes payer identifers and produces industry-standard requests for them using the RequestProcessor tool. The RequestProcessor's job is to take the metadata and form it into a set of requests that Stage PRS can understand. To do so, RequestProcessor depends on an abstract interface, RequestFinder, that defines the needed behavior.

A request consists of a patient information file for the payer to test. This patient file is *protected health information* (PHI) according to federal Health Insurance Portability and Accountability Act (HIPAA) regulations. Recondo receives a set of PHI during the process of signing a new hospital, and the files required by Analyzer are to be provided by the client. The files are XML files in the *DalMap* format, which is a data structure used internally by Recondo's products. The requests are files on disc, and an implementation of RequestFinder called FlatFileRequestFinder handles the file I/O and does not expose it to the rest of Analyzer.

One problem with the requests is that there is currently no way to easily choose a file for a payer, since the file name does not contain a unique payer identifier that Analyzer can look for. Currently, Analyzer assumes that the user has placed all the PHI for a particular payer in a subdirectory labeled with the payer identifier in the payer CSV.

The requests on disk do not have the correct payer identifier in them, and they also lack the correct NPI. To fix this, a class called RequestTransformer parses each request as it is returned by the RequestFinder and adds the NPI and payer identifier. Since the behavior of RequestProcessor and RequestTransformer relies on the use of a single resource (the RequestFinder), the two classes were implemented using the *Singleton* pattern to avoid the instantiation of multiple RequestProcessors or RequestTransformers.

After the metadata is fully formatted into requests, the Requester posts them to Stage PRS, using a separate queue for each product that Rehearsal supports (Sure Pay Health, EligibilityPlus, and Authorization-Denial Prevention). These queues are already in Recondo's existing system, and Requester interfaces with production and development servers already in operation and posts these requests alongside real-time production requests. Figure 3.2 on the following page shows how the modules for running a full activation test work to get requests, transform them, and send them to Stage PRS.

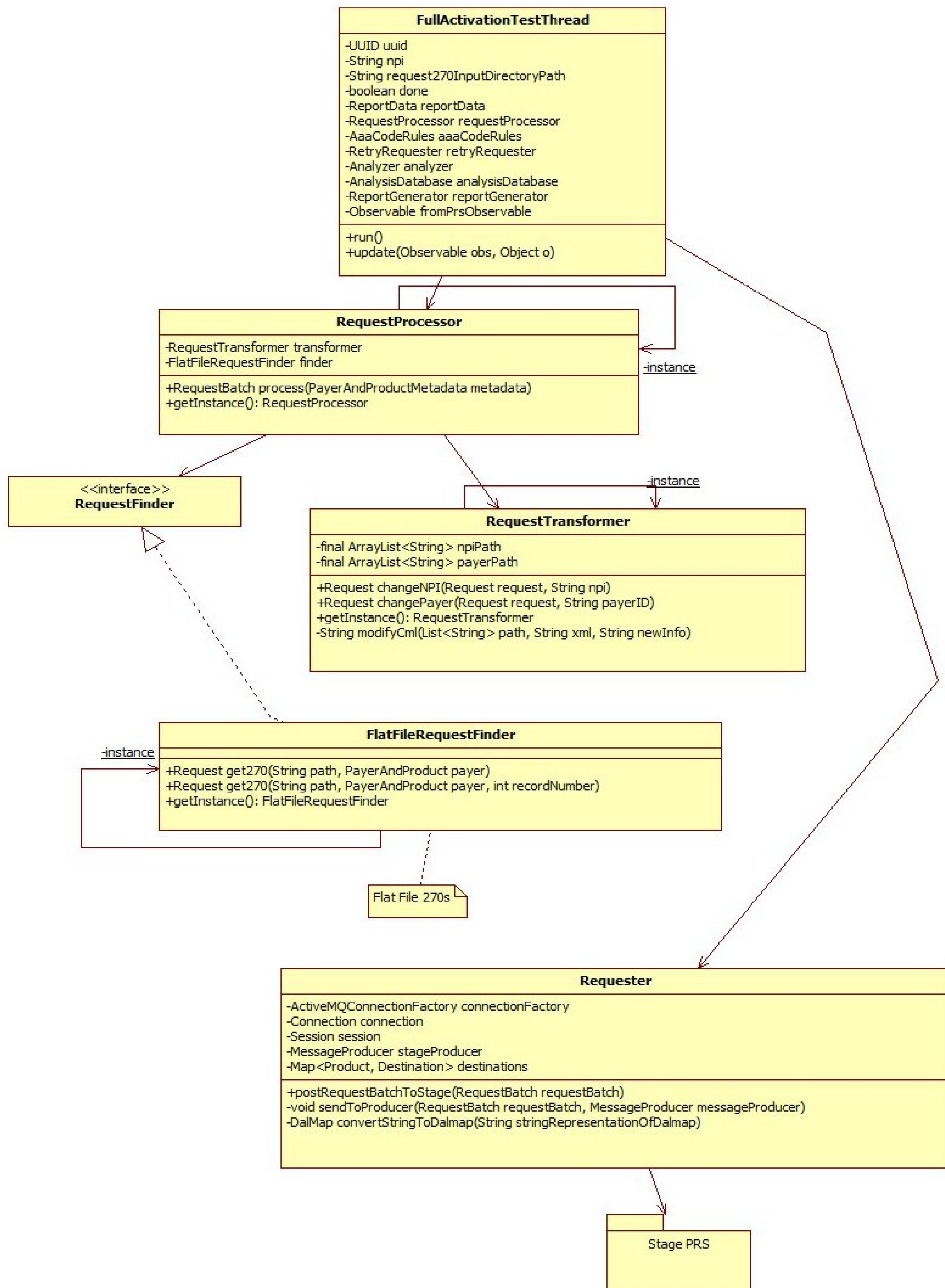

Figure 3.2: Making Requests with Analyzer

Only a full activation test requires finding and transforming requests, so only the FullActivationTestThread needs the ability to deal with requests. The metadata that the FullActivationTestThread received when it was created contains a list of PayerAndProduct instances, which is simply an object that links a payer identification code and a Product, which is an Enum, one value for each product that Recondo needs to run activation tests for. RequestProcessor loops through all the PayerAndProduct instances and uses RequestFinder and RequestTransformer to return a set of well-formatted requests that PRS can process. Then the FullActivationTestThread forwards this list of requests to the Requester, which is the portion of Analyzer that is responsible for interfacing with the existing Stage PRS system through ActiveMQ. Recondo uses a custom data structure for requests, so Requester also has the responsibility to convert from the Rehearsal Request object, which is not the custom class, to an instance of the custom class, a DalMap, so that Stage PRS can read it.

When Stage PRS is finished with a request, it places the corresponding response back on the server running Analyzer, where it is received by MQResponseListener, which is analogous to MQInputListener, but accepts responses from PRS rather than metadata from Sender.

From MQResponseListener, the response is passed to the Analyer module itself. In meetings with the client, there was a major design shift from the earlier strategy of analyzing each response individually when it was received from PRS. Instead, Analyzer now queries a logger, *txLogger*, for *all* of the responses associated with a message from PRS and analyzes all of them at once.

The benefit of this change is that Analyzer can use txLogger to access and derive many useful pieces of information, including how many payer requests were run for a test suite, whether or not a given test suite is complete or is still waiting for responses, and it provides access to the whole test suite's data at once. Before this design change, Analyzer would have to store incomplete test suites itself, and thus would have to have another database, an interface to that database, and additional logic to provide data safety and security. An abstract interface, AnalysisDatabase, provides the necessary behavior used by Analyzer to query txLogger, and a concrete TxLogReader class, borrowed from an existing Recondo tool, provides the implementation. Before being used by Analyzer, the raw database response is mapped into a set of objects by a class called Mapper. If the user's intent was to regenerate a report rather than run a full test, this is where Analyzer would begin, rather than interfacing with Stage PRS.

Analyzer uses a series of rules parsed from a configuration file to determine if a payer passes or fails its test, and these rules are encapsulated in the AaaCodeRules object. In the HIPAA standard, the AAA tag indicates that an error occurred when a response was being generated, and it provides a code to notify the user what kind of problem occurred. Some codes are more serious than others, so a requirement was

that Rehearsal could report different levels of error. The most serious codes result in red errors (the colors are an invention to help understand the severity of an error), which means that execution of a test must be halted because there is no way it will ever succeed (e.g. an invalid NPI). A less serious error is a yellow, and Analyzer will allow a number of yellow errors to occur without reporting that a payer failed (e.g. unable to receive a response), up to a threshold. If the threshold is exceeded, however, the payer is marked as failing and this information is included in the report. For example, Recondo may need to know if the payer failed to send a response many times, as this may suggest a deeper problem. A level of error between a yellow and a red, like an invalid date, is an orange, and is treated in the same manner as a yellow error, but with a special note to the user to indicate the heightened severity. If no error code is found in a response, it is marked as passing. Figure 3.3 below shows how the Analyzer module processes responses into a ReportData model.

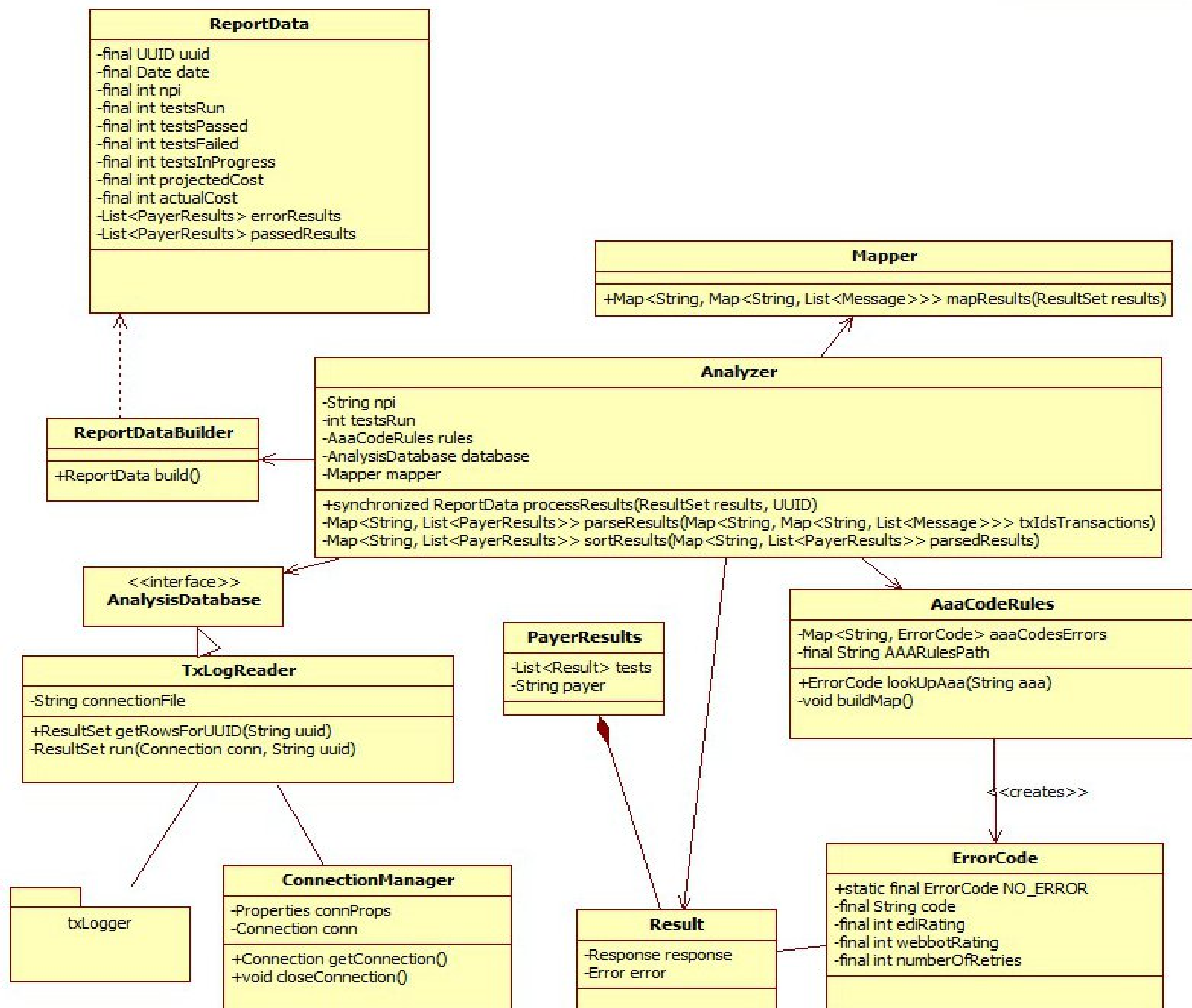

Figure 3.3: Analysis

Analyzer's main function is to take the raw logs from txLogger and produce from them a ReportData object, a model of all the information Recondo needs in a report. In order to access txLogger, Analyzer relies on the AnalysisDatabase interface, which defines the behavior needed by Analyzer to do its job. Rehearsal has borrowed from an existing tool in the implementation of TxLogReader. This class is almost entirely from a tool used by Recondo to analyze txLogger using a GUI interface, tweaked a bit in order to allow Analyzer to use its behavior.

After it receives the logs, Analyzer then parses them into a Map<String, List<PayerResults>> structure. The String key of the first map is a txId, an alternate identifier parsed from the logs that associates each request with its response. The List of PayerResults objects holds the results of each payer's connectivity test. Each PayerResults object corresponds to exactly one payer using one product, or what Rehearsal calls the payer-product combo.

Each PayerResults object holds a list of Result objects, each of which corresponds to one request and one response. A Result object can either pass or fail, depending on whether or not it contains an error code. The PayerResults object, however, takes severity of all its constituent Result members' errors into account. A small number of YELLOW errors with many successes will not mark the payer's connectivity as having failed the test, but just one RED error will. The rules determining the severity of an error is located in the AaaCodeRules class, which can be configured through an external CSV file. After parsing, Analyzer sorts its PayerResults into those that failed, those that passed, and those that need to be retried, and returns the ReportData containing those lists.

In addition, some error codes require that the test be rerun with different PHI. In order to facilitate this, a class called the RetryRequester collects a list of all responses that must be rerun during the analysis phase and reposts them. To do so, RetryRequester packages the response in a SenderMetadata object and places it on the same queue that Sender uses. The retry, then, is in the same format as metadata sent from Sender, and so will behave in the same manner. The only difference between the two types of metadata is that the retry requests have a marker to indicate that an alternate patient file should be used. Figure 3.4 below illustrates a simplified model of how Analyzer can post retries.

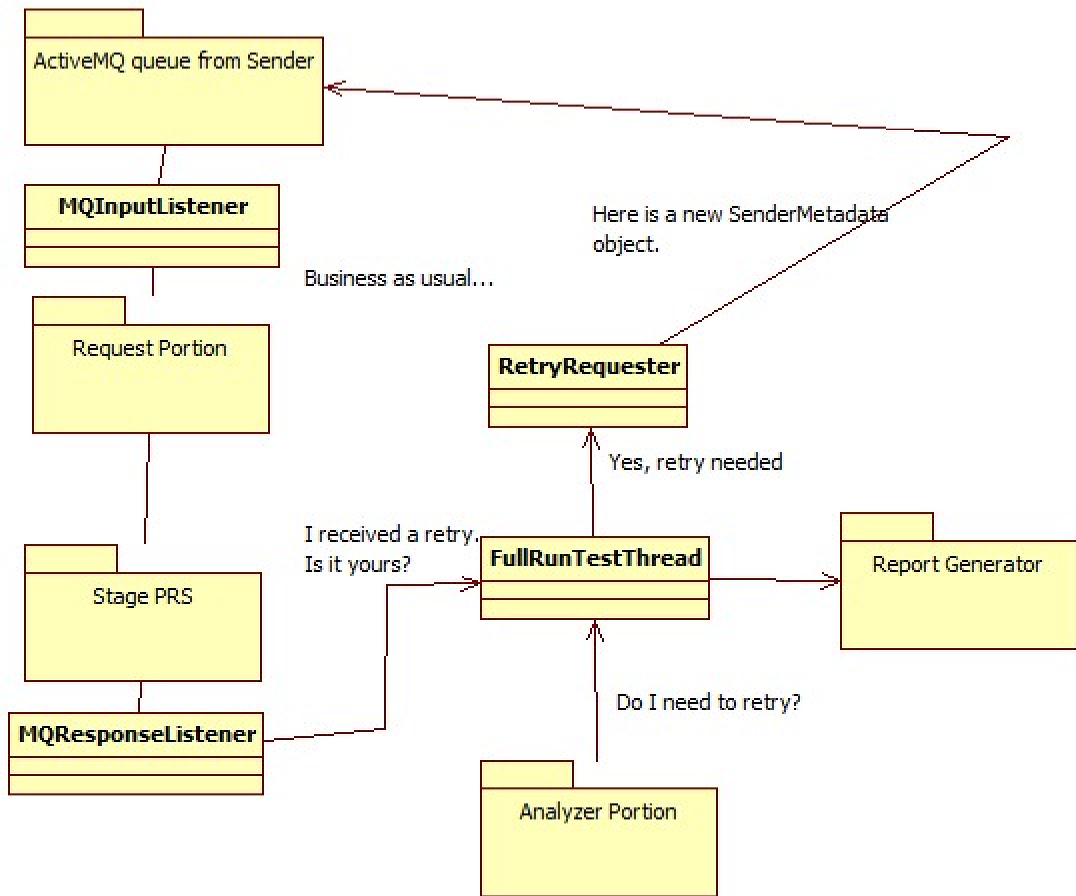

Figure 3.4: Redux

When it is parsing a set of logs into a ReportData object, Analyzer can determine if a retry is necessary by counting the number of times that a particular payer-product pair occurs in the logs. If Analyzer determines that a retry is needed, it adds the Result that must be retried into a list of Results and returns it in its ReportData object.

The FullRunTestThread checks to see if Analyzer found any results that need to be retried. If it discovers that Analyzer determined that retries are needed, FullRunTestThread uses the RetryRequester to post them to ActiveMQ. The results are translated back into PayerAndProduct instances and packaged in a new SenderMetadata object, and will behave just as if it had come from Sender. When MQResponseListener receives a response from PRS, it will notify all the threads currently running and pass them a UUID. If the UUID matches any of the thread's resident UUID, the matching thread will consume the response and generate a new report. This runs Analyzer again, and Analyzer can again determine if a retry is necessary. This loop runs until the Analyzer determines that no further retry is necessary, after which the thread running the test will be terminated after its report is generated.

Whether or not a test suite requires retries, a report is generated each time a message is received from Stage PRS. The Analyzer, after parsing all of its responses read from txLogger, creates and fills a ReportData object. The ReportData object is a model of all the information Recondo needs in the report. It is intended to allow multiple *views* of this model to be established in the future. For example, currently the report is written to disk, but in the future, Recondo would like a GUI interface to read the reports. The use of a model allows any number of views to represent its contents.

Analyzer's last responsibility is to generate a report from the ReportData object. As mentioned in the last paragraph, Analyzer currently generates and writes a report to disk as a plain text file.

Figure 3.5 below shows the design of the simple report generation system.

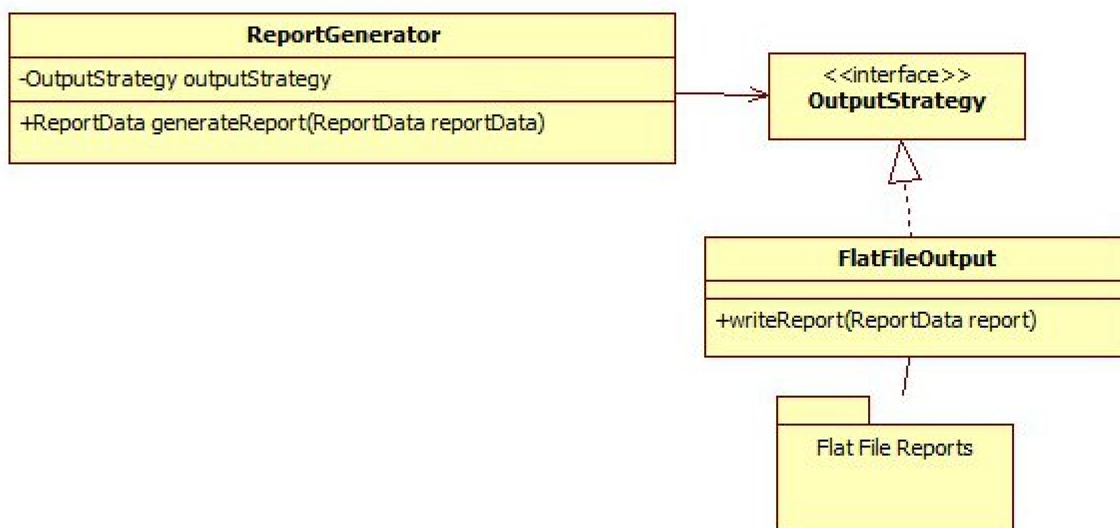

Figure 3.5: Generating reports

Analyzer's report generation system is very simple. All it does is construct a view of the ReportData that is receives as an argument in generateReport(). The OutputStrategy interface is another example of the Strategy pattern. The Strategy pattern was used here because it allowed the easy addition of other views of the ReportData model. All a developer must do is implement the OutputStrategy interface to get access to the model.

Implementation Details

Recondo's existing system is written in Java, and Rehearsal is as well. It was developed using Eclipse IDE, and makes heavy usage of Apache ActiveMQ, the Java Message Service (JMS), the Java DataBase Connectivity system (JDBC), and PostgreSQL.

The Java Message Service is an interface used to implement a message-based communication system between modules in Java programs, and even between

programs running in different virtual machines on separate servers. Apache ActiveMQ is one of several open-source implementations of the JMS interface, and is the one currently in use at Recondo. Rehearsal uses ActiveMQ because it is necessary to interface with Stage PRS.

The database that Rehearsal uses to access Stage PRS's logs to analyze and generate reports for tests is a PostgreSQL relational database maintained by Recondo. Every transaction that goes through PRS is logged in this database, called txLogger, for development purposes and record-keeping. Rehearsal accesses txLogger in its Analyzer component using the PostgreSQL JDBC driver.

In addition to these third-party tools and sources, Rehearsal uses code recycled from existing Recondo architecture. For example, much of the implementation of TxLogReader, which handled the connection to txLogger, was taken from Recondo's existing TxLoggerAnalyzer project. Also, Requester and RetryRequester borrow from another Recondo tool that interfaces with Recondo's ActiveMQ system, a tool called PRSAnalyzer.

Issues related to development but not to coding included security concerns and the use of PHI. HIPAA regulations require that PHI remains secure at all times, and the team struggled to find places to work that could provide this security. Further complicating the issue, the team had several issues with the Subversion source code control system used by Recondo that led to several missed hours of work.

Several other problems were discovered throughout the course of the project as well. At a few junctures, the client had to intervene and alter the existing Recondo system to allow Rehearsal to communicate with it and to give Rehearsal access to the data it needed. Some of these problems have not yet been fully solved. Currently, Rehearsal cannot receive messages from Stage PRS due to security concerns, and a complete test of Rehearsal must wait until the tool can be deployed on a server in the production environment. Also, the cost-analysis and NPI validation portions of Rehearsal are suspended until such time as Recondo can provide access to the cost information.

Results

Rehearsal's success was defined as the receipt of a report given an NPI, list of payers, input paths, and output paths. Due to security concerns, Rehearsal as a whole has not been tested, but the team has tested both the Sender and Analyzer portions of Rehearsal extensively and is confident that such a full-system test will succeed.

Conclusions and Future Directions

The development of Rehearsal, as a tool, will never truly be complete. Recondo is growing rapidly, and Rehearsal is proving to be an integral tool that will be used by employees of varying levels of technical skill across multiple departments.

There were several features that the client desired but were not within the scope of the limited time of a six-week field session. One of the most desired was the development of a GUI using Google Web Toolkit to enable non-developers to easily use Rehearsal. While the team did not have time to implement a GUI during field session, special attention was given to the design of Rehearsal's interfaces to allow a simpler addition of a GUI. These will need some modification, however, to better handle exceptions in a user-friendly fashion.

In implementing Rehearsal, the team was able to explore the art and science of object-oriented development and message-based communication in Java. The experience with ActiveMQ's powerful ability to completely decouple modules and programs proved to be extremely useful and its use resulted in a more robust, flexible product.

Glossary

270 - the industry standard file format to send PHI to request a patient's elibility information

AAA tag - an XML markup tag defined by HIPAA regulations as being the location of an error code in a response file.

activation testing - the process of ensuring that a hospital has connections to all the payers it needs before Recondo's service is activated.

Analyzer - the portion of Rehearsal that runs as a service. Analyzer's responsibilities include formatting industry-standard requests, interfacing with the existing Recondo architecture, analyzing responses, retrying certain tests, and generating reports.

Apache ActiveMQ - an open-source message brokering system that supplies an implementation of the Java Messsage Service (JMS) interface

Authorization Denial Prevention - one of three products that Recondo has developed that requires activation testing.

Comma separated value (CSV) file - a format of a table or relation written such that each row is separated by a newline and each column's values are seperated by commas on that line.

DalMap - a data structure used by Recondo's configuration databases and by Stage PRS.

EligibilityPlus - one of three products that Recondo has developed that requires activation testing.

Health Insurance Portability and Accountability Act (HIPAA) - from HIPAA website <http://www.hhs.gov/ocr/privacy/hipaa/understanding/index.html>: The HIPAA Privacy Rule provides federal protections for personal health information held by covered entities and gives patients an array of rights with respect to that information. At the same time, the Privacy Rule is balanced so that it permits the disclosure of personal health information needed for patient care and other important purposes. Recondo is considered a “covered entity” under HIPAA.

Java DataBase Connectivity system (JDBC) - an interface that enables communication between a JAVA program and a database. Using a database requires a JDBC driver, which is an implementation of the JDBC interface specific to that database.

Java Message Service (JMS) - an interface that defines a message-based communication protocol between Java programs and modules.

Metadata - “data about data”. In Rehearsal, metadata is used to send information concerning how and when a test is to be run.

National Provider Identifier (NPI) - a unique identifier issued by the federal government to a healthcare provider.

Observer - a design pattern in which one module notifies another of a changed state

Payer - an entity that pays some or all of a patient’s healthcare costs.

Payer Resolution System (PRS) - the name of Recondo’s back-end software development and maintenance department.

PostGres - relational database system using SQL.

Protected health information (PHI) - PHI is defined by HIPAA regulations as anything that links a patient to a medical service or benefit.

Rehearsal - the name of the tool developed to automate activation tests.

Sender - the portion of Rehearsal that runs as a command-line tool allowing users to post activation tests or report regeneration tasks.

Singleton - a design pattern which enforces that one and only one object of the given type ever exists at a time

Strategy - a design pattern that encapsulates behavior into an object, so that different behavior can occur simply by passing a different object

Sure Pay Health (SPH) - one of three products that Recondo has developed that requires activation testing.

txLogger - a logging system that holds transaction records on all activity that occurs on Stage PRS

UUID - Universally Unique IDentifer - a hash generated to provide a unique identifier for some entity.