The Design and Implementation of FIG: a Record/Playback Mechanism for ETMS Feeds
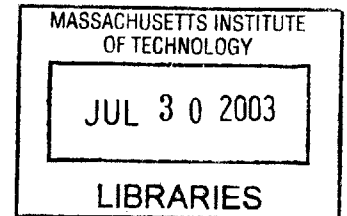
by

Roshan Gupta

Bachelor of Science in Computer Science and Engineering, June 2000.

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

January 17, 2003

Author_____ _____
Department of Electrical Engineering and Computer Science
January 17, 2003

Certified by_____   _____
                                                    Daniel Jackson
                                                  Associate Professor
                                                  Thesis Supervisor

Accepted by_
                                                  Arthur C. Smith
              Chairman, Department Committee on Graduate Theses

The Design and Implementation of FIG: a Record/Playback Mechanism for ETMS Feeds

by

Roshan Gupta

Bachelor of Science in Computer Science and Engineering, June 2000.

Submitted to the
Department of Electrical Engineering and Computer Science

January 17, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design, analysis, and implementation of FIG—a tool that records, analyzes, and plays ETMS air traffic data feeds. Its creation was motivated by the need to test air traffic control applications against a static source of air traffic data. Three fundamental requirements of the system are: 1) maintain the fidelity of the original ETMS feeds, 2) store feed data in portable, platform-independent files, and 3) allow information in recorded feeds to be easily analyzed.

The core functionality of the application, including the recording and playback mechanisms, is implemented in a small-footprint, platform-independent library. This allows 3rd party applications to programmatically manipulate FIG files. A separate application bundled with FIG provides a standard GUI from which to record and play ETMS feeds.

Dependencies in the implementation of FIG can be traced back to major design decisions. Overall, FIG's design is flexible and can support future enhancements.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgements

I would like to thank my advisor Professor Jackson for his advice and insight; Gregory

Dennis for his help with TSAFE; and my family and friends for all their support.

To Millie, thank you for your constant support and words of encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Feed Input Generator (referred to as FIG) is a tool that records, analyzes, and plays ETMS[1] air traffic data feeds. The recorded feeds are stored in portable, platform-independent files that can be streamed to a variety of third-party applications. Such a tool has many practical uses. Researchers can record real air traffic data for use in their studies. Developers can utilize pre-recorded feeds to demonstrate the functionality of their applications. Software testers can build test suites of analyzed feeds to effectively exercise all paths in their systems.

The primary goal of this thesis is to explore important design decisions that were made during the development of FIG. In particular, we hope to examine each key decision in terms of the dependencies it created between system components, and the consequences it had on the design and implementation of the application in general. Ultimately, this thesis should serve as an interesting and well-documented case study on software design whose lessons can hopefully be applied to other software projects.

The remainder of this chapter provides some background on current air traffic research and describes the motivation behind creating the FIG application.

Chapter 2 defines the functional requirements that were established at the beginning of the development cycle.

Chapter 3 discusses the general design approach that was taken and describes in detail the final system implementation.

---

[1] The Enhanced Traffic Management System (ETMS) feed is a special air traffic data feed that is used mainly for research purposes—it consolidates air traffic information from a variety of sources into a single stream.

8

Chapter 4 explores the design of FIG and expounds upon some of the key design decisions that were made.

Chapter 5 concludes this paper by providing a retrospective on the development of FIG and by mentioning some extensions to the tool that are being considered.

1.1 Background and Motivation

The national air traffic control infrastructure is a mission-critical software system that must achieve the highest levels of performance, modularity, fault tolerance, and scalability. The current air traffic control (ATC) system was originally built in the 1960s; since then, air traffic has increased immensely, and it has become increasingly more difficult to maintain safety in the sky. The National Aeronautics and Space Administration (NASA) has begun deploying a new ATC system to upgrade the existing but aging infrastructure. One of the main components of this new system is the Center-TRACON Automation System (CTAS). This application provides a suite of tools to air traffic controllers to help them reduce delays, schedule landings, and maintain safe separation between aircraft. Given the recent advances in technology, most of these tools are vastly superior to their earlier counterparts. Despite these improvements, however, CTAS is still fundamentally limited in its scalability as it was designed with human input at the core of its workflow.

A new concept known as the *Automated Airspace Concept* (AAC) has been proposed by Heinz Erzberger, a senior scientist at NASA, in a recent paper [Erz01]. The idea is to place the responsibility of maintaining safe separation between aircraft on a computer rather than a human. Computer systems are much more scalable and can

handle far more tasks than even the best air traffic controller. This concept is nothing new—telecom companies have been using computers for the past few decades to dramatically increase the capacity of their switching networks. If automation has been proven a success in other areas, the question that comes to mind is: why has it not been applied to the air traffic control domain? The reason is that even a simple flaw within the computer system could have catastrophic consequences. Unless there is some way to guarantee that mistakes in the program can be prevented from causing serious harm, the Federal Aviation Administration (FAA) will continue to rely on a tried-and-tested, but limited, human-based ATC system.

Erzberger proposed that this problem could be overcome by developing a lightweight, failsafe mechanism to monitor the progress of the automated system and take over in case a dangerous situation presented itself. The tool he proposed building is the Tactical Separation Assisted Flight Environment module, or TSAFE. TSAFE detects potential collisions between aircraft within a time period of about two minutes; if one is detected, it automatically sends a conflict avoidance clearance to each plane that instructs the pilots on how to prevent the collision [Erz01]. It is essentially a "last line of defense" against aircraft collisions caused by pilot or system failures. A prototype of TSAFE is currently being developed by the Software Design Group of the Laboratory of Computer Science at MIT.

Before the Federal Aviation Administration would be willing to deploy TSAFE, however, it has to be assured that the application will work exactly as intended. It is therefore imperative that the component be rigorously tested. For this reason, the Software Design Group supported the development of the FIG application. FIG allows

TSAFE developers to test their system against a pre-recorded ETMS input feed (as opposed to a dynamic, real-time feed). This use of recorded air traffic data facilitates testing in a number of ways:

- *Path coverage* – a collection of pre-recorded test feeds can be assembled that exercises all execution paths within the system.

- *Regression testing* – new versions of TSAFE can be validated against suites of previous test feeds.

- *Measurable improvement* – testing against static air traffic data makes it easier to quantify and track progress.

- *Failure analysis* – testing against a known set of inputs allows failure conditions to be easily traced.

- *Environment requirements* – testing can proceed even when access to a live ETMS feed is limited

Given the importance of testing TSAFE and other air traffic control applications, it is not surprising that tools to record air traffic data have already been developed. CTAS contains such a tool, although it is tightly integrated within the CTAS environment and only records air traffic data from publicly unavailable feeds. An archival system that records ETMS feed data was built by MIT students Micah Gutman [Gut00] and Joyce Lin [Lin02]; unfortunately, this system destroys the format of the original feed, and thus only supports historical queries and not playback. In short, none of the existing tools satisfied the needs of the Software Design Group and the TSAFE project in particular. By developing a proprietary mechanism, we could ensure that FIG was compatible with

ETMS data and optimized to preserve the fidelity of the original feed. In addition, we could easily make changes to FIG and release copies of the tool to other researchers at our discretion. To sum up, we believe that FIG will have its place as an ETMS-based air traffic archival tool targeted toward the research community.

Chapter 2

Functional Requirements

When designing FIG, there were three high level requirements that helped shape the functionality of the application. The first requirement was to preserve, as much as possible, the fidelity of the original feed. Since FIG was primarily being developed to test air traffic applications, it was important that the recorded feeds be indistinguishable from the original feeds to prevent the skewing of test results. This implied that FIG would need to preserve feed characteristics such as inter-message delays, message ordering, and transmission errors. It also implied that the final system would have to be fast enough to record and replicate these characteristics in real-time. This fidelity requirement ultimately affected almost every aspect of the application—including the file format, recording strategy, playback strategy, and filtering mechanism.

The second high level requirement was to store the recorded feed data in a portable file format. Portability in this case meant three things. First, the final size of the data files (often called "FIG files") should be limited to a reasonable amount. This allows the FIG application to achieve its goal of creating archived feeds that can be easily distributed to other researchers or assembled into test suites. Second, the FIG files should be platform-independent. Third, the mechanism to play FIG files should be both lightweight and platform-independent. Even if the FIG files happen to be small and platform-independent, they are not truly portable unless they can be utilized in a similarly straightforward way.

The third high level requirement was to design the FIG files so that they are amenable to analysis. This achieves two important goals. First, it allows FIG to be useful as a testing tool, since users of FIG can map out the exact characteristics of their test feeds. Second, FIG can take advantage of the optimized file format to provide a filtering mechanism that allows users to exclude certain messages from playback. This feature might be used by researchers, for example, to increase the performance of their applications by focusing on relevant information only. To sum up, this high level requirement implied that the final application would have to be designed with feed analysis in mind in order to preserve the overall system's performance and scalability.

The next few sections discuss in detail the requirements for FIG's core functionality—namely the recording, playing, and filtering of feeds. Note that there are additional requirements for the application dealing with the user interface, usability, and so forth that are not discussed here.

2.1 Recording Mechanism

The first requirement of FIG is the ability to record messages from a live ETMS feed (or another FIG file) and store the data to disk. The recording mechanism should take into account the following constraints:

2.1.1 Fidelity

The recorded feed should preserve the fidelity of the original feed as much as possible. This requirement has numerous consequences. First, the recording mechanism will have to preserve the content of messages and the relative ordering of messages. Second, the

recording mechanism will need to keep track of the transmission delay between every[2] pair of messages. Third, the recording mechanism will have to preserve any transmission errors and badly formatted messages.

2.1.2 File Format

The recorded feed data should be stored in a portable, platform-independent file format. The chosen file format should be optimized to support the playback, analysis, and filtering of its data. However, it should also be flexible enough to preserve transmission errors and badly formatted messages. Finally, the resulting files should be compressed or otherwise manageable in terms of size.

2.1.3 Multiple File Recording

FIG should allow users to record from an ETMS feed for any length of time. However, given the high data rate of the ETMS feed, the size of the resulting FIG file would in most cases conflict with the portability requirement mentioned in the previous section. Hence, a final requirement of the recording mechanism is the ability to record a continuous stream of data to a set of multiple FIG files.

2.2 Playback Mechanism

Another basic requirement of FIG is the ability to stream the data stored in a single FIG file. The playback mechanism should take into account the following constraints:

---

[2] It is not sufficient to only record the delay between *consecutive* messages. The reason for this is that if some messages are filtered from a recorded feed during playback, the playback mechanism should have enough information to reproduce the transmission delays between the messages that remain.

2.2.1 Multiple File Streaming

Users should be allowed to play back messages from an arbitrary set of FIG files. In this case, messages from the first FIG file in the set would be streamed, followed immediately by the messages in the second FIG file, and so forth. Note that this functionality is the counterpart of the multiple file recording requirement mentioned above.

2.2.2 Playback Rate

The playback mechanism should reproduce the transmission delays between messages that were recorded from the live feed. In addition, however, the user should be allowed to temporarily alter these delays to control the final rate of playback. For instance, the user should have an option to scale the transmission delays by a fixed factor—scaling by a factor of 0.5 reduces the delays by half and effectively doubles the playback rate. Users should also have the flexibility of removing the transmission delays altogether, or inserting a fixed delay between messages.

2.2.3 Filtered Playback

Users should have the ability to filter the messages that are streamed from a FIG file. This filtering capability should be dynamic—in other words, the user should be able to change the filtering conditions each time a FIG file is streamed. Filtering is discussed in more detail in the next section.

2.3 Filtering Mechanism

Another requirement of FIG is the ability to filter messages from a recorded feed during playback. Whether a particular message is filtered should depend on any combination of the following message characteristics:

- *Type* – see Table 2.1 for a summary of the ETMS message types

- *Airline* – the airline that owns the flight referenced in the message

- *Facility* – the facility (geographical location) where the message originated

- *Integrity* – whether the message contains transmission or formatting errors

Note that these message characteristics are only a small subset of the possible filtering conditions; we chose to require these four specific conditions because they are straightforward to identify. Future versions of the tool can easily expand the number of filtering conditions supported. For example, FIG might allow messages to be filtered based on the type of flight they represent, such as a general versus commercial or en-route versus terminal flight.

Finally, FIG should also include a profiling mechanism that allows users to examine each of the above characteristics for all messages in a given FIG file.

Message Type	Description
AF	Flight Plan Amendment
AZ	Flight Arrival
DZ	Flight Departure
FZ	Flight Plan
HB	ETMS Heartbeat
RT	ETMS Flight Predictions
RZ	Flight Cancellation
TO	ETMS Trans-Oceanic Position Tracking
TZ	Flight Position Tracking
UZ	Flight Plan Update (boundary crossing)

Table 2.1: ETMS Message Types

2.4 3rd Party Integration

A final requirement of FIG is the creation of a lightweight mechanism to allow 3rd party applications to programmatically record and play FIG files. This mechanism should be platform-independent and small in size to facilitate its distribution. The purpose of this component is to allow other applications to utilize FIG files without the need to run the large, main FIG application each time.

Chapter 3

Implementation

This section of the thesis discusses the implementation of FIG. First, an overview of the design strategy is given, followed by a detailed description of the final system. Note that the system implementation described here is not the ideal implementation—in fact, some of the lessons learned during the development of FIG will be later used to make an improved version of the tool. Finally, keep in mind the three high level functional requirements discussed in the previous section—fidelity, portability, and ease of analysis. The influence that these three requirements had on the final implementation should be apparent as you read the remainder of this chapter.

3.1 Design Strategy

We decided early on to isolate the core functionality of FIG—i.e. the recording, playing, and filtering of feeds—into a small-footprint, platform-independent library (referred to as the IO library). This strategy has numerous advantages. First, it achieves the functional requirement of having a lightweight recording and playback mechanism that can be integrated into 3rd party applications. Second, it reduces the complexity of the main FIG application (referred to as simply FIG) in that the main application only needs to provide a user interface that sits on top of the IO library. Third, this strategy achieves a balanced separation between functionality and presentation. Because the GUI and the IO library are separate components, changes to one component can be isolated from the second

component. Figure 3-1 depicts the relationship between the IO library, FIG, and 3rd party applications.
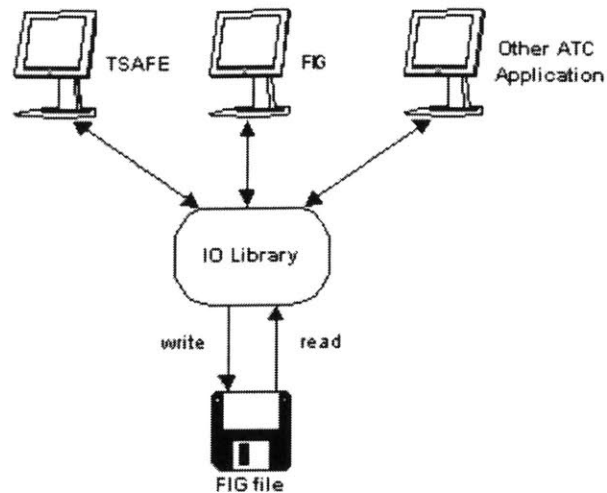
Figure 3-1: Interaction between FIG system components

3.2 IO Library

The IO library is implemented in Java and contains all the classes necessary to read and write FIG files. The library is made up of three main abstractions:

- *FIG file* – a file that stores all the information necessary to replicate a recorded ETMS feed.

- *FIG file writer* – a mechanism to record ETMS feed data to a FIG file.

- *FIG file reader* – a mechanism to play the ETMS feed data in a FIG file.

In the next few sections, we discuss how each of these three abstractions is implemented.

3.2.1 FIG File

The FIG file represents a recorded ETMS feed. It is a single file that can be easily distributed and replicated, and it can function across platforms. It contains all the information necessary to replicate the recorded feed, including the feed's message sequence, inter-message delays, and transmission errors.

The information contained in a FIG file is stored primarily in a database. There is also a separate configuration file that stores meta-information about the FIG file as a whole. A compression algorithm is used both to archive the multiple data files into a single unit and to reduce the final size of this archive.

Figure 3-2 contains a module dependency diagram (MDD) [Jac02] of the classes that make up the FIG file abstraction.
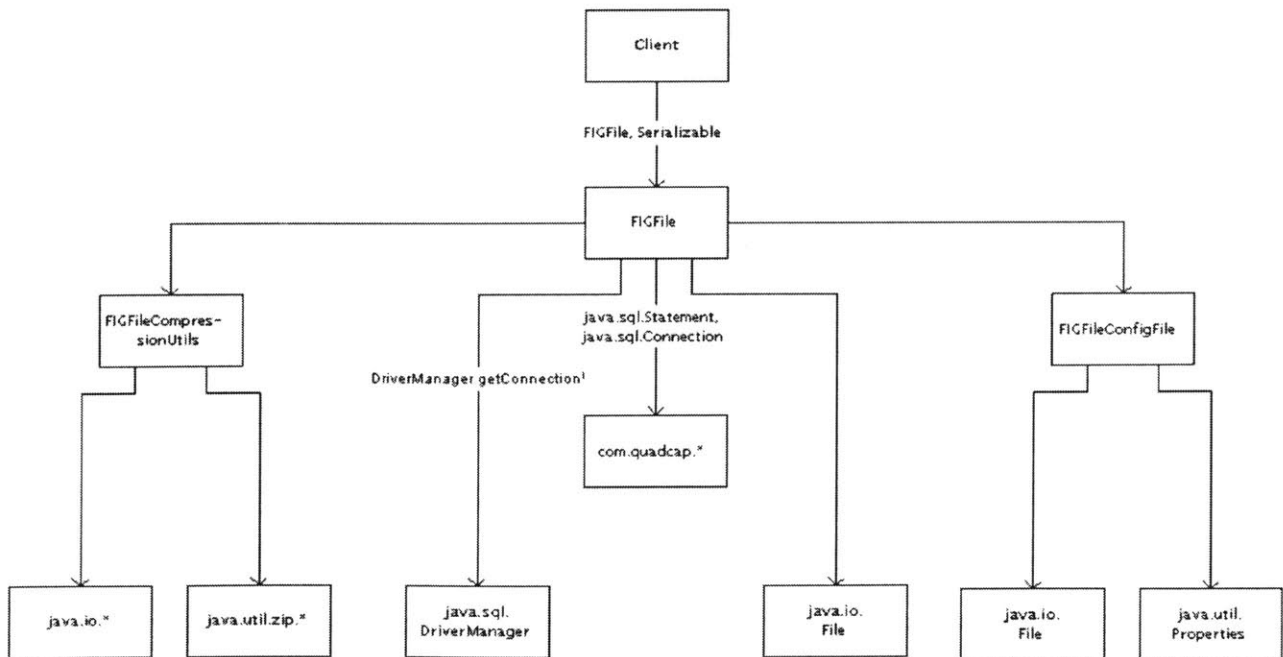
Notes:

[1] The JDBC database URL passed to this method is dependent on QED's particular URL format.

Figure 3-2: FIG File MDD

21

3.2.1.1 Database

Each FIG file contains a database. The database's schema is designed to store individual

ETMS messages as well as meta-information about each message. The schema consists

of two tables—the *Feed* table and the *Message* table. A detailed description of each table

can be found below.

The Feed table is used to store ETMS messages as well as frame information about each message.
Each row in this table represents a single ETMS message.

Column Name	Description	Type
fts	FIG-assigned timestamp (**primary key**)	timestamp(32)
seqnum	Sequence number	int
ets	ETMS-assigned timestamp	timestamp(32)
fclty	Facility identifier	char(4)
msg	Complete message text	varchar(0)

Table 3.1: Feed table

The Message table is used to store field information extracted from ETMS messages.
Each row in this table represents a single ETMS message.
Note that this table can be joined with the Feed table on the "fts" column.

Column Name	Description	Type
fts	FIG-assigned timestamp (**primary key**)	timestamp(32)
mtype	ETMS message type	char(2)
aline	Airline referenced in message	char(3)

Table 3.2: Message table

A snapshot of a sample database is shown in Figure 3-3. Note that each message

is assigned a unique timestamp by FIG that identifies when the message was initially

recorded. This timestamp serves two purposes. First, it preserves the sequence of the

ETMS messages. Second, it allows the transmission delay between any pair of messages

to be calculated. Also note that the third message in the database represents a

transmission error; it has been flagged as a badly formatted message by inserting null values into some of its table columns.

Feed Table

fts	seqnum	ets	fclty	msg
2002-10-24 17:20:22.753	F2DB	24211746	KZAB	F2DB24211746KZABTZ UAL214/114 515 370 3601N/10229W
2002-10-24 17:20:22.762	F2DC	24211746	KZAU	F2DC24211746KZAUTZ AAL1918/435 539 330 3203N/10509W
2002-10-24 17:20:22.773	-1	null	null	F2DETZ DAL1736/938 538 405C 3536N/10042W

Message Table

fts	mtype	aline
2002-10-24 17:20:22.753	TZ	UAL
2002-10-24 17:20:22.762	TZ	AAL
2002-10-24 17:20:22.773	null	null

Figure 3-3: Database snapshot

The database is implemented by Quadcap's Embeddable Database[3] (QED). QED is a small footprint, low-administration database, which makes it ideal for use in FIG files. It is also open-source software, allowing it to be distributed without cost for non-commercial purposes. Most importantly, QED is implemented in Java, allowing it to be platform independent.

The content of a FIG file's database is accessed only through the Java Database Connectivity (JDBC) API. This preserves the platform independence of the FIG file. Direct access to the database is limited to package-friendly classes, such as `FIGFileReader` and `FIGFileWriter`. These classes use a JDBC connection, for

---

[3] Quadcap Software: http://www.quadcap.com

example, to read from and record to a FIG file. Third party applications wishing to access the information in a FIG file must do so through one of these two classes.

3.2.1.2 Configuration File

Each FIG file is associated with a configuration file. This file can store an arbitrary number of key-value pairs. Currently, this configuration file serves two purposes. First, it contains a version number that facilitates the evolution of the FIG file format. Second, its existence serves as one of the methods to distinguish authentic FIG files from other files in a directory.

The configuration file (i.e. class `FIGFileConfigFile`) is implemented using Java's `Properties` class, as shown in Figure 3-2.

3.2.1.3 Compression

A compression algorithm is used to make each FIG file manageable. First, the multiple files that make up the FIG file's database and configuration properties are archived into a single file. This facilitates the distribution and manipulation of the FIG file; it also requires, however, that the FIG file be expanded into a temporary directory each time its contents are accessed. Second, this archive is compressed to reduce its final size and increase the FIG file's manageability.

The compression (and archival) tool used is Java's integrated ZIP package. Testing has shown that the ZIP algorithm achieves a typical compression rate of 10x for FIG files. The logic to perform the compression is isolated in the class `FIGFileCompressionUtils`, as shown in Figure 3-2. The class `FIGFile` is

24

responsible for triggering compression and decompression as necessary. Note that the use of compression is not visible beyond the FIG file abstraction.

3.2.2 FIG File Writer

The FIG file writer is the mechanism through which ETMS feeds can be recorded to FIG files. It is modeled after Java's IO package, and it implements all of the functional requirements for the recording mechanism as stated in section 2.1.

To record an ETMS feed, an application simply needs to instantiate a `FIGFileWriter` object. It can then read data from the feed source and pass it to the FIG file writer via the `java.io.Writer` interface. Figure 3-4 contains sample code that demonstrates this procedure. Note that the `FIGFileWriter` object does not actually write the feed data to disk—instead, it passes the data to a `PhysicalFIGFileWriter` object that parses the information and then physically creates the FIG file.

```
public void writeFigFile(String filename, Reader feedSource) {

    // Create the FIG file.
    FIGFile file = new FIGFile(filename);

    try {
        // Create a writer.
        Writer writer = new FIGFileWriter(file);

        // Now read directly from the feed and write the
        // data to the FIG file.
        char[] buffer = new char[1024]
        int numCharsRead = feedSource.read(buffer);
        while (numCharsRead != -1) {
            writer.write(buffer, 0, numCharsRead);
            numCharsRead = feedSource.read(buffer);
        }

        writer.flush();
        writer.close();
    }
    catch (IOException ioe) {
        // Handle exception here.
    }
}
```

Figure 3-4: Recording an ETMS feed

Figure 3-5 contains a module dependency diagram of the classes that make up the

FIG file writer abstraction.

Figure 3-5: FIG File Writer MDD
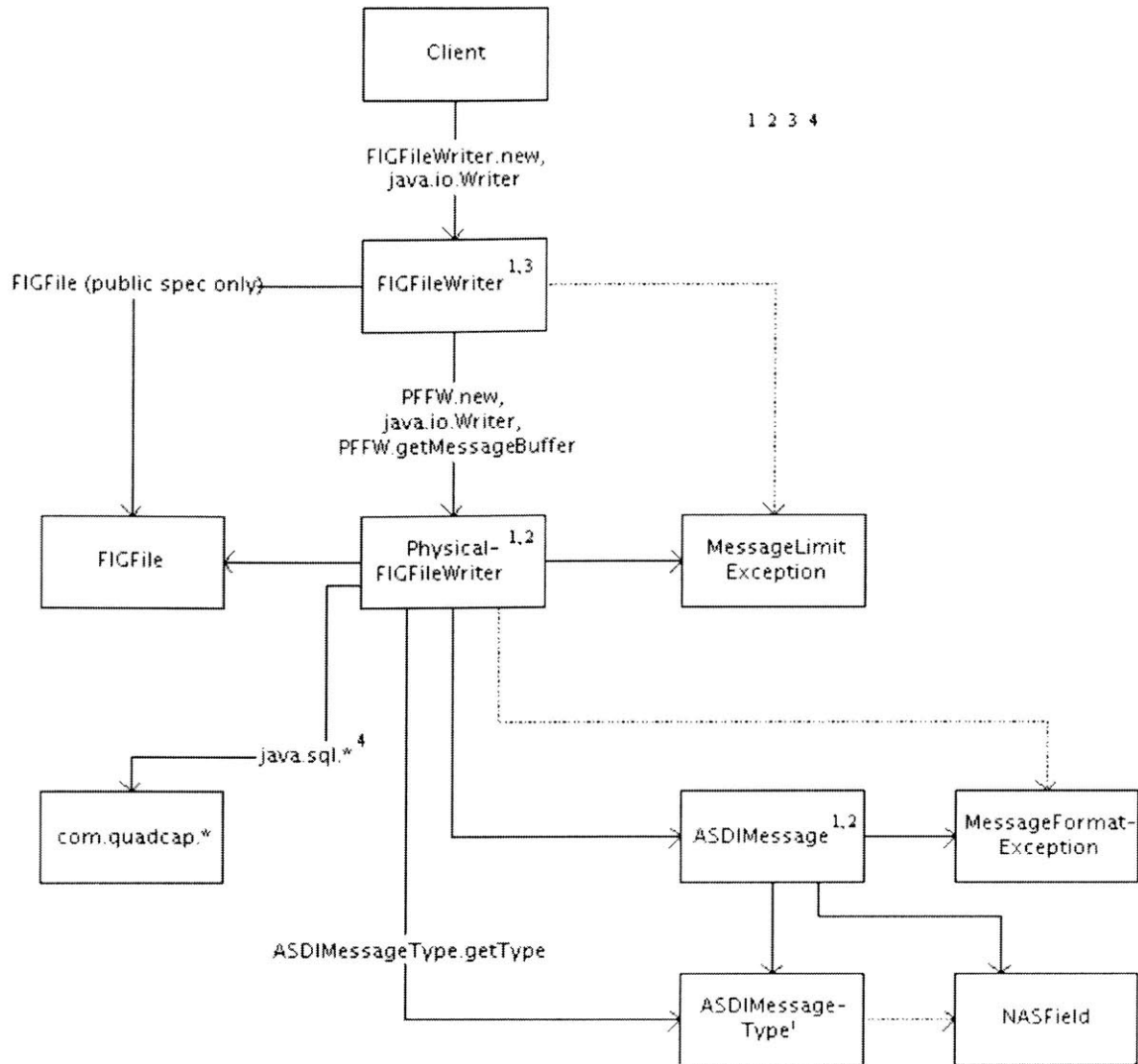
Notes:

[1] Not shown: dependency on java.util.*
[2] Not shown: dependency on java.sql.Timestamp
[3] Not shown: dependency on java.io.File via method "getPath"

[4] An optimization involving assigning timestamps depends on QED's particular SQLException format for primary key violations

3.2.2.1 PhysicalFIGFileWriter

The `PhysicalFIGFileWriter` is responsible for writing incoming feed data to a

single FIG file on disk. This entails creating the new FIG file, obtaining a connection to

the FIG file's database, and using the JDBC API to insert information into the appropriate tables. To prepare the feed data for insertion, this class uses a separate parser (discussed in the next section) to extract information about each message. In addition, it assigns a timestamp to each message that will later be used to sequence the messages and simulate transmission delays.

The `PhysicalFIGFileWriter` is also responsible for handling transmission errors. When the parser signals that a message is formatted incorrectly, the `PhysicalFIGFileWriter` inserts the message into the database along with the appropriate null values.

3.2.2.2 Parser

A parser is used to extract information from each ETMS message. Since it was designed to only parse the fields required by the database schema (i.e. message type, airline, etc.), it has a very simple and straightforward implementation. The classes `ASDIMessage`, `ASDIMessageType`, and `NASField` make up the parser, as shown in Figure 3-5. Note that future versions of FIG might choose to expand the parser's capabilities. For example, a more sophisticated parsing engine might extract altitude or flight plan data from messages. This would increase the complexity of the engine, however, since this information is transmitted in a compressed, proprietary format.

The parser is also responsible for detecting transmission errors. If it encounters a badly formatted message, it will report the error by throwing a `MessageFormatException`. The parser is *not* responsible for detecting content errors, however. If a message is formatted correctly but does not contain valid data—for instance, it contains a facility identifier that does not exist—no error will be reported.

28

This behavior was intentional in order to reduce the complexity of the parser. Note however that the invalid message will still be recorded intact to the database and thus will appear as expected whenever the feed is played back.

3.2.2.3 Multiple File Recording

When an application instantiates a `FIGFileWriter`, it can optionally specify the maximum number of messages that can be recorded to a single FIG file. This lets the application record a large number of messages from a feed while keeping the sizes of the resulting FIG files manageable. It is the `FIGFileWriter` object's responsibility to manage the multiple file recording; when the object receives notification from the current `PhysicalFIGFileWriter` that the message limit has been reached, it closes the current FIG file and opens a new file for writing. This new file will be named after the original FIG file, but with a timestamp appended to it to prevent naming conflicts.

3.2.3 FIG File Reader

The FIG file reader is the mechanism through which recorded ETMS feeds (i.e. FIG files) can be played back. It is modeled after Java's IO package, and it implements all of the functional requirements for the playback mechanism as stated in section 2.2.

To stream a FIG file, an application simply needs to instantiate a `FIGFileReader` object. It can then read data from this object as if it were a live ETMS feed, using the `java.io.Reader` interface. The application can optionally filter the messages that are played back by specifying filtering conditions in a `FIGFileContentFilter`. Figure 3-6 demonstrates this procedure. Note that the `FIGFileReader` does not directly read the feed data from disk—instead, much like the

29

`FIGFileWriter`, it uses a separate `PhysicalFIGFileReader` object to actually access the information.

```
public void readFigFile(String filename) {

    // Instantiate the FIG file.
    FIGFile file = new FIGFile(filename);

    // Create a content filter that only includes "TZ" messages from
    // American Airlines.  Play the feed at twice the speed.
    FIGFileContentFilter filter = new FIGFileContentFilter();
    filter.setMessageTypes(new String[] {"TZ"});
    filter.setAirlines(new String[] {"AAL"});
    filter.setDelayScalar(0.5);

    try {
        // Create a reader with the given content filter.
        Reader fileReader = new FIGFileReader(file, filter);

        // Now read each message in the feed and display it on
        // the screen.  Since each message is terminated by a
        // newline character, use a BufferedReader to extract
        // the messages.
        BufferedReader reader = new BufferedReader(fileReader);
        String message = reader.readLine();
        while (message != null) {
            System.out.println(message);
            message = reader.readLine();
        }

        reader.close();
    }
    catch (IOException ioe) {
        // Handle exception here.
    }
}
```

Figure 3-6: Playing a FIG file

Figure 3-7 contains a module dependency diagram of the classes that make up the FIG file reader abstraction.
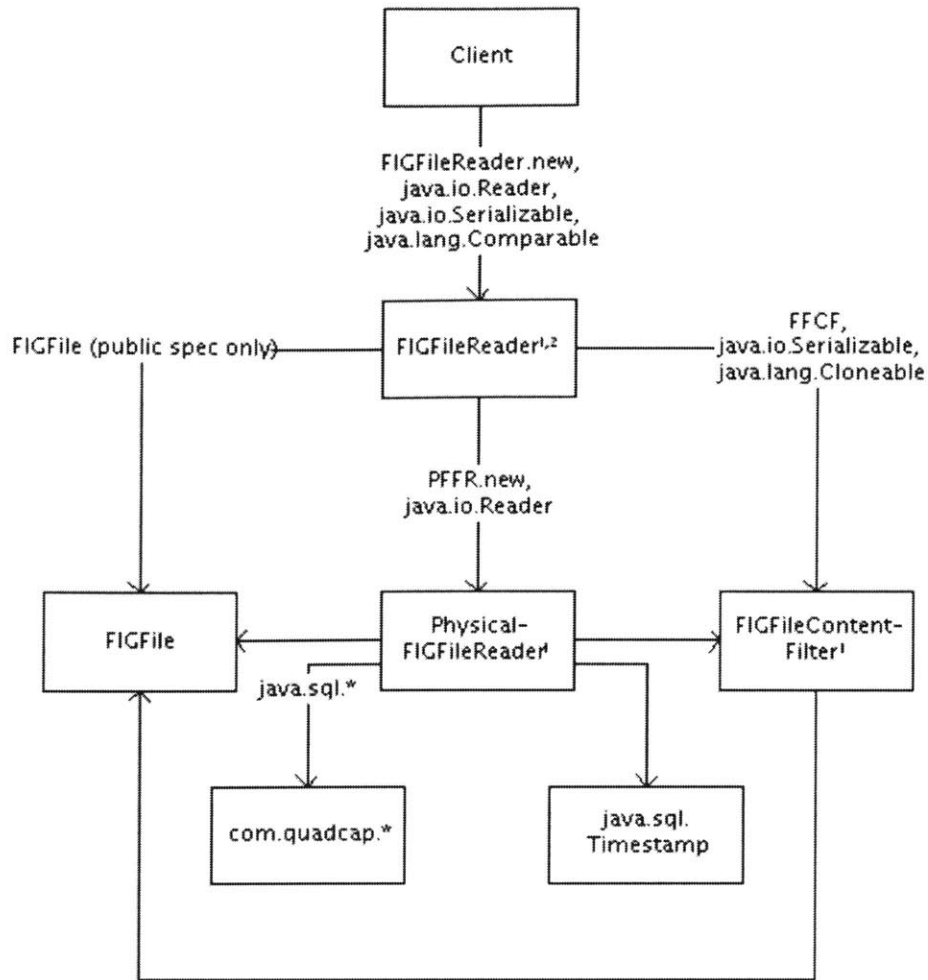
Notes:

[1] Not shown: dependency on java.util.*
[2] Not shown: name dependency on java.io.File

Figure 3-7: FIG File Reader MDD

3.2.3.1 PhysicalFIGFileReader

The `PhysicalFIGFileReader` is responsible for reading feed data from a single

FIG file on disk. This entails opening the FIG file, obtaining a connection to the FIG

file's database, and using the JDBC API to query the database tables for information. As

an example, to play back a FIG file without filtering, the following SQL query would be used:

SELECT feed.fts, feed.msg FROM feed ORDER BY feed.fts

This query returns the ETMS messages in the sequence in which they were originally recorded. The `PhysicalFIGFileReader` object would then return each message in the result set in order, using the timestamps to simulate the transmission delay between consecutive messages.

3.2.3.2 Content Filter

If an application wishes to play back a filtered version of a recorded feed, it would have to instantiate a `FIGFileContentFilter` and specify filtering conditions using the object's convenience methods. It would then pass this content filter to a `FIGFileReader` object that will ultimately use the filter to modify its SQL queries. For example, if an application creates a filter to play back *TZ* messages from American Airlines (as shown in Figure 3-6), the FIG file reader would use the following modified SQL query[4]:

> **SELECT feed.fts, feed.msg**
> **FROM feed INNER JOIN message ON feed.fts = message.fts**
> **WHERE (message.aline = 'AAL') AND (message.mtype = 'TZ')**
> **ORDER BY feed.fts**

Once this query has been processed, the `PhysicalFIGFileReader` would return the messages in the result set as described in the previous section.

---

[4] For a discussion on the performance costs of this query versus a similar query based on a single-table schema, refer to section 4.2.2.

Note that the `FIGFileContentFilter` also provides convenience methods to scale or specify transmission delays. The FIG file reader will take this information into account when returning messages from a result set.

3.2.3.3 Multiple File Streaming

An application can stream multiple FIG files in a row by instantiating a `FIGFileReader` object with a list of files. The `FIGFileReader` will stream each file individually and in turn using a `PhysicalFIGFileReader` object, until the list is exhausted. Note that the `FIGFileReader`'s content filter will be applied to each file in the list.

3.2.4 Package Summary

Table 3.3 below summarizes the files that make up the IO library. An API of the library's public methods can be found in the Appendix.

Filenames in boldface type indicate public classes; all other classes are package-friendly.

Filename	Description
FIG File Abstraction:	
FIGFile.java	Implements the FIG file abstraction
FIGFileConfigFile.java	A FIG file's configuration properties
FIGFileCompressionUtils.java	Logic to compress/decompress FIG files
FIG File Writer Abstraction:	
FIGFileWriter.java	Implements the writer abstraction; Handles multiple file recordings
PhysicalFIGFileWriter.java	Writes feed data to disk
MessageLimitException.java	Indicates maximum # of messages written
ASDIMessage.java	ETMS message parser
ASDIMessageType.java	ETMS message parser
NASField.java	ETMS message parser
MessageFormatException.java	Indicates a transmission error
FIG File Reader Abstraction:	
FIGFileReader.java	Implements the reader abstraction; Handles multiple file playback
FIGFileContentFilter.java	Specifies filtering conditions
PhysicalFIGFileReader.java	Reads feed data from disk

Table 3.3: IO Library files

The library contains approximately 4800 lines of code, of which around one-third is documentation. The library's files are packaged into a single JAR file of size 0.6MB (0.5MB of which is taken up by the QED database's JAR file). We feel that we achieved the goal of creating a portable, platform-independent library that implements the core functionality of FIG.

3.3 Main Application

The main FIG application provides a GUI to access the functionality built into the IO library. It treats live ETMS feeds and recorded ETMS feeds the same—that is, users can play and record both types of data. The application also provides some enhanced features applicable only to FIG files. Users can filter the information played back from a FIG file

by using the GUI to construct a custom content filter. Users can also use the GUI to control the rate of playback by modifying the transmission delays between messages. Finally, users can use the GUI to profile the contents of a FIG file to display information about the recorded feed's message types, airlines, facilities, and so forth.

A screen shot of FIG is provided in Figure 3-8. Also shown is the application's context-sensitive help system, which was implemented using the open-source JavaHelp package.
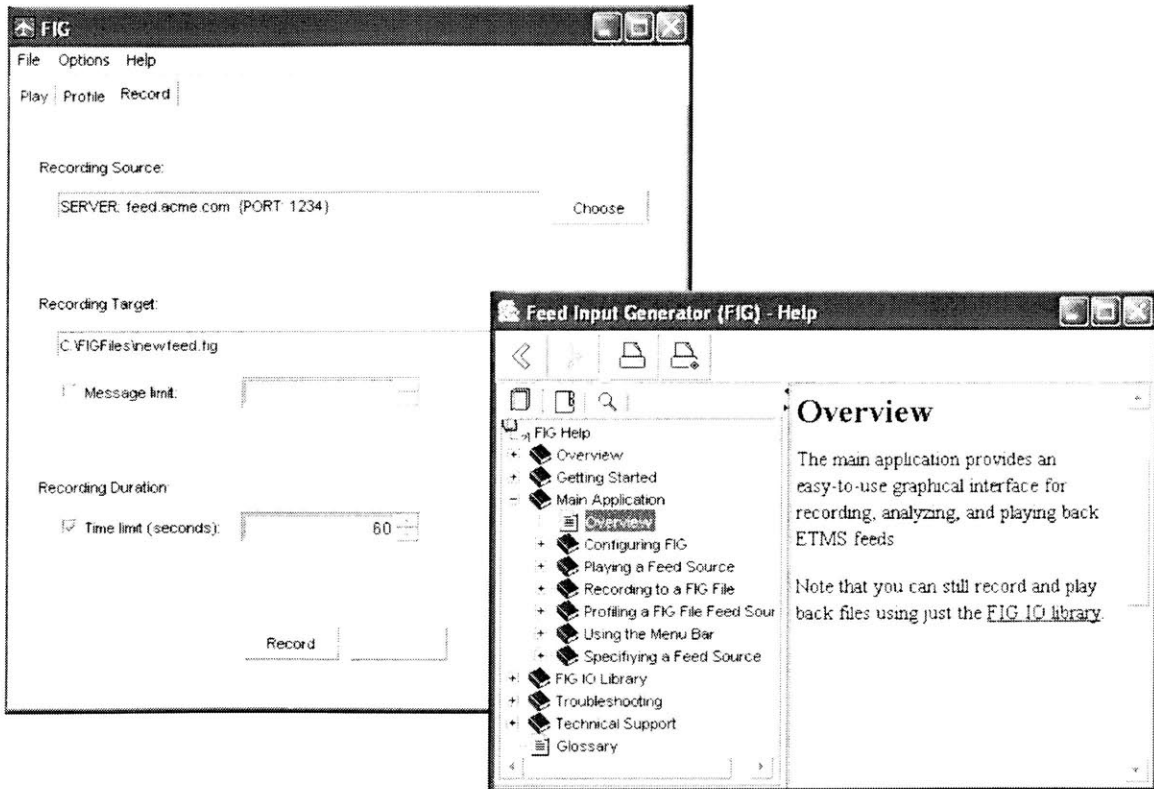
Figure 3-8: Screenshot of FIG with JavaHelp

3.3.1 Package Summary

The main FIG application contains approximately 16,000 lines of code, of which around one-third is documentation. The final size of the entire application is 1.6MB, which includes the IO library, API documentation, and JavaHelp files. We feel that the compact size of the complete application makes FIG an ideal research tool that can be easily distributed via the Internet.

Chapter 4

Design Analysis

This chapter elaborates on the design of FIG and reveals the reasoning behind the major design decisions that were made during its development. The primary objective of this chapter is to explain why certain dependencies exist within the system, as well as how these dependencies affected the final implementation.

The first section discusses general design patterns that helped shape the system. The remaining sections explore the design of the three main abstractions—the FIG file, FIG file writer, and FIG file reader.

4.1 Design Patterns

There were two design patterns that influenced the implementation of FIG—the Strategy pattern and the Decorator pattern [GHJV95]. The Strategy pattern was used more in spirit than in form—it was the motivation behind splitting the IO library's writing and reading mechanisms into two classes each. Like the *Strategy* class from the design pattern, the `PhysicalFIGFileWriter` and `PhysicalFIGFileReader` classes encapsulate the logic to manipulate FIG files on disk. The `FIGFileWriter` and `FIGFileReader` classes, which act like the *Context* participant in the pattern, utilize these "strategies" to record and play FIG files, respectively. Since these two classes interact with the strategy classes through standard interfaces (see Figures 3-5 and 3-7), the strategy classes can theoretically be easily substituted for new implementations, albeit at compile time only. This design, motivated by the Strategy pattern, lets the algorithms

to physically access FIG files on disk vary independently of the general recording and playback mechanisms. In addition, it allows us to isolate dependence on the FIG file format to the strategy classes only.

The Decorator pattern was the motivation behind having the `FIGFileWriter` and `FIGFileReader` classes implement the `java.io.Writer` and `java.io.Reader` interfaces, respectively. These two classes were being designed to be compatible with the standard Java IO classes; the Java IO package takes full advantage of the Decorator pattern to allow its classes to be wrapped in one another (such as wrapping a `Reader` object in a `BufferedReader` object). Some of the complications that arose from adhering to this pattern are discussed in section 4.3.2.

4.2 FIG File

The FIG file forms the foundation of the FIG application. Its implementation influences the entire system, including the system's performance, portability, scalability, and complexity. There were three key decisions that were made when designing the FIG file abstraction. The first concerned the disk representation of FIG files—that is, how to physically represent recorded ETMS feed data on disk. The second decision centered on the design of the database schema. The final decision involved choosing an appropriate compression algorithm to reduce the final size of the FIG files. Each of these decisions is discussed in more detail below.

4.2.1 Disk Representation

The most basic decision made while designing FIG was how to physically store feed data on disk. It was also the most important, since the final choice would significantly impact

the rest of the system. The disk representation affects system performance—how fast FIG can record, play, and filter ETMS feeds. It affects scalability—the maximum number of messages that can be recorded to and retrieved from a FIG file. It affects portability—the physical size of each FIG file, and the versatility (platform-independence) of each FIG file. It also helps determine development costs—the time and effort required to implement the solution, and the additional effort that would be required to make changes to the implementation. Finally, it influences the overall system's complexity—how well the system's modules are abstracted from the storage mechanism.

Two different disk representations were considered. The first alternative was to store message data in a flat file. At a minimum, this would entail writing each message as it is received consecutively into a text file; each message would have to be timestamped in order to aid in the calculation of transmission delays. It is likely, however, that the actual file format would be more complex. The storage mechanism would need a fast method of identifying characteristics of messages if it has to filter the feed—hence, during recording the mechanism might pre-parse fields from each message and store the information somewhere in the file (much like a database index). The primary advantage of using a flat file is that the file format is proprietary. This allows the format to be easily extended at a future date; more importantly, it makes it easier to guarantee that the files are compact and platform-independent. Another advantage of flat files is that the storage mechanism itself can be made lightweight and platform-independent—only the features that are really used will be included in the mechanism, and the mechanism can be implemented in a portable language such as Java. There are some major disadvantages to flat files as well, however. For instance, specifying what

information to retrieve from a file might be difficult since there is no formal query language like SQL for databases. Also, anything other than a straightforward query might be difficult or costly to carry out. For example, if very selective filtering conditions have been specified, the storage mechanism would have to scan through the entire file to find relevant messages unless some sort of index had been previously created. Even if an index is present, complicated queries might not be carried out in the most efficient manner unless the mechanism also has a way to determine the optimal search method. A last disadvantage of flat files is that it may be difficult to reorder messages, add new messages, or otherwise modify a feed that is stored in a flat file. If the file format has been optimized to improve query performance, such actions might be complex and difficult to implement.

The second alternative considered was to store feed data in a relational database. This method is described in more detail in section 3.2. There are many advantages to using a database. First and foremost, databases are optimized to store a large number of records for fast retrieval. They can even handle complex queries (i.e. queries with many filtering constraints) efficiently. And, unlike pure flat files, databases can easily support additional modifications to a recorded feed. Another advantage of relational databases is that they support SQL, a widely-used and well-defined query language. Java directly supports this query language through its JDBC API. Databases have disadvantages as well. Probably the biggest disadvantage is the additional space overhead a database introduces. We calculate (see Figure 4-1 for more details) that a database requires 18 times as much space as a corresponding flat file that has messages and timestamps only (i.e. no index). This space overhead becomes more significant the longer an ETMS feed

is recorded. There is also the issue of portability. Unless we are careful, we might

choose a database implementation that has hidden platform dependencies.

All statistical calculations appearing in this paper were carried out on a random sample of 100 FIG files from a total of 1,539 files. This sample represents around 6 hours of feed data (from a total of 96 available hours).

Performance tests were carried out on a 1.6 GHz Pentium 4 processor with 512 MB of memory.

Figure 4-1: Statistic calculation and performance testing methods

We ultimately decided to use a database to store feed information on disk. We

felt that the advantages of using a database outweighed its disadvantages. We also felt

that from a development perspective, a database made the most sense. First, there are

many open-source databases available that implement the JDBC API. This implied that

we would not have to spend time developing our own solution. Moreover, FIG's design

allows us to easily substitute a new database implementation at a later time to achieve

better performance, more features, and so forth; the MDDs in Figures 3-2 and 3-5 show

FIG's only two dependencies on the QED database—all other dependencies on the

underlying database implementation are mitigated by the JDBC API. A second

advantage from a development perspective is that the use of databases reduces the cost of

future modifications to the system. Because disk access is handled by the database

mechanism, we can easily extend or change the data schema without also having to

update the access logic. A third advantage is that the popularity of databases means that

there are many 3rd party tools available to aid in our development efforts; as an example,

we used an open-source database viewer—iSQL-Viewer[5]—to debug FIG's recording and playback mechanisms. Note that the one major disadvantage to using a database—the added space overhead—was not as severe as initially expected. As described in section 3.2, FIG uses a compression algorithm to reduce the final size of its FIG files. We calculate that a compressed FIG file is on average only 4 times as large as a corresponding compressed flat file (as opposed to 18 times as large when comparing the uncompressed versions). The other disadvantage that was mentioned—portability and platform-independence—was not an issue since we chose a database that is implemented in 100% Java.

4.2.2 Database Schema

Once we decided to use a database to store feed data on disk, the next decision we faced was choosing an appropriate schema. We considered two variations—a single-table and a dual-table schema. The single table version is summarized in Table 4.1; it is essentially the *Feed* table and *Message* table from Tables 3.1 and 3.2 combined. One advantage of this schema is that it saves space—by consolidating all the information into a single table, it eliminates the overhead associated with multiple tables. Another advantage is that data insertions are fast— recording a message to the database requires accessing only one table. A third advantage is that complex queries are also fast—retrieving messages based on filtering conditions will not require resource-intensive table joins.

---

[5] iSQL-Viewer Homepage: http://www.isqlviewer.com

The Main table is used to store ETMS messages, their frames, and certain pre-parsed fields.
Each row in this table represents a single ETMS message.

Column Name	Description	Type
fts	FIG-assigned timestamp (**primary key**)	timestamp(32)
seqnum	Sequence number	Int
ets	ETMS-assigned timestamp	timestamp(32)
fclty	Facility identifier	char(4)
mtype	ETMS message type	char(2)
aline	Airline referenced in message	char(3)
msg	Complete message text	varchar(0)

Table 4.1: Main table

The dual-table schema was described in Tables 3.1 and 3.2. The main advantage of this schema over the single-table version is that the simplest query—a request for all messages in the database—should execute faster. The reason for this is that the *Feed* table containing the full message texts is smaller than the *Main* table described in Table 4.1; hence, a straightforward query should involve fewer disk accesses and result in better overall performance. Of course, complex queries involving filtering conditions will take longer in this schema since their execution will require a table join. Essentially, the dual-table design optimizes the common case of playing back complete, unfiltered feeds.

We ultimately decided to use the dual-table schema. At the time of the decision, we lacked the infrastructure to run comprehensive performance tests on both schema types; instead, we went with our intuition and chose to optimize the playback of unfiltered feeds, which we believed would be the most frequently used feature of the system. We have now completed performance testing of both schemas, and in retrospect we believe that the single-table schema would have been a better choice. The results of these tests are summarized in Figure 4-2. The performance of the single-table schema on the execution of the simple query is very close to the dual-table schema's performance.

In other words, the perceived advantage of the dual-table over the single-table schema no longer seems to be valid. Moreover, the single-table schema is *significantly* faster at processing filtered queries. We have also calculated the difference in space requirements for both schemas; we have discovered that FIG files based on the single-table schema are on average 2 times smaller in size, regardless of whether compression is used. In short, the single-table schema seems to offer the best combination of performance and portability.

The performance tests measured the average execution times of both schemas on two types of queries:

- The **simple query** selects all ETMS messages from the database, ordered by the FIG-assigned timestamp.
- The **filtered query** selects only *TZ* messages belonging to American Airlines, ordered by the FIG-assigned timestamp.

The table below summarizes the results of the tests (execution times are measured in milliseconds). Note that the tests were carried out as described in Figure 4-1.

	Single-table Schema	Dual-table Schema
Simple query	1376.8 ms	1346.3 ms
Filtered query	535.62 ms	1186.1 ms

Figure 4-2: Performance test results on schema types

Since we knew that our choice of schema might turn out to be wrong—and to support evolutionary changes to the ETMS feed format—we purposely designed the IO library to isolate the dependency on the data format to as few classes as possible. The FIGFile class contains a set of package-friendly constants that encapsulate the database schema. The class itself uses the schema information in only one place—to create a new

database whenever a brand new FIG file is created. In the FIG file reader abstraction, only the `PhysicalFIGFileReader` and the `FIGFileContentFilter` classes depend on the schema—`PhysicalFIGFileReader` uses basic schema information to parse its query results, and `FIGFileContentFilter` uses the information to translate filtering conditions into SQL queries. Note that in comparison, the `FIGFileReader` class depends only on the public specification of `FIGFile` (as shown in the MDD in Figure 3-7). Finally, in the FIG file writer abstraction, only `PhysicalFIGFileWriter` depends on the schema to insert messages appropriately into the database (see the MDD in Figure 3-5). To sum up, with this design we estimate that the change from a dual-table to a single-table schema can be completed within one hour.

4.2.3 Compression

FIG files are compressed using the ZIP algorithm, which achieves an average compression rate of 9.72. We chose the generic ZIP algorithm over a more content-specific compression algorithm (such as the one used in Gutman [Gut00] and Lin's [Lin02] archival system) because we wanted to minimize the dependence of the compression logic on the rest of the system. From the MDD in Figure 3-2, you can see that the compression mechanism is isolated; there is a dependence of the FIG file on the mechanism, but not vice-versa. This eliminates a shared dependency between the FIG file, FIG file writer, and FIG file reader—namely, a dependence on the data format. As a result, each subsystem can be modified independently—we can easily switch compression algorithms without affecting the rest of the system, and we can alter the main system without propagating those changes to the compression mechanism. We felt

45

that this was a desired feature of the design since we viewed compression as only playing an ancillary role in the system, hence the algorithm used should not have any strong dependencies leading back to the main system.

Note that the use of compression in general added some complexity to the rest of the system. For example, one of the biggest sources of complexity in the `FIGFile` class is the need to expand FIG files before their contents can be accessed. There is also the need to compress these FIG files once they have been closed. To accomplish these tasks, `FIGFile` uses the compression mechanism to expand a file into a temporary local directory; this directory is uniquely identified by the original name of the FIG file. However, given that FIG files are treated like any other files and can thus be renamed at any time, care has to be taken by the user to not rename the original FIG file while it is expanded and its contents are being accessed. The object model in Figure 4-3 makes the relationship between a FIG file, its name, and its expansion directory more explicit.
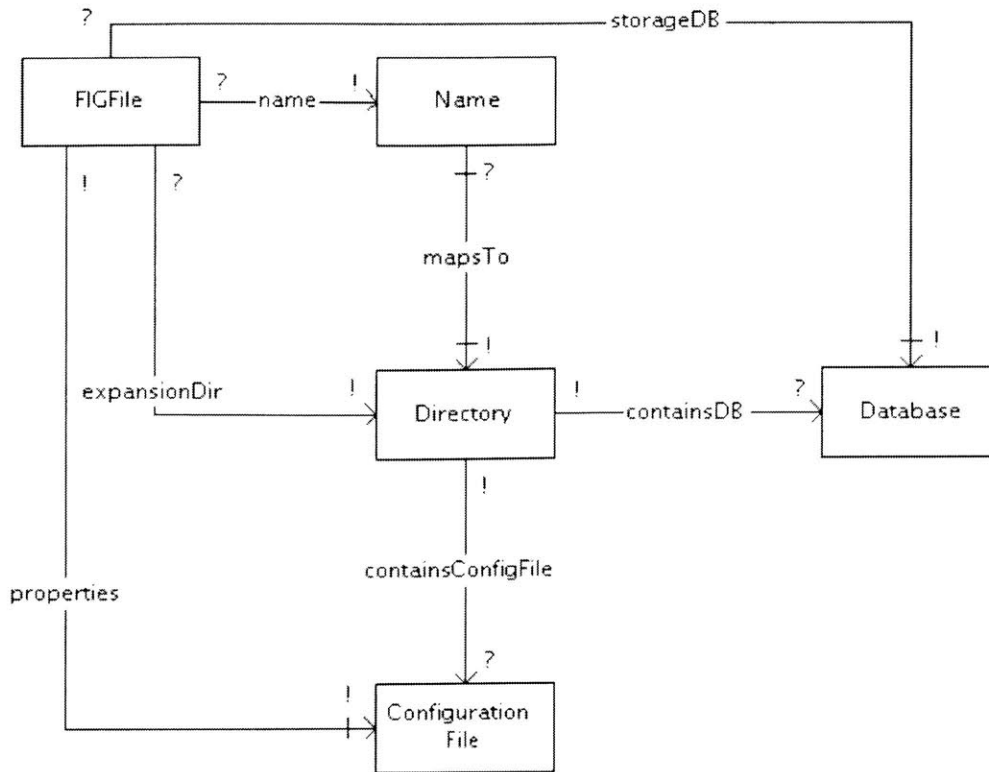
Notes:

all disj f, g: FIGFile | f.name != g.name
    – This OM describes the contents of a single directory in the
    filesystem. Two different FIG files can have the same name if
    they reside in different filesystem directories.

all f: FIGFile | f.name.mapsTo = f.expansionDir

all f: FIGFile | f.expansionDir.containsConfigFile = f.properties

all f: FIGFile | f.expansionDir.containsDB = f.storageDB

Figure 4-3: Object model of expanded FIG file

4.3 FIG File Writer

The FIG file writer is an important part of the final system, since it is the mechanism

through which ETMS feeds can be recorded. We explore the design of this abstraction in

the sections below.

47

4.3.1 PhysicalFIGFileWriter

One of our early design decisions was to simplify the implementation of the writing mechanism by splitting it into two classes—`PhysicalFIGFileWriter` and `FIGFileWriter`. The `PhysicalFIGFileWriter` class contains the logic to record data to a single FIG file, while the `FIGFileWriter` class handles the details of multiple file recordings. The relationship between these two classes is depicted in the MDD in Figure 3-5 and the object model in Figure 4-4:
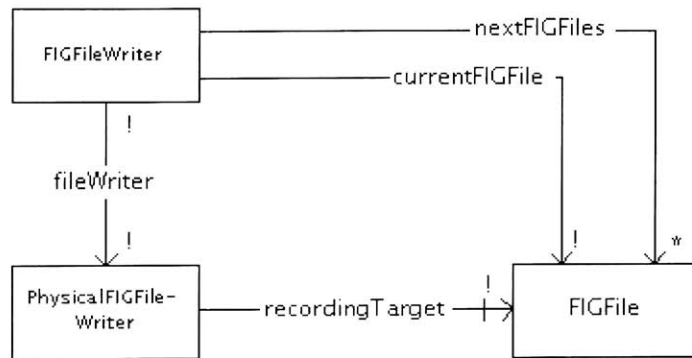
Notes:

all w: FIGFileWriter | w.currentFIGFile !in w.nextFIGFiles

all W: FIGFileWriter | w.currentFIGFile = w.fileWriter.recordingTarget

Figure 4-4: Object model of dual-class writer implementation

Unfortunately, during the implementation phase we discovered a design flaw that ultimately broke the abstraction between the `FIGFileWriter` and `PhysicalFIGFileWriter`. We had incorrectly assumed that isolating the logic of recording messages to disk also entailed isolating the parsing mechanism; as shown in

48

Figure 3-5, the parsing engine is hidden behind the `PhysicalFIGFileWriter` class. A side-effect of this design is that the `FIGFileWriter` class has no way of tracking how many messages are being sent to its `PhysicalFIGFileWriter`. The problem arises when the `PhysicalFIGFileWriter` object reaches the maximum number of messages allowed for its file. It is forced to stop writing messages to disk, but it may still have messages remaining in its buffer. To avoid losing these messages during a multiple file recording, the `FIGFileWriter` must extract these remaining messages and initialize the next `PhysicalFIGFileWriter` object with this data. This creates an undesirable dependence of `FIGFileWriter` on the `PhysicalFIGFileWriter`'s particular buffering mechanism (mediated through the `getMessageBuffer` method, as shown in Figure 3-5), violating the separation of responsibilities between these two classes that was intended by the original design. In retrospect, we feel that the correct design is to have `FIGFileWriter` pass incoming feed data directly to the parsing engine and then pass the resulting messages to the `PhysicalFIGFileWriter`. In this way, the `FIGFileWriter` can keep track of the number of messages being written yet still be abstracted from the actual recording of messages to disk.

Another interesting issue that came up while implementing the `PhysicalFIGFileWriter` class centered on assigning timestamps to recorded messages. Since the FIG-assigned timestamp is the primary key in both the *Feed* and *Message* tables, every recorded message must have a unique timestamp. Unfortunately, Java's system clock returns timestamps accurate up to 10 milliseconds only, which resulted in a number of primary key exceptions during testing. We modified our code to catch these exceptions in order to try another timestamp (as opposed to just failing), but

49

we were forced to inspect every exceptions' message syntax in order to determine which ones truly represented primary key violations[6]. This created FIG's dependence on the QED database's particular exception syntax as shown in Figure 3-5[7]. We eventually added an optimization to the code that remembers the last timestamp that was assigned and ensures that the next timestamp occurs at least one millisecond later. This removed all occurrences of the primary key violation, but the exception handling mechanism previously described has not yet been removed from this module.

4.3.2 Writer Interface

We decided to have the `FIGFileWriter` class implement the `java.io.Writer` interface as opposed to a proprietary interface. Since the FIG file writer was going to be made available to 3rd party applications, it was to our advantage to have the recording mechanism mimic the already familiar Java IO classes. The trade-off we made was the addition of new dependencies into our system. One example is a dependency on the user to not wrap the `FIGFileWriter` object in any IO class that performs buffering (such as the `BufferedWriter` class). The problem with these classes is that they accumulate messages in their buffers before sending them to the `FIGFileWriter`— this causes the FIG file writer to record inaccurate delays between those messages.

---

[6] The JDBC API supports a generic `SQLException`, but it has no standardized method of classifying the root cause of such an exception.

[7] Note that this dependency is caused by a flaw in the JDBC API, hence it would exist for any JDBC-compliant database implementation.

4.3.3 Parser Design

Another choice we made when designing the FIG file writer abstraction was to include only a minimal implementation of an ETMS message parser. This was done to reduce the size and complexity of the writing mechanism, as well as to save development time. It did, however, also limit the capabilities of FIG's filtering mechanism by reducing the set of possible filtering conditions. One notable feature that was left out of the parser was the ability to detect content errors in messages. As a result, correctly formatted messages containing invalid data do not get marked as bad messages, and such messages cannot be automatically detected or filtered from recorded feeds. We had briefly entertained the idea of allowing a 3rd party parser to be added to FIG, but we ultimately rejected this idea due to the difficulty in ensuring the correctness of its parsed messages. However, we now believe that it may be possible to support a smaller 3rd party mechanism that only detects content errors in messages. The mechanism would have the power to mark each parsed message as a bad message—this information would be stored in a separate column in the *Message* table, and users of FIG would have the option of filtering based on this new flag. Note that the inclusion of this mechanism would not affect the quality of the parsed messages, thus ensuring the correctness of the recording mechanism[8].

4.4 FIG File Reader

The design of the FIG file reader is analogous to the design of the FIG file writer in many ways. For example, the reader uses dual-classes—`PhysicalFIGFileReader` and `FIGFileReader`—to simplify the implementation of the reading mechanism. Also,

---

[8] Of course, if the error-detection mechanism contains buggy code, it could cause FIG to crash. However, this is a better outcome than allowing a 3rd party parser to record corrupted data.

the FIG file reader implements the `java.io.Reader` interface to allow the mechanism to be compatible with existing Java IO classes.

There were, however, two notable design issues that were specific to the reader abstraction. The first centered on how to specify filtering conditions when streaming a feed. The second issue concerned the effect of reading delays while playing multiple files. These issues are discussed in more detail below.

4.4.1 Content Filter

One of the functional requirements of FIG is the ability to filter the messages that are streamed from a FIG file. We first considered adding convenience methods to `FIGFileReader` that would allow a user to specify filtering conditions. However, we ultimately decided to isolate the filtering mechanism into its own class (i.e. `FIGFileContentFilter`). This strategy has many advantages. First, it simplifies the implementation of `FIGFileReader`. Second, it allows sets of filtering conditions to be constructed and manipulated independently; an application can capitalize on this ability by providing a library of pre-defined content filters. Third, it allows classes not associated with the reader mechanism to take advantage of the filtering logic in this class; for example, the profiling tool included with FIG uses the filtering logic to allow users to analyze filtered versions of feeds.

One of the interesting design choices we made regarding `FIGFileContentFilter` was to isolate within this class the translation of filtering conditions into SQL queries. This greatly simplified the implementation of the FIG file reader, but it did add a dependency to the system between the content filter and the database schema (as shown by the dependence arrow between

`FIGFileContentFilter` and `FIGFile` in Figure 3-7). It also caused an indirect dependence of the content filter on the correct behavior of the FIG file writing mechanism.

Finally, it should be noted that the design of the content filter is a bit inflexible. When specifying a content filter, all filtering conditions are conjunctive. For example, using the `FIGFileContentFilter`'s `setAirlines` method to specify American Airlines and the `setMessageTypes` method to specify the *TZ* message type results in all messages getting filtered except for TZ messages associated with American Airlines (as shown in Figure 3-6). There is currently no way to specify a filter that includes only TZ messages *or* any messages belonging to American Airlines.

4.4.2 Multiple File Streaming

There is a delay that occurs between when a FIG file is first opened and when its contents can be accessed. This delay is caused by the need to first decompress and then expand the FIG file into its expansion directory; there is also the delay associated with executing a query against this FIG file's database. While this startup penalty is not really an issue when streaming a single file, it does create a noticeable lag during the playback of multiple files.

There are two possible solutions to this problem. The first solution is to minimize the size of the files included in a multiple file playback. A smaller file size reduces both the decompression time and the query execution time. However, this solution is up to the user to implement, and its effectiveness can vary across platforms and hardware configurations. A better solution is to implement a multi-file look-ahead mechanism in `FIGFileReader`. Currently, `FIGFileReader` maintains a connection to only one

FIG file at a time—the current FIG file must first be closed before the next FIG file can be opened for reading. The multi-file look-ahead mechanism would in contrast maintain open connections to multiple files in the playback list—the goal is to amortize the cost of opening a new FIG file over time. We feel that the second solution is the right choice, but it has not yet been implemented in the current version of FIG.

Chapter 5

Conclusion

5.1 Retrospective

Designing a recording and playback mechanism for ETMS feeds turned out to be a much harder problem than first anticipated. We initially thought that implementing the tool would simply involve recording incoming messages to a text file and then playing them back. However, as we began to specify the exact functionality of the application, we realized that the requirement to maintain the portability and fidelity of recorded feeds necessitated a much more careful design. As a result, we were forced to postpone a lot of the functionality originally planned for the tool; some of these proposed features are mentioned in the next section regarding possible extensions to FIG.

Fortunately, we compensated for this loss of functionality by creating a flexible design for FIG that could incorporate any anticipated future needs. For example, we feel that our decision to use a database as the underlying storage mechanism for FIG files was the correct choice. Although we did make a mistake—mostly in the interest of making progress—of using a dual-table schema rather than a single-table schema, our use of the database makes the task of changing data formats relatively easy. Another example of FIG's flexible design is the use of versioning in FIG files—if we do eventually change the data format, the version number stored in each FIG file's configuration properties will allow FIG to maintain backwards compatibility. As you can see, we are already experiencing the benefits of the careful design we put into the application.

5.2 Future Directions

There are a number of useful and interesting extensions to FIG. Examples include:

- *Modification of feeds* – New functionality can be added to the IO library to support the modification of recorded feed data. This might include the ability to merge two FIG files, reorder the messages in a feed, or edit the content of individual messages.

- *Visualization of feeds* – An alternative to streaming recorded feeds to 3rd party applications is to load them into FIG and provide various visualizations of the data. For example, a user might wish to view a three-dimensional snapshot of a recorded feed's information. Not only could this mechanism be used by software testers to better understand their input feeds, but it could also be used to support a more user friendly method of modifying the information in a recorded feed.

- *Generated scenarios* – One way to enhance FIG's usefulness as a testing tool is to design a mechanism to generate fictitious ETMS feed data based on high-level specifications. For example, a user might wish to introduce a fictitious flight into a recorded feed in order to force a conflict situation to occur. The mechanism would allow the user to specify the time, place, and nature of the collision, and it would then use these specifications to generate plausible scenarios.

While we feel FIG is useful in its current implementation, we hope that the lessons learned from this paper will help motivate efforts to further expand the functionality of this tool.

Chapter 6

Appendix

6.1 IO Library API

public class **fig.io.FIGFile** implements java.io.Serializable

A file containing pre-recorded messages from an ETMS feed.

Note that FIG files are compressed to save space. As a result, a FIG file will expand into a temporary directory during its use. To avoid possible file corruption, do not access or modify any files in this temporary directory.

Constructors

public FIGFile(String filename)
Constructs a new FIG file.
Parameters
      filename - the name of the FIG file
Throws
      NullPointerException - if filename is null

public FIGFile(File file)
Constructs a new FIG file.
Parameters
      file - the FIG file
Throws
      NullPointerException - if file is null

Methods

public java.io.File getFile()
Returns the physical file that represents this FIG file.
Returns
      a File object

public boolean exists()
Returns true if this FIG file exists; false otherwise.
Returns
      true if this FIG file exists; false otherwise

public boolean isValid()
Returns true if this file is a valid FIG file.
Returns
      true if this file is a valid FIG file; false otherwise

public int hashCode()

public boolean equals(Object o)

public java.lang.String toString()

Fields

public static final FIG_EXTENSION
The FIG file extension.

public class **fig.io.FIGFileContentFilter** implements java.io.Serializable, java.lang.Cloneable

Filters the content that is read from a FIG file. You can customize the filtering constraints that determine which ETMS messages get filtered.

Constructors

public FIGFileContentFilter()
Constructs a content filter that incorporates delay and filters nothing.

Methods

public java.lang.Object clone()

public int hashCode()

public boolean equals(Object o)

public java.lang.String toString()

public void resetFilteringConditions()
Resets this content filter such that it incorporates the original message delays and does not filter any messages from the feed.

public void setConstantDelay(long delay)
Sets the constant delay between consecutive messages. A constant delay of 0 implies that there will be no delay between messages. If the specified delay is less than zero, the original (variable) message delays will be used instead.
Parameters
        delay - a non-negative value representing the constant delay between consecutive messages (in milliseconds); a negative value causes the original (variable) message delays to be used instead

public long getConstantDelay()
Returns the constant amount of delay (in milliseconds) between consecutive messages. If the original variable message delays are being used instead, returns a negative value.
Returns
        the constant amount of delay between consecutive messages, or a negative value if the original variable message delays are being used instead

public void setDelayScalar(double scalar)
Sets the amount by which the delay between consecutive messages (whether variable or fixed) will be scaled. A scalar of 2 indicates that the messages will be played twice as slow. A scalar of 0.5 indicates that the messages will be played twice as fast. A scalar of 1 indicates that the actual delay will remain unchanged. A scalar of 0 indicates that there will be no delay between messages. Note that a scalar less than 0 will be interpreted as a 0.
Parameters
        scalar - the amount by which the delay should be scaled; any value less than 0 will be treated as a scalar of 0

public double getDelayScalar()
Returns the amount by which the delay between consecutive messages (whether variable or fixed) will be scaled.
Returns
        the amount by which the delay will be scaled

public void setFilterBadMessages(boolean filter)
If filter is true, badly formatted messages will be filtered out during playback.
Parameters
        filter - true to filter out badly formatted messages

public boolean getFilterBadMessages()

Returns true if badly formatted messages will be filtered out during playback; false otherwise.
Returns

       Returns true if badly formatted messages will be filtered out during playback; false otherwise

public void setAirlines(String[] newAirlines)

If the list of airline abbreviations is not empty, only messages belonging to those airlines will be included during playback. If the list is empty, messages from all airlines will be included.

Note that the airline abbreviations are typically the 3-letter ICAO airline codes. To represent private aircraft that do not officially belong to an airline, use the abbreviation --this abbreviation is used because the aircraft IDs of these private aircraft all begin with the single letter "N". In general, if the airline is less than three characters in length, pad the airline abbreviation with spaces until it is of length 3.
Parameters

       newAirlines - a list of airline abbreviations

public java.lang.String[] getAirlines()

Returns the list of acceptable airlines. If all airlines are acceptable, returns an empty array.
Returns

       array containing airline abbreviations

public void setFacilities(String[] newFacilities)

If the list of facility IDs is not empty, only messages belonging to those facilities will be included during playback. If the list is empty, messages from all facilities will be included.

Note that the facility abbreviations are the 4-letter identifiers defined in the ETMS specifications. Consult the ETMS (ASDI) feed documentation for more details.
Parameters

       newFacilities - a list of facility IDs

public java.lang.String[] getFacilities()

Returns the list of acceptable facilities. If all facilities are acceptable, returns an empty array.
Returns

       array containing facility IDs

public void setMessageTypes(String[] newTypes)

If the list of message types is not empty, only messages with those message types will be included during playback. If the list is empty, messages of all types will be included.

The acceptable message types are defined in the ETMS specifications. Consult the ETMS (ASDI) feed documentation for more details.
Parameters

       newTypes - a list of message types

public java.lang.String[] getMessageTypes()

Returns the list of acceptable message types. If all types are acceptable, returns an empty array.
Returns

       array containing message types

public class **fig.io.FIGFileReader** extends java.io.Reader implements java.io.Serializable, java.lang.Comparable

Opens any number of FIG files for reading. When the messages from one FIG file have been exhausted, the next FIG file in the list (if any) will be opened.

It is important that you close this reader once you are done to avoid corrupting the FIG file(s).

Constructors

public FIGFileReader(String filename)
Constructs a new FIG file reader.
Parameters
    filename - the name of the FIG file
Throws
    IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(String[] filenames)
Constructs a new FIG file reader for multiple files.
Parameters
    filenames - the names of the FIG files, in the order in which they should be read
Throws
    IOException - if the files cannot be found or if some IO error occurs

public FIGFileReader(String filename, FIGFileContentFilter filter)
Constructs a new FIG file reader with the given content filter.
Parameters
    filename - the name of the FIG file
    filter - the content filter
Throws
    IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(String[] filenames, FIGFileContentFilter filter)
Constructs a new FIG file reader for multiple files with the given content filter.
Parameters
    filenames - the names of the FIG files, in the order in which they should be read
    filter - the content filter
Throws
    IOException - if the files cannot be found or if some IO error occurs

public FIGFileReader(File file)
Constructs a new FIG file reader.
Parameters
    file - the FIG file
Throws
    IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(File[] files)
Constructs a new FIG file reader for multiple files.
Parameters
    files - the FIG files, in the order in which they should be read
Throws
    IOException - if the files cannot be found or if some IO error occurs

public FIGFileReader(File file, FIGFileContentFilter filter)
Constructs a new FIG file reader with the given content filter.
Parameters
    file - the FIG file
    filter - the content filter
Throws
    IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(File[] files, FIGFileContentFilter filter)
Constructs a new FIG file reader for multiple files with the given content filter.
Parameters
>files - the FIG files, in the order in which they should be read
>filter - the content filter

Throws
>IOException - if the files cannot be found or if some IO error occurs

public FIGFileReader(FIGFile file)
Constructs a new FIG file reader.
Parameters
>file - the FIG file
Throws
>IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(FIGFile[] files)
Constructs a new FIG file reader for multiple files.
Parameters
>files - the FIG files, in the order in which they should be read
Throws
>IOException - if the files cannot be found or if some IO error occurs

public FIGFileReader(FIGFile file, FIGFileContentFilter filter)
Constructs a new FIG file reader with the given content filter.
Parameters
>file - the FIG file
>filter - the content filter
Throws
>IOException - if the file cannot be found or if some IO error occurs

public FIGFileReader(FIGFile[] files, FIGFileContentFilter filter)
Constructs a new FIG file reader for multiple files with the given content filter.
Parameters
>files - the FIG files, in the order in which they should be read
>filter - the content filter
Throws
>IOException - if the files cannot be found or if some IO error occurs

Methods

public void close()

public void reset()

public boolean markSupported()

public boolean ready()

public int read(char[] cbuf, int off, int len)

public fig.io.FIGFile[] getFiles()
Returns the FIG files to be read by this reader. If there are no files, returns the empty array.
Returns
>an array containing the FIG files in the order in which they will be read; note that modifying the array will have no effect on this reader

public fig.io.FIGFileContentFilter getContentFilter()
Returns the content filter associated with this reader.
Returns
>the content filter associated with this reader; note that modifying the filter will have no effect on this reader's content filter

public int hashCode()

public boolean equals(Object o)

public java.lang.String toString()

public int compareTo(Object o)

public class **fig.io.FIGFileWriter** extends java.io.Writer

Opens a FIG file for writing. This writer expects messages from an ETMS feed as input. Note that incorrectly formatted messages will also be recorded to accurately reflect the input feed.

You can optionally specify a message limit for the maximum number of messages to record to a single file. When the number of messages in the current file exceeds this limit, another file is opened in its place. This allows you to record a large number of messages to a manageable set of FIG files. Note that any files created due to message overflow will have the original file's name with the message limit and a timestamp (accurate up to a millisecond) appended to it.

It is important that you close this writer once you are done to avoid corrupting the FIG file(s).

Constructors

public FIGFileWriter(String filename)
Constructs a new FIG file writer.
Parameters
        filename - the name of the FIG file
Throws
        IOException - if some IO error occurs

public FIGFileWriter(String filename, long messageLimit)
Constructs a new FIG file writer with the specified message limit.
Parameters
        filename - the name of the FIG file
        messageLimit - the maximum number of messages to store in a single FIG file; any value less than 1 will be interpreted as no limit
Throws
        IOException - if some IO error occurs

public FIGFileWriter(File file)
Constructs a new FIG file writer.
Parameters
        file - the FIG file
Throws
        IOException - if some IO error occurs

public FIGFileWriter(File file, long messageLimit)
Constructs a new FIG file writer with the specified message limit.
Parameters
        file - the FIG file
        messageLimit - the maximum number of messages to store in a single FIG file; any value less than 1 will be interpreted as no limit
Throws
        IOException - if some IO error occurs

public FIGFileWriter(FIGFile file)

Constructs a new FIG file writer.
Parameters
> file - the FIG file

Throws
> IOException - if some IO error occurs

public FIGFileWriter(FIGFile file, long messageLimit)

Constructs a new FIG file writer with the specified message limit.
Parameters
> file - the FIG file
>
> messageLimit - the maximum number of messages to store in a single FIG file; any

value less than 1 will be interpreted as no limit
Throws
> IOException - if some IO error occurs

Methods

public void close()

Closes this output stream. Note that if an incomplete message remains in the buffer, it will be thrown away (the assumption is that the input feed terminated prematurely as opposed to the last message being incorrectly formatted).
Throws
> IOException - if an IO error occurs

public void flush()

public void write(char[] cbuf, int off, int len)

public int hashCode()

public boolean equals(Object o)

public java.lang.String toString()

Chapter 7

Bibliography

[Erz01]   Heinz Erzberger. *The Automated Airspace Concept*. Paper #160, 4th
            USA/Europe Air Traffic Management R&D Seminar. Santa Fe, NM.
            Dec 3-7, 2001.

[GHJV95]   Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design
            Patterns: Elements of Reusable Object-Oriented Software*. Addison-
            Wesley Professional Computing Series. Addison-Wesley Publishing
            Company. New York, NY. 1995.

[Gut00]   Micah Gutman. *A Robust Flight-Based Archival System for the Aircraft
            Situational Display to Industry*. MEng Thesis. MIT, Cambridge, MA.
            2000.

[Jac02]   Daniel Jackson. *Lecture 9: Dependences and Decoupling*. 6.170 lecture
            notes. MIT, Cambridge, MA. Fall 2002.

[Lin02]   Joyce Lin. *VisualFlight: The Air Traffic Control Data Analysis System*.
            MEng Thesis. MIT, Cambridge, MA. 2002.

[Par79]   David L. Parnas. *Designing software for ease of extension and
            contraction*. IEEE Transactions on Sofware Engineering, SE-5, 2, 1979.

[Vol00]   Volpe Center, Automation Application Division, DTS-56. *Aircraft
            Situation Display to Industry, Functional Description and Interfaces*.
            4.0 edition. Cambridge, MA. August 2000.