REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
AN INTELLIGENT USER INTERFACE TO SUPPORT AIR FORCE WEATHER PRODUCT GENERATION AND AUTOMATED METRICS

5. FUNDING NUMBERS

6. AUTHOR(S)
Darryl N. Leon, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/00M-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ AFWA/XP
Attn: Mr. George Coleman
106 Peacekeeper Drive, Suite 2N3
Offutt AFB, NE 68113-4039

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Major Michael L. Talbert, ENG, DSN: 785-3636, ext. 4280

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Air Force pilots require dependable weather reports so they may avoid unsafe flying conditions. In order to better gauge the accuracy of its weather products, Air Force Weather has established the requirement for an Air Force-wide automated weather metrics program. Under the guidelines for this program, forecasts will automatically be compared to observed weather to determine their accuracy. Statistics will be collected in the hopes of determining forecast error trends that can be corrected through education and training. In order for the statistical data produced by such a program to draw reliable conclusions about forecast accuracy, however, the correct format of the raw forecasts and observations must be ensured before the reports are disseminated.

Beyond a simple check for typographical errors, however, the system must also have weather domain knowledge to understand when the input data content does not fit the context of the report, even though it has been formatted properly. This thesis proposes the application of an intelligent user-interface "critic" advice system, to ensure not only correct product format and provide content quality control, but to collaborate with and advise the forecaster or observer during product generation, with the ultimate goal of producing more accurate weather products.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Intelligent User Interface, Critic Systems, Weather Forecasting, Weather Forecast Verification, Automated Verification, Weather Forecast Metrics, Automated Metrics	212
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

An Intelligent User Interface to Support Air Force Weather Product Generation and Automated Metrics

THESIS

Darryl N. Leon, Captain, USAF

AFIT/GCS/ENG/00M-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Approved for public release; distribution unlimited
DTIC QUALITY INSPECTED 4

AFIT/GCS/ENG/00M-15

The views expressed in this thesis are those of the author
and do not reflect the official policy or position of the
United States Air Force, the Department of Defense
or the U.S. Government.

AFIT/GCS/ENG/00M-15

An Intelligent User Interface to Support Air Force Weather Product Generation and Automated Metrics

Thesis

Presented to the Faculty

Department of Systems and Management

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

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

Darryl N. Leon, B.S.

Captain, USAF

March 2000

20000815 168

Approved for public release, distribution unlimited

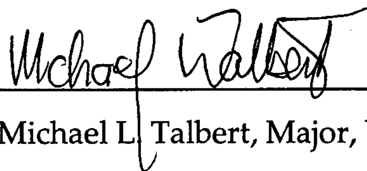
AFIT/GCS/ENG/00M-15

An Intelligent User Interface to Support Air Force Weather Product Generation and Automated Metrics

Darryl N. Leon, B.S.

Captain, USAF

Approved:



Michael L. Talbert, Major, USAF (Chairman)

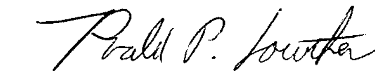
1 Mar 2000

Date

Cecilia A. Miner, Lt Col, USAF

1 Mar 2000

Date



Ronald L. Lowther, Lt Col, USAF

18 Feb 00

Date

Approved for public release, distribution unlimited

For Dakota
who makes it all worthwhile.

Acknowledgments

It would have been impossible to complete a project such as this without the support, guidance, and advice of my colleagues, committee members, professors, sponsors, and family.

I'd first like to express sincere thanks to my thesis advisor, Maj Michael Talbert, for taking a chance on this unique research endeavor, and also for giving me the opportunity to pursue a computer science master's degree though my undergraduate degree was in meteorology.

Maj Talbert has a masterful ability to perceive a complex problem from various angles, analyze it, and unearth its simple components. It was just this sort of insightful guidance that helped me keep my focus—and my sanity—from the start of this project to the end.

Additionally, I'd like to thank my other committee members, Lt Col Cecilia Miner and Lt Col Ronald Lowther. Their weather domain knowledge and literary advice greatly improved the quality of my work. The TAF Verification Program background material provided by Lt Col Lowther proved to be especially useful to my research.

Though not on my committee, Dr. Thomas Hartrum deserves special thanks, as well. Not only did he allow me to build the design specification model that supports my research as part of the class project for his "Introduction to Software Engineering" course, but he also spent an inordinate amount of extra time critiquing my design, helping to ensure its proper development.

I'd also like to express my sincere appreciation to Capt Meriellen Joga and Capt Jim Douglas for tackling other aspects of the complex issues involved with a centralized, automated metrics program. Meriellen and Jim are two of the most ambitious, dedicated, and intelligent people I've ever met. I've learned a lot from them, and it has been a pleasure working with both of them.

I am very grateful to Brg. Gen. Fred Lewis for the opportunity to focus my research on this important aspect of Air Force Weather. Additionally, I'd like to thank Lt Col Larry Key, Mr. George Coleman, and Mr. John Zapatocny for their support in this effort, not only arranging for first-hand tours of Air Force Weather Agency production systems, but for ensuring the meteorological and historical accuracy of this document.

Finally, and most importantly, I'd like to give special thanks to my mother, Coralee; my father, Noel; my grandmother, Cora; and my wife, Keri for their undying love and support. I'd especially like to thank my mom for her expert editorial advice (free of charge from the time I was eight-years-old!), and Keri for feeding me whenever I forgot to eat.

Table of Contents

Acknowledgments iv

List of Figures ix

List of Tables x

Abstract xi

1 Introduction 1

1.1 Air Force Weather Reengineering 1
1.1.1 Forecast Responsibility 2
1.1.2 The Operational Weather Squadrons 2
1.2 Problem Background 3
1.2.1 Product Quality Control 4
1.3 Research Focus 7
1.3.1 Format QC 7
1.3.2 Content QC 8
1.3.3 Forecast Advice 8
1.3.4 Real-time Forecast Verification 9
1.3.5 The Modified TAF Generation Process 10
1.3.6 Design Considerations 12
1.4 Outline of this Document 13

2 Summary of Current Knowledge 14

2.1 Overview 14
2.2 Computer-Human Interaction 14
2.2.1 Intelligent User Interfaces 15
2.2.2 Interface Agents and Autonomous Agents 16
2.2.3 What Makes an Interface Intelligent? 16
2.2.4 Wizards and Guides 17
2.2.5 Critics 18
2.2.6 Critic Application to the Weather Problem 22
2.3 Rule-Based Expert Systems 23
2.3.1 Rules and Facts 24
2.3.2 Knowledge-Base and Inference Engine 25
2.3.3 The Java Expert System Shell 26
2.4 Weather Domain Background 32
2.4.1 The TAF Verification Program 32
2.4.2 The METAR Weather Code 38
2.4.3 Weather System Integration 38
2.5 Summary 41

3 Design Methodology 42

3.1 Overview 42
3.2 User Interface Design 43
3.2.1 Design Tools and Language Paradigm 43
3.2.2 Design Concept and Focus 44
3.2.3 Report Generation Process 44
3.2.4 System Components 51
3.3 Summary 68

4 Implementation and Functional Analysis 69

4.1 Overview 69
4.2 Interface Implementation 70
4.2.1 Language 71
4.2.2 JESS 71
4.2.3 Weather Data Object Attributes 72
4.2.4 Report Elements 76
4.2.5 Warning Messages and Report Status 80
4.2.6 Interface Display 81
4.2.7 Implemented Functions 84
4.2.8 JESS Rules 85
4.2.9 System Process 87
4.3 Implementation Testing and Analysis 89
4.3.1 System Analysis 90
4.3.2 AFW TAF and METAR Observation Analysis 91
4.3.3 AFW System Integration 96
4.4 Summary 97

5 Summary and Conclusions 99

5.1 Overview 99
5.2 System Benefits 99
5.2.1 Improved Forecast Accuracy 100
5.2.2 Improved Forecaster Ability 101
5.2.3 Operational Use of Laboratory Research 102
5.2.4 Analysis of Research Results 102
5.3 Future Research 103
5.3.1 Critic Research 103
5.3.2 Meteorological Research 104
5.3.3 Other Areas for Research 105
5.4 Summary 105

Appendix A – METAR Coding Example/Explanation 107

Appendix B – JESS Rules Implementation 112

Appendix C – Design Specification Model 119

Appendix D – Specification Data Dictionary 123

Bibliography 195

Curriculum Vitae 198

List of Figures

Figure 1-1. Current process for TAF generation. 6
Figure 1-2. Proposed process for TAF generation 11
Figure 2-1: The Critiquing Cycle [20] 21
Figure 2-2. Basic Concept of an Expert System Function [23] 25
Figure 2-3: Rete Match Algorithm pattern tree [19] 28
Figure 2-4: Optimized Rete Match Algorithm tree structure [19] 29
Figure 3-1: User Interface Design Concept 45
Figure 3-2: Report Template-Weather Report Relationship 46
Figure 3-3: JESS Inference Engine Design 48
Figure 3-4: System Template and Report Repositories 49
Figure 3-5: Base-Type and Category Sets 52
Figure 3-6: Base and Category UML diagram segment 53
Figure 3-7: SystemMaintenanceInterface UML diagram segment 54
Figure 3-8: Rule Set Repository 55
Figure 3-9: Template Repository 57
Figure 3-10: WeatherReportTemplate and TemplateRepository UML diagram segment 59
Figure 3-11: *WeatherReport* Hierarchy 65
Figure 3-12: Concurrent Weather Sets and the External Source Repository 67
Figure 4-1: Simplified *WeatherReport-WeatherCondition* object hierarchy 75
Figure 4-2: Sample report element GUI components 77
Figure 4-3: Screen-capture depicting the display for the user interface 83
Figure B4: System Maintenance Design Specification 120
Figure B5: System Design Specification Model 121
Figure B6: *ReportElement* Object Subclasses 122

List of Tables

Table 2-1: TAFVER ceiling and visibility categories [26] 34
Table 2-2: TAFVER Sample Output [26] 35
Table 2-3: TAFVER II Sample Output (Ceiling < 200 feet at 3-hr point) [10] 36
Table 4-1: TAF and METAR observation report corrections 92
Table 4-2: TAF Correction Totals 93
Table 4-3: METAR Observation Correction Totals 95

Abstract

Air Force pilots require dependable weather reports so they may avoid unsafe flying conditions. In order to better gauge the accuracy of its weather products, Air Force Weather has established the requirement for an Air Force-wide automated weather metrics program. Under the guidelines for this program, forecasts will automatically be compared to observed weather to determine their accuracy. Statistics will be collected in the hopes of determining forecast error trends that can be corrected through education and training. In order for the statistical data produced by such a program to draw reliable conclusions about forecast accuracy, however, the correct *format* of the raw forecasts and observations must be ensured before the reports are disseminated.

Beyond a simple check for typographical errors, however, the system must also have weather domain knowledge to understand when the input data *content* does not fit the context of the report, even though it has been *formatted* properly.

This thesis proposes the application of an intelligent user-interface "critic" advice system, to ensure not only correct product format and provide content quality control, but to collaborate with and advise the forecaster or observer during product generation, with the ultimate goal of producing more accurate weather products.

An Intelligent User Interface to Support Air Force Weather Product Generation and Automated Metrics

1 Introduction

Air Force Weather (AFW) is undergoing a comprehensive reengineering. Under the new design, forecast responsibility will shift from individual weather flights (WFs) to Operational Weather Squadrons (OWSs) or regional "hubs." Simultaneously, an entirely new, globally integrated systems architecture is being defined. A "System Requirements Document (SRD) for the Reengineered Air Force Weather Weapon System (AFWWS)" [15] to define the new architecture has been drafted by the Electronic Systems Center, Air Force Weather Systems (ESC/ACW), Hanscom AFB, Massachusetts.

A major goal of the reengineering effort is improved forecast accuracy. For this reason, the SRD lists automated metrics as a specific requirement. Automated metrics will provide a flexible and dynamic way to measure forecast accuracy, and to identify areas for improvement. If automated metrics are to provide useful information, however, forecast and observation data must be reliable. Because of this, the capability to correct erroneous forecasts and to automatically verify forecasts on a real-time basis before they are submitted becomes critical. This thesis describes the conceptual design, prototyping, and evaluation of a user interface that would provide this function and pave the way for reliable, useful automated metrics.

1.1 Air Force Weather Reengineering

Air Force Weather launched its reengineering effort to better exploit technological advances so that it may provide improved service to its customers. It involves a comprehensive overhaul of all aspects of weather product

production and dissemination. Areas under revision range from organizational structure and information flow to recommendations for member career paths [1].

1.1.1 Forecast Responsibility

One of the most significant changes under the new system is forecasting responsibility. Before reengineering, with the exception of large-scale (regional, hemispheric, and global) forecasts, the daily forecast production responsibility has been at the WF level, at the base weather stations (BWSs). A duty forecaster at the BWS produced and disseminated all forecasts for which his weather station was responsible. A primary forecast responsibility of the WF has been *resource protection*, which involves the issuance of *weather watches, weather warnings* and *weather advisories* during severe weather events. Additionally, forecasters at most bases have been responsible for producing *Terminal Aerodrome Forecasts* (TAFs)—24-hour coded forecasts used by inbound and outbound aircrews to assess local weather conditions. (See §2.4.2 for more information on the weather code).

In addition to TAFs, the duty forecaster has usually been responsible for producing a variety of other products such as forecasts for nearby drop zones, landing zones, and air refueling routes, as well as specialized reports, such as target weather. Under the reengineered AFW, the production responsibility for all of the products mentioned above will shift to forecasters at regional hubs.

1.1.2 The Operational Weather Squadrons

Each of these hubs, or OWSs, will be assigned an area of responsibility (AOR). The mission of each hub will be "to provide theater-scale battlespace forecasts, drop zone/range/air refueling forecasts, fine-scale target forecasts, and issue weather warnings and terminal forecasts for Air Force and Army installations within their area of responsibility" [1]. Though the weather observations (to include observed weather advisories and warnings such as

reports of lightning within close proximity to the base) will still be produced and disseminated at the base level, the hub will take over all forecasting responsibility—to include resource protection—for that particular AOR.

There will be 11 regional hubs worldwide. In the CONUS, Davis Monthan AFB, Arizona; Barksdale AFB, Louisiana; Scott AFB, Illinois; and Shaw AFB, South Carolina will serve as the OWSs. Additionally, Patrick AFB, Florida and Vandenberg AFB, California will serve as OWSs supporting Air Force Space Command. Overseas, Sembach AB, Germany will be the OWS supporting United States Air Forces Europe (USAFE), and the Pacific will be covered by OWSs at Yongsan AB, Korea; Yokota AB, Japan; Pearl Harbor, Hawaii; and Elmendorf AFB, Alaska [1].

1.2 Problem Background

The process by which AFW has measured its forecast accuracy in the past is dated and inflexible. One reason for this is that systems previously employed to create and submit forecasts were not integrated with systems used to track forecasting trends. In some Air Force Major Command (MAJCOM)-driven forecast verification program implementations, forecasters had to manually annotate forecasts and verification observations on a clipboard. The hand-written data were later manually compiled—usually into a Microsoft Access database file or spreadsheet. Meanwhile, efforts to establish an automated, Air Force Weather-wide centralized verification program failed to measure up to expectations (See §2.4.1 regarding the TAFVER program).

Because of this, even MAJCOM-driven programs had to ensure that any tracked metrics were extremely simple. In fact, because it had been such a labor-intensive process, aside from metrics related to severe storm events (thunderstorms, wind gusts, etc.) or those tracking typographical format errors, only a single forecast metric has typically been tracked (in two categories): cloud

ceiling above or below 1500 feet, and visibility above or below 3 miles. Under these *manual* systems, more detailed metrics were just not feasible, since more complex methods would have increased the workload of already task-saturated forecasters.

1.2.1 Product Quality Control

The re-designed, globally integrated, open-ended architecture at the heart of the reengineering effort provides a mechanism by which this and other related problems can be addressed. To begin with, because the system will be entirely integrated, automated metrics become possible: The system used to create the forecasts can be easily integrated into the system that evaluates forecast accuracy. For automated metrics to be reliable, however, the issues of forecast *format* accuracy must also be addressed. Currently, no automated *front-end* quality control (QC) is conducted on any AFW weather products.

1.2.1.1 The Automated Weather Distribution System

The Automated Weather Distribution System (AWDS) used to produce weather reports of all kinds at the WFs was contracted by Air Force weather in the late 70s, and functionally established as a viable system in 1984 [14]. Since that time, AWDS has seen several version changes and upgrades. It was initially contracted from GTE as a single unit, software and hardware together, such that incremental upgrades required entire software—and sometimes hardware—rebuilds. Its standard for information interchange called "Appendix 30" was proprietary, and could not easily be integrated with other weather data systems. Currently shipped on a Sun™ Sparc workstation, the AWDS system provides forecasters and observers with an integrated tool to perform graphical and textual weather product analysis, creation, and dissemination.

The AWDS text product production interface contains the necessary fields for TAF and observation composition, as well as a limited macro capability for

creating standardized forecast reports, such as warnings and advisories. The AWDS interface does not, however, provide any useful QC capability. This means that forecasts with format errors can be (and are) submitted into the global network.

1.2.1.2 N-TFS and AMIS

With reengineering come more enhancements to AWDS. Unlike previous version changes, however, the current upgrade involves a fundamental change in philosophy from the old proprietary standard to a more open architecture. Specifically, through a series of hardware and software upgrades, AWDS itself will evolve into what is now termed the "New Tactical Forecast System" (N-TFS). The most notable change in the architecture is a switch from the UNIX-based Suns to a Windows NT-based system.

Part of the software portion of the upgrades (in progress as of this writing) is called the "Advanced Meteorological Information System" (AMIS). AMIS not only provides a more "user-friendly" Windows-type graphical user interface (GUI) for forecast and observation report generation, but it also conforms to Department of Defense standards of information interchange, rather than proprietary ones.

It is important to note, however, that though it certainly provides an easier and more functional interface, its inclusion in the system reengineering is for the primary purpose of replacing the old Appendix 30 standard, making AWDS (and N-TFS) Year 2000 compliant, and addressing data security issues. AMIS does not provide any more product quality control capability than its predecessor.

1.2.1.3 The TAF Generation Process

Figure 1-1 presents a high-level depiction of the TAF production process. When a forecaster produces a TAF, the current policy in most WFs is for

someone else, usually a senior forecaster, Station Chief (senior NCO), or perhaps the observer, to examine the report for errors. In reality, most of the time everyone else is just as busy as the forecaster, and a second pair of eyes doesn't see the product until an error report is generated and sent down from the MAJCOM level to the flight commander. No automatic QC is done at all.

When the forecaster sends the TAF, a copy is stored on a local database. Typically TAFs (and observations) are stored for 24 hours. The original intention was to keep the TAFs for verification purposes. In practice, TAFs are rarely retrieved and examined unless someone notices an error.[i] Even then, a print out of the feedback copy (a duplicate of the TAF generated by AFWA and sent back to the sending WF) is usually used for future reference.

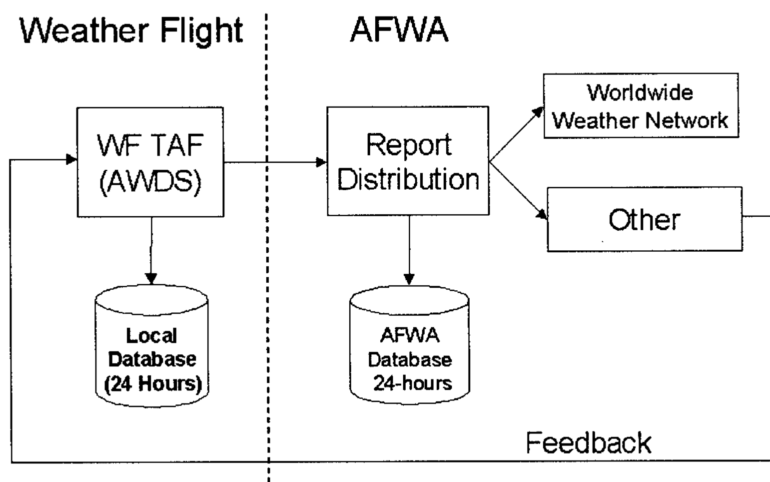


Figure 1-1. Current process for TAF generation.

At The Air Force Weather Agency (AFWA) at Offutt AFB, Nebraska, algorithms are employed to parse and decode TAFs and disseminate them throughout the weather community. Inevitably, forecasts make it into the global weather network with format errors.

---

[i] From time to time, the duty forecaster will receive courtesy phone calls from stations at other bases if the TAF was used to brief a pilot, for instance, and an error was noticed.

Once the TAF is processed by the *back end* system at AFWA, as mentioned above, a feedback copy is returned to the WF, and is filed for future reference.

1.3 Research Focus

As mentioned initially, the focus of this research is the design, prototyping and evaluation of a user interface to be used by forecasters (and observers) to produce and disseminate weather products. Although the results of this research may be generalized to apply to the production of many AFW forecast products, the specific process examined here is TAF generation and QC.

1.3.1 Format QC

Format QC is the process by which user input is validated based on the typographical rules of a particular language or code. In the case of a TAF or an observation, weather information input by the forecaster or observer must comply with the METAR code standard. (See §2.4.2 and Appendix A for more information on the METAR code format). As an example, in a forecast, if a forecaster wanted to convey that there would be a cloud layer with a 3,000 foot base occupying 5/8ths of the sky, to do so following the METAR format, he or she would use the syntax "BKN030" (in the appropriate place in the report).

As will be discussed later in Chapter 3, as implemented in the user interface design, each individual element in a particular weather report (a TAF for instance) should be self-validating. That is, when the forecaster enters data into a text box for a particular element, the computer code associated with the element text box should recognize whether or not the he or she has used appropriate syntax for that weather element (following the typographical rules of the code). In keeping with the object-oriented paradigm (see §1.3.6), calling the element-object's "*verify*" method should perform the desired format QC. Furthermore, the system will ensure that any reports containing elements that are invalid (that is, they fail the QC process) cannot be finalized and submitted.

1.3.2 Content QC

Though format QC as described in the previous section is a vital part of ensuring "clean" data for use with any automated metrics program, it doesn't go far enough. The system architecture design as implemented based on the SRD provides the possibility of performing limited *content* QC as well as format QC.

Since the TAFs and observations are input via the same, standardized fields described above, forecast data entered could conceivably be compared with current observation information. If the entered data for the current forecast period is significantly different from the current observation, the forecaster could be notified of the discrepancy.

For instance, if the current observation reports the altimeter to be 29.94 inches of mercury, and the forecaster mistakenly typed 28.96 instead of 29.96 for the next hour, the interface could alert the forecaster of the error. Previously, even though it's just a typographical error, it would have been considered a forecast error—the *format* was right, but the meteorological *content* was wrong. A carefully configured and tuned interface such as the one proposed could catch this kind mistake.

Subsequent lines in the forecast are also "content QC'd" against the previous line. For instance, in a situation analogous to the altimeter example above, if at the 12 hour point in a TAF the altimeter is forecast to be 29.94 inches, but at the 13-hour point, it's forecast to be 28.96 inches, the forecaster could be flagged for a possible content error.[ii]

1.3.3 Forecast Advice

As can be seen in the previous section, a "content QC" requires the system to make domain-knowledge-based inferences in order to perform the desired

[ii] Of course, specific meteorological parameters must be designed into the system and be custom configurable to help guard against "over-warning." For instance, in some circumstances, such as during a severe thunderstorm, an inch drop in the altimeter setting is not unheard of.

function. For instance, in the altimeter example cited above, the usefulness of the QC function is directly related to the system's level of understanding of meteorology.

For instance, the system could "know" that altimeter settings (station pressure) typically range from 28.00 to 32.00. Anything outside this very broad range would be considered an error, and reported as such. If the system also knew the station elevation, a finer range of valid altimeter settings could be determined, and more accurate QC could be accomplished. If the system had access to more information—the altimeter setting from the previous observation, typical climatological values for altimeter settings, weather model predictions of altimeter setting, etc., the QC the system could provide would be even more accurate.

As can be seen, the more meteorological knowledge the system has, the more accurate and refined the QC function can be. In fact, as the system makes more and more meteorologically based inferences based on the forecaster's forecast, the system's role advances beyond that of simple QC to that of an expert knowledge system that can advise the forecaster during forecast production. (See §2.2.5 for a more in-depth discussion of such a system and its application to forecast product generation).

1.3.4 Real-time Forecast Verification

A natural by-product of this system is the ability to automatically verify the TAFs on a real-time basis. That is, to compare current TAFs to station observations (Obs) and alert the forecaster about discrepancies, not only before initial TAF transmission (as with the content QC), but throughout the forecast period. As observations are entered into the local database, the data could be retrieved and compared to the currently valid line of the TAF, and notification sent to the forecaster if discrepancies were found.

In addition, simple trend analyses could be automated for certain user-defined parameters in order to alert the forecaster of current conditions that might not match the forecast. For example, if the TAF indicated ceilings increasing from 4,500 feet initially to 6,000 feet over a certain time period, but the actual, observed ceiling dropped to 3,500 feet, the forecaster would be alerted to the trend. Though such a change would not be "out of category"—that is, by rule the forecaster would not have to amend the TAF unless the ceiling dipped below 3000 feet—the forecaster would be made aware of the trend, and perhaps be able to issue an amendment before the fact rather than after. (See §2.4.1.1 for an explanation of the category system of forecast verification).

This trend analysis capability would be especially beneficial as forecasting responsibility shifts to the regional OWSs, since a handful of forecasters will now be responsible for keeping tabs on the performance of many forecast reports. Under the old system, the duty forecaster was able to work closely with the observer to get a first-hand feel for trends in the weather. Under the reengineered system of regional hubs, forecasters will have to rely almost exclusively on their own data analysis.

1.3.5 The Modified TAF Generation Process

As mentioned above, under the current architecture, much of the capability described above is not practicable. With the advent of reengineering, however, the architecture will be in place to support such a capability.

With the implementation of an interface as described above and detailed in Chapter 3, it is useful to revisit the envisioned TAF generation process. Figure 1-2 provides a high-level depiction of this revised process.

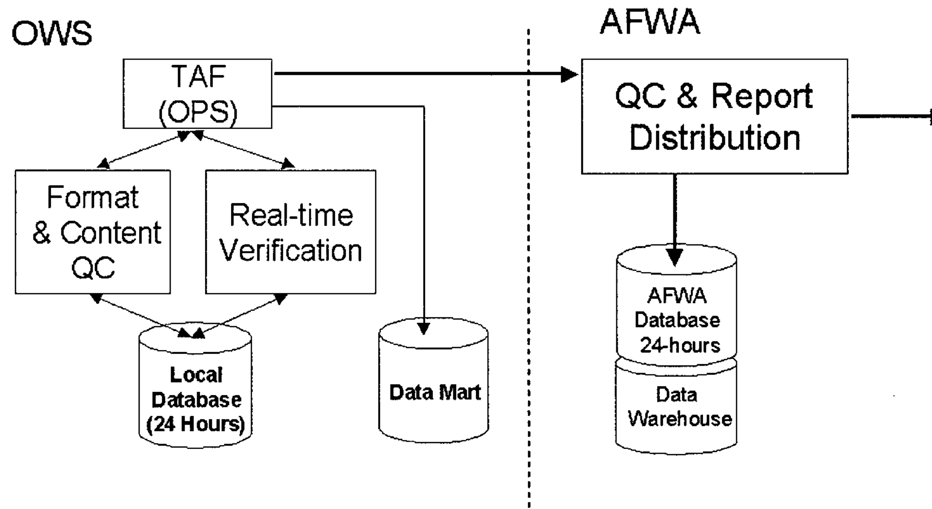


Figure 1-2. Proposed process for TAF generation

In the proposed process, the local database (now at the OWS) contains not only forecast information but observation data as well. When a TAF is generated on the "OWS Production System" (OPS) interface (the OWS weather product production computer—see §2.4.3.2), it is QC'd for format *and* content, as described above. Additionally, the submitted forecast information may be stored locally in a data mart and/or at the regional center in a data warehouse for direct use in an automated metrics program (see [27] for information on data warehousing and automated metrics).

Note that the AFWA processor still performs an additional QC, however. This is because though the QC logic employed at the AFWA *back* end should be identical to the *front* end QC performed by the interface, the network data transfer can generate format errors that need to be identified and corrected.

Finally, throughout the forecast period, the interface monitors new observations and warns the forecaster of trends not conveyed in the current forecast. With OWS forecasters now responsible for producing forecasts for not just one but many locations, this real-time automated verification component is vital.

1.3.6 Design Considerations

The SRD establishes specific requirements for a component-based, object-oriented system [15]. As much as possible, this research adhered to the standards addressed in this document to ensure smooth integration. Additionally, the following were considered in the prototype design:

- Modularity. The incorporation of sound object-oriented software engineering principles and modularization to facilitate software reuse and to allow for easier validation, verification, and, if necessary, modification of the code in the future.
- Extensibility. Though its current focus is TAFs and observations, the design should be adaptable to other products (point warnings, weather warnings and advisories, regional forecasts, etc.) as well.
- Efficiency. The design should be able to handle typical and peak workloads in a timely manner.
- Flexibility. The design should be customizable to enable AFWA, as well as individual OWSs and WFs, to adapt any application to their specific needs while complying with AFW requirements.

In addition, in keeping with the goal of extensibility and modularity, the design will allow for logical changes and growth of the system given the framework of the reengineering effort. Possible areas for growth are:

- Changes in amendment criteria
- Changes in METAR format
- Automated verification/metric evaluation of additional products (refueling forecasts, drop zone/landing zone forecasts, launch forecasts, point warnings, etc.)

1.4 Outline of this Document

The rest of this document will provide a step-by-step discussion of the research. Chapter 2 gives background information regarding a variety of aspects of the user interface. Included is a discussion of the current and past work regarding human-computer interaction, and, specifically, intelligent user interfaces, as they apply to this thesis. Also included is a more detailed discussion of the OWS systems architecture, as well as an outline of the pertinent specifications dictated for the OPS computer, and how these will impact design and implementation decisions. Finally, some background of applicable weather domain-specific issues, including previous TAF Verification implementations and the METAR code are presented.

Chapter 3 details the object-oriented design specification of a critiquing system that employs the JESS inference engine. Included are a discussion of the design concept and focus, a high-level overview of system components, and a detailed description of how system objects interact.

Chapter 4 entails a discussion of a design implementation to show proof of concept. The discussion includes an analysis of the interface from a software engineering and weather perspective based on specific rules developed.

Finally, Chapter 5 presents conclusions drawn from this research and suggests areas for future study.

2 Summary of Current Knowledge

2.1 Overview

The development of a user interface such as the one introduced in the previous chapter requires the integration and application of information and techniques from a variety of domains. This section provides further background and discusses current and past work in these areas as it applies to this project. Specifically, it addresses current work in the area of Computer-Human Interaction, and provides background on weather domain-specific topics important to this research, such as past attempts at centralized, automated TAF verification, the METAR standard weather code, and the new weather system architecture.

2.2 Computer-Human Interaction

Computer-Human Interaction (CHI) is the field of study that concerns itself with the way humans communicate with computers. In recent years, so much attention has been given to the development of user interfaces that the Association of Computing Machinery (ACM) has established the Special Interest Group for Computer-Human Interaction (SIGCHI), dedicated to their study and development.

Much of the research focus of CHI is geared toward constructing interfaces that can be manipulated with maximum efficiency and ease. The following extract from the home page of the ACM's SIGCHI sums up the direction of CHI work:

> The scope of SIGCHI consists of the study of the human-computer interaction process and includes research and development efforts leading to the design and evaluation of user interfaces. The focus of SIGCHI is on how people communicate and interact with computer systems [2].

Because of its broad scope, people from a variety of disciplines conduct CHI research. Among these disciplines are computer science, psychology, cognitive science, human factors, industrial design, graphic design, anthropology, sociology, management science, software engineering, and others [2]. Though much of CHI activity is not applicable to the present research, one area of CHI study that does apply is that of Intelligent User Interfaces (IUI).

2.2.1 Intelligent User Interfaces

Over the past decade, researchers in CHI have explored the role of agents as they apply to user interfaces. Though there is some disagreement in the literature of the Artificial Intelligence community and in the area of CHI concerning the definition of intelligent user interface agents [29], there is a broad spectrum of research concerning the role of agents to assist the user.

Some of the research involves assistance with the user interface itself, such as Microsoft's Office Assistant [30]. With every version upgrade to any off-the-shelf application comes the addition of sometimes hundreds of new features. As these applications offer more features, user interfaces necessarily become more and more complex. At some point, users will simply be overwhelmed by the complexity of the interface and be unable to take advantage of the new features available [13]. Microsoft's Office Assistant is an attempt to aid the user in learning a new, more complex interface by monitoring the user's input and suggesting alternative methods (tips) to accomplish the task.

Other research involves domain-specific task assistance through agents. In this case, the agents contain domain-specific "knowledge," and can provide suggestions, corrections, task completion assistance, or other guidance to aid the user in completing his or her task. An example of this type of "critiquing" agent is *Janus*, an application used in the design of residential kitchens [20]. In *Janus*, the critiquing agents have domain-specific knowledge—that is, they understand

the design rules associated with building a kitchen (e.g., required window area, space between appliances). As the user constructs a kitchen using the *Janus* application, the agent will point out design problems and offer solutions that follow correct design guidelines and while remaining consistent with the user's design.

2.2.2 Interface Agents and Autonomous Agents

Henry Lieberman[i], a Research Scientist at the Massachusetts Institute of Technology's Media Laboratory, cites the need for agents that are both "interface agents" *and* "autonomous agents," though he contends that in the past, these have been considered mutually exclusive. He defines an "interface agent" as "software that actively assists a user in operating an interactive interface," while an "autonomous agent" is "software that takes action without user intervention and operates concurrently, either while the user is idle or taking other actions." An "autonomous interface agent" is an agent that operates in the interface, and concurrently with the user (as opposed to having a sequential conversation with the user—that is, invoked solely by user actions) [29].

According to Lieberman, the important characteristic of an "autonomous interface agent" is that the agent "may need to interact with the interface while the user is also interacting with the interface [29]."

2.2.3 What Makes an Interface Intelligent?

Though software can interact with a user interface, it is not necessarily "intelligent." For instance, Microsoft's Auto-correct feature can correct misspelled words as a user types, but this is a simple look-up table procedure. Because of this, the definition of "intelligent interface" in the literature is subject to as much disagreement as the definition of "intelligent agent."

[i] Henry Lieberman has been with the MIT Media Laboratory since 1987. From 1972-1987 he was a researcher at the MIT Artificial Intelligence Laboratory, originally involved in the development of Logo. He holds a doctoral-equivalent degree from the University of Paris VI [28].

Lieberman's opinion is that from a user's perspective, an agent is an assistant or helper rather than a tool, which displays "some (but perhaps not all) of the characteristics that we associate with human intelligence: learning, inference, adaptability, independence, creativity, etc." He goes on to say that a user will "delegate" a task to an agent, rather than "command" the agent to perform the task [29].

Not only is there disagreement in the literature concerning the definition of "intelligent interfaces," but there is also debate regarding how much artificial intelligence an interface really *should* contain. During the ACM International Conference on Intelligent User Interfaces in 1997, several leading researchers, including Lieberman, gave their opinions on the issue [3]. The consensus was that the amount of intelligence required in an interface is solely dependent on the content of the interface, rather than, as Larry Birnbaum[ii] put it, the application of whatever "magic bullet" algorithm is available. In other words, according to Birnbaum, "the focus [should be] on the task being performed jointly by [the] user and [the] system" and not on the algorithms used in building intelligent interfaces. Lieberman's perspective is that the worth of interface's intelligence is entirely dependent upon how the user *perceives* it. It might be helpful, but it might also be annoying. His advice is to apply it conservatively and thoughtfully, keeping the user in mind at all times.

2.2.4 Wizards and Guides

Other agents that reside in the interface and aid the user in task completion are "wizards" and "guides." Most of us have had experience with wizards in, for instance, installing software. A wizard is an agent that assists the user through a linear, step-by-step process to complete a task. It is not autonomous, in that it performs no action without user stimulus. Conversely,

[ii] Larry Birnbaum is the Associate Professor of Computer Science and Department Chair for Northwestern University's Institute for the Learning Sciences. He has a Ph.D. from Yale University, 1986.

the user performs no action until the wizard asks. And though users may perceive wizards to be "intelligent," they usually contain no artificial intelligence. They simply present the user with choices along an efficient path to task completion [11].

Guides, on the other hand, are intelligent agents that monitor a user's interaction with an interface, infer the goal of the task, and physically annotate the interface to direct the user to the best next step in the process. A comparative study of wizards and guides indicated that they are best suited for use with infrequent, difficult and/or important tasks [11].

2.2.5 Critics

Critics, alluded to earlier in §2.2.1, are rule- or procedural-based agents that contain specialized domain knowledge of specific aspects of a task or product. Critics are useful in *cooperative problem-solving*, the process by which human and computer cooperate to solve a specific problem or complete a particular task. Each entity brings its own "expertise" to bear on the problem at hand: the human uses common sense, defines the goal, and sub-divides problems, while the computer provides external memory, ensures consistency, hides irrelevant information, and summarizes or visualizes information. As the user performs actions or inputs information along a path to a desired goal, the computer uses critics to make inferences about the user's input, and make "educated" suggestions for improvements, highlight inconsistencies, or identify errors [20].

Researchers at the University of Colorado, Boulder, have sought to characterize critics and determine their role in cooperative problem-solving systems. Their view of the critic's role is as follows:

> The core task of critics is to recognize and communicate debatable issues concerning a product.[iii] Critics point out errors and suboptimal conditions that might otherwise remain undetected. Many critics also advise users on how to improve the product and explain their reasoning. Critics thus help users avoid problems and learn different views and opinions [20].

In some instances, as discussed in [20], each and every rule or procedure can be referred to as a critic. In this document, however, a critic embodies the concept of a rule- or procedure-based interface agent that satisfies the definition given above.

2.2.5.1 Domains Suitable for Critics

The research at Boulder concluded that critics are "particularly well suited for design tasks in complex problem domains." Problems that can be precisely specified and for which optimal solutions can be found algorithmically are less well suited for the application of critics. Problems which cannot be precisely defined or for which optimal solutions cannot readily be found, on the other hand, are typically well suited for critics [20].

The researchers identify two types of domains where critics would apply. The first type, partially defined domains, are domains which are not sufficiently understood, so that it would be impossible to create a complete set of principles that adequately captures their domain knowledge (and therefore unreasonable to allow an intelligent agent to work completely autonomously on behalf of the user). Examples cited were domains such as computer network design and user interface design [20].

The second type of domain identified is one in which the domain is "so vast that a tremendous effort is needed to acquire all relevant knowledge." High-

[iii] The authors define a "product" to be anything from a computer program, to a kitchen design (such as in the *Janus* application referred to in §2.2.1) to a medical treatment plan. The "product" is therefore simply the task or problem the user wishes to solve.

functionality computer systems are cited as an example of this latter domain type [20].

2.2.5.2 Characteristics of Critics

The distinction between domains suitable for the application of critics and those more suitable to the application of intelligent agents in expert knowledge systems is precisely what differentiates a critic from an intelligent agent. Expert knowledge systems leave the human out of the decision-making process, leaving all of the "intelligent" decisions to the computer [20]. Given the domains described above, this would be relatively impossible since the agent cannot possibly be given all of the relevant domain information.

On the other hand, critics are very similar to Lieberman's definition of an "autonomous user interface" in [29] (see §2.2.2). They operate within the user interface rather than external to it, and they are autonomous—performing tasks on behalf of the user, concurrent with the user's actions. The amount of "intelligence" applied to critics should be, as in the expert opinions described in §2.2.3 above, content (or problem) specific.

2.2.5.3 The Critiquing Process

Critics are best applied with a collaborative or cooperative problem-solving approach. Critiquing is the application of a "reasoned opinion" about a problem or action. Figure 2-1 illustrates the iterative critiquing process. The human user brings domain expertise and established goals to the table, and begins constructing a solution. The computer assistant/critic applies its knowledge of the domain and the user model to the user's solution, and may or may not offer a critique. Depending on the design of the system, the user may choose to disregard the critic's suggestion or incorporate the suggestion into the solution.

Something not obvious from figure 2-1 is that the "Proposed Solution" need not be complete. The critiquing process applies at all stages of solution construction. In the case of the *Janus* kitchen design critic for instance, a complete design need not be constructed before the critic would offer suggestions and provide design critiques.

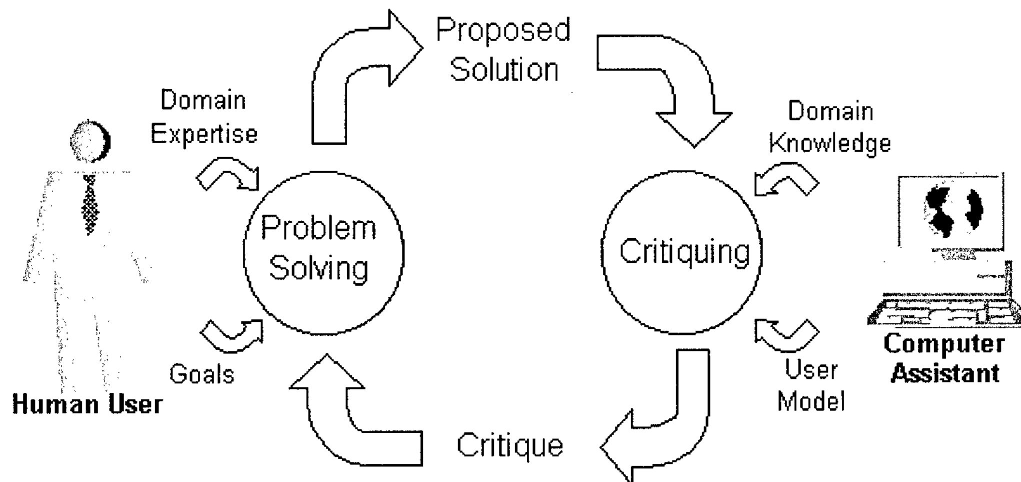


Figure 2-1: The Critiquing Cycle [20]

On the other hand, the critic itself must be cognizant of how far along the user is in solving the problem. Again, referring to the *Janus* critic, if a user placed a small window on one wall of the kitchen, the critic shouldn't automatically inform the user about the square-footage of window area required. In this case, the user is still in mid-process, and may include more windows in the design. Should the critic offer such premature advice, it could be seen as intrusive [20].

To be effective, the role of critics for a particular application must be well conceived. "Intervention strategies" must be developed to ensure that critiquing is not intrusive, and that critiques don't disrupt the user's cognitive process and cause short-term memory loss. Additionally, critics should be able to adapt to the user. For instance, if a user disagrees with a suggestion and either explicitly or implicitly disregards it, the critic should not repeatedly make the same suggestion [20].

2.2.5.4 Critics as Teaching Tools

The Boulder researchers found that the use of critics not only aided users in the completion of their tasks, but also helped them *learn* more about the task at hand. "By showing that the artifact under construction has shortcomings, critics cause users to pause for a moment, to reflect on the situation and to apply new knowledge to the problem as well as to explore alternative designs." In this way, they serve as "skill-enhancing" tools [20].

As the researchers put it, the goal of using critics is to design a "system for experts," rather than an expert system.

2.2.6 Critic Application to the Weather Problem

Upon analysis of the literature involving CHI, and specifically intelligent user interfaces, it becomes evident that agents applied in an expert knowledge system, and most logically a critic-type system, can be applied to the problem introduced in Chapter 1. The weather domain is an extremely large and complex one, involving a tremendous number of variables interacting with each other in countless ways. An optimal solution to the forecaster's problem (i.e., to produce an accurate forecast) is one that cannot be attained with a simple application of algorithms. In fact, the field of Numerical Weather Prediction (developing accurate models that predict the value of specific weather parameters) is as close as we have been able to come to such a process, thus far. The algorithms used to generate these models, however, consider only a fraction of the continuous spectrum of meteorological "data" that describe the weather. For this reason, these models are used only as tools for the forecaster to consider in developing a forecast.

The forecaster's task is to sift through the piles of raw surface observations, upper air observations, model charts, local area weather charts (LAWCs), and other data; and through application of domain knowledge and

reasoning, finally compose (using appropriate syntax, of course) an accurate forecast of what the weather will be in the future.

As in the *Janus* kitchen design program, there is an infinite number of possible solutions. The forecaster's goal, like the *Janus* goal, is implicit. That is, the goal is predefined and built into the system. In *Janus*, the goal is to design a kitchen so that it meets standards and is functional and appealing. The forecaster's goal is to produce a forecast that approaches the optimal (the true behavior of the weather).

In *Janus*, the design must adhere to certain predefined guidelines. The role of the *Janus* system, therefore, is to ensure that the user meets the guideline standards, and to offer any suggestions that the system "thinks" (by virtue of its domain knowledge) would enhance the functionality or appeal of the kitchen.

In weather forecasting, the forecaster must meet the specific syntax requirements of the METAR code (see §2.4.2 below). The role of a weather critiquing system, therefore, might be to ensure correct syntax and, based on knowledge of the domain and current data, to suggest alternatives or enhancements to the forecast.

As mentioned in §1.3.2, content QC (and not just format QC) is important to producing an accurate forecast. The application of a critiquing system such as described here would do precisely that. The depth of the assistance provided by the system (format, comparisons to current observations, or more complex reasoning) would therefore depend on the complexity of the critiquing system and the amount of data at its disposal.

2.3 Rule-Based Expert Systems

The critic system introduced above is a knowledge-based expert system that can make "inferences" about a specific domain. It does this through the application of domain-specific rules. When the system is initialized, it must be

given a set of rules or conditions that apply in certain circumstances. This section describes the sorts of rules upon which expert systems can act, and introduces expert system shell software that is incorporated into the application design in Chapter 3.

2.3.1 Rules and Facts

An expert system's knowledge is its set of rules. Rules are simply statements of conditional logic that relate different circumstances. For instance, an example of a rule might be:

"If I am speeding and if a policeman sees me, I get a ticket."

This is not to say that I *am* speeding, nor does it suggest that a policeman sees me. It just specifies the resulting situation given the stated circumstances. Rules can be fairly simple (as in the above example) or very complex, depending on the problem domain. Rules can also be (and often are) dependent upon one anther.

For instance, to continue the example above, a second rule might be established such that:

"If I get a ticket, I pay a fine."

In this case, if the conditions for the first rule are satisfied, it follows that the conditions for the second rule will be satisfied as well.

Of course, the rules described in the previous section are useless without knowing the circumstances surrounding them. A rule-based expert system employs (or *fires*) rules at specific times based on its knowledge of related "facts."

In the examples given above, a fact might be "I am speeding." Given only this knowledge, however, the system would not conclude that "I will get a ticket." Only when the fact "A policeman sees me" is also ascertained are all of the conditions necessary to satisfy the rule established, and will the system conclude that "I get a ticket." It *infers* a new fact—namely that "I get a ticket"—from *firing* a rule that was satisfied given the circumstances. Note that this new

fact is sufficient information to fire the second rule and conclude that "I pay a fine."

2.3.2 Knowledge-Base and Inference Engine

A mechanism used to apply rules and facts to a specific problem domain is called an *inference engine*. The *knowledge-base* is the set of rules—that is, the understanding the system has about its particular problem domain. These two components make up an *Expert System* [23]. Though the system can be provided with a set of "base-line" facts, it typically gets its circumstantial information—its facts—from the user. The inference engine uses the facts provided to satisfy any applicable rules, and returns domain-specific "expertise" to the user (see figure 3-1).

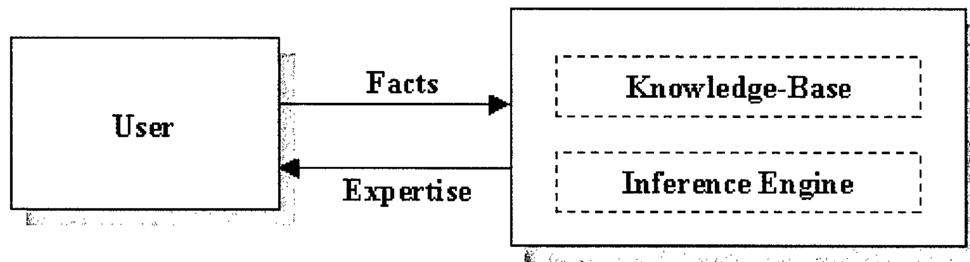


Figure 2-2. Basic Concept of an Expert System Function [23]

The power of an expert system, of course, is derived from the amount of knowledge the system has. The more rules and facts the system is aware of, the more opportunity the system has to provide the user with useful information. A system may have hundreds or even thousands of rules [23]. It should be clearly understood, however, that the system is only as good as the human that established its rules. For an expert system to perform well, it must be "taught" and fine-tuned by a human with domain expertise.

2.3.3 The Java Expert System Shell

In the design for the user interface presented in Chapter 3, the expert system employed is built around the *Java Expert System Shell* (JESS). JESS is a Java-based shell that supports the development of rule-based expert systems [19]. It provides a mechanism to establish a set of domain-specific rules that will fire based on provided facts.

JESS is strongly linked with the Java language (Java can be programmed from within JESS, and JESS rules and facts can be established from Java). It is very closely related to its predecessor, the C Language Integrated Production System (CLIPS), a C-based expert system shell [23, 32].

CLIPS was initially developed at Johnson Space Center for NASA in 1984. Since then, it has been become public domain, and is maintained by its original authors (no longer at NASA) [32]. JESS, written in 1995 by Ernest J. Friedman-Hill at the Sandia National Laboratories in Livermore, California, was initially a clone of CLIPS, but has grown into a Java-language specific tool designed for integration with Java programs and applets [19].

The core syntax of CLIPS and JESS is very similar—many expert systems written for CLIPS will work with JESS—but since JESS has been expanded to take advantage of features of the Java language, the reverse is not necessarily true.

2.3.3.1 Expert System Shells

An expert system shell is designed to be precisely that: a program shell that implements the functionality of an expert knowledge system, but without being domain-specific. Its primary function, therefore, is to maintain a set of "if-then" rules such as those described above in §2.3.1, and a set of facts to which the rules can be applied.

When a fact or set of facts matches the pattern requirements for a particular rule, the rule fires, and some action is taken. The action taken may be

anything from printing a message to the standard output, to asserting a new fact, to calling a method of an external object.

The most important aspect of an expert system is the efficiency with which it processes its rules and facts. In a complex problem, there could be literally thousands of rules, each involving patterns of one or more facts. Every time any fact is asserted, deleted, or changed, the set of all of its rules must be checked to see if they are affected by the change. Depending on implementation, this operation can be extremely computationally expensive, and inadequate for large-scale applications. In fact, if this "brute force" method of testing each rule is used, the computational complexity is on the order of $O(RF^P)$ where R is the number of rules, F is the number of facts, and P is the average number of patterns per rule [19].

2.3.3.2 The Rete Algorithm

In 1982, Dr. Charles L. Forgy published a paper that addressed the "Many Pattern/Many Object Pattern Match Problem" [21]. In it, Forgy describes the Rete (pronounced "ree-tee[iv]") Match Algorithm, which has been shown to reduce the computational complexity of the pattern match problem to as low as $O(RFP)$, or linear based on the number of facts (rules and patterns typically remain constant) [19, 21].

The key to the algorithm's success is recognition that statistically speaking, from one rule-checking cycle to the next, the percentage of facts that actually change is small. This means that most of the time, the rule test results on each cycle will be identical to the test on the previous cycle. The inefficiency mentioned above, therefore, is due largely to checking and re-checking rules whose fact-pattern matches have not changed.

[iv] Rete is a Latin word meaning "network," and in Latin is pronounced Reh'-tay (accented on the first syllable). The Webster's Encyclopedic Unabridged Dictionary of the English Language gives the definition as "a network, as of fibers, nerves, or blood vessels," and gives the pronunciation as rē'tē.

The Rete Match Algorithm addresses this problem by remembering past test results with each cycle. It is implemented with a tree-sorted node structure in which each pattern is associated with a list of the facts that it matches. Every time a fact is added to or removed from the list of facts maintained by the system, it is inserted into or deleted from the tree at the appropriate place.

Figure 2-3 illustrates the process. Each node in the tree associates either one or two facts. The single-input nodes (at the top of the figure) represent facts on a fact list. As a particular fact is asserted, it advances down one or more branches in the tree. For instance, X and Y are facts that match two patterns, so they advance down both branches.

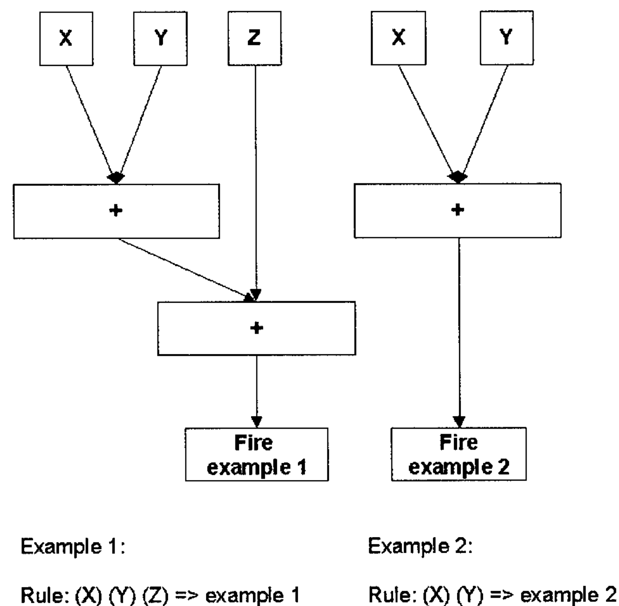


Figure 2-3: Rete Match Algorithm pattern tree [19]

The nodes labeled "+" are two-input nodes. They fire (or allow progression down the tree) when both of their associated facts have been asserted. In the figure, when X and Y are asserted, the "example 2" rule fires. The "example 1" rule will not fire, however, until Z is asserted.

Optimizations can be made to the above tree structure. Such is depicted in Figure 2-4, since X and Y are associated with patterns for both rules. The resulting network of nodes not only requires less memory, but also takes less time to traverse.

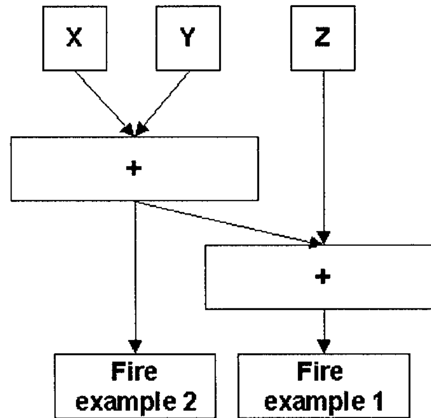


Figure 2-4: Optimized Rete Match Algorithm tree structure [19]

The Rete algorithm has been implemented in several generations of expert system shells, including OPS5, ART and CLIPS [24].

2.3.3.3 Expert System Shell Languages

In order to take advantage of the Rete algorithm, rules and facts must be properly formatted such that they may be correctly processed into the tree network. As mentioned previously, CLIPS and JESS use a very similar syntax, and the basic premise behind the two languages (and all expert system shell languages) is the same,

As an example, if we wanted to create a rule that printed the make and year of all Dodge automobiles known to the system, we could do so as follows:

```
(defrule dodge-make-and-year
  "Prints the make and year of all Dodges"
  (automobile (make ?m) (model ?mdl) (year ?y))
  (test (eq ?m Dodge))
  =>
  (printout t "Found a Dodge… Model: " ?mdl ", Year: " ?y crlf)
```

The rule effectively examines the make, model and year of every automobile in the system. If the make is a Dodge, the rule fires and the text is printed. The rule will only fire when an automobile is found in which the make, model, and year are all defined, and the model matches "Dodge." Note that if the "test" statement were not included in this rule, all cars with the three stated fields defined would be printed.

Similarly, facts are asserted as follows:

```
(assert (automobile (make Dodge) (model Durango) (year 1998)))
(assert (automobile (make Dodge) (model Ram)))
(assert (automobile (make Chevrolet) (model Suburban) (year 1996)))
```

As each fact is asserted, the "dodge-make-and-year" rule fires accordingly. The first facts asserted will fire the rule, and the last two will not. The rule will not fire for the second rule, because the year is not defined. It will not fire for the third rule because the automobile is not a Dodge.

As well as being able to define rules and facts, the JESS and CLIPS languages contain a multitude of commands designed to make the language more flexible, and to allow interaction between programs in native Java or C, respectively. For more information regarding JESS and CLIPS language syntax, see [19, 32].

2.3.3.4 JavaBeans and JESS

As mentioned in the previous sections, JESS has evolved into more than just a Java implementation of CLIPS. It has been designed to incorporate Java-specific capabilities. Among them is the ability to interact with JavaBeans. The design specification described in Chapter 3 takes advantage of JESS's ability to interact with JavaBeans to efficiently act upon weather data from a variety of sources. For this reason, it is useful to have an understanding of how JavaBeans work.

JavaBeans is a network-aware component architecture for Java. On first inspection, a JavaBean looks like any other Java class. The difference is that through specific method and attribute syntax, JavaBeans provide public interfaces to the services provided within each bean. Typically JavaBeans have been used to construct graphical user interface (GUI) components. Unlike other Java classes, JavaBeans contain properties that can be manipulated at runtime to change the behavior or appearance of a particular aspect of the component. In this regard, a JavaBean is a "self-describing" object that provides services for other components to use. Furthermore, through Java's implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), JavaBeans can be registered for use across networks and between heterogeneous platforms [25].

One feature of JavaBeans that is particularly useful is event notification. When one of the properties associated with a JavaBean (either one of the inherent properties of a Java library component, such as the background color of a button, or one of the attributes of the class itself) changes, a "property-change event" fires. Other JavaBean components can listen for these property changes, and take action based upon them.

Through specific definition statements, JESS allows the declaration of entire JavaBeans as facts. For instance, if a JavaBean class called "Automobile" were constructed with attributes of "make," "model," and "year," instances of the Automobile class properly registered as JavaBean object instances in JESS will be asserted as facts. For each of these facts, JESS employs a property change listener that fires based on changes to the attributes. This means that any changes to the value of any of the attributes in the JavaBean will effect a change in the value of the corresponding JESS fact.

2.4 Weather Domain Background

The previous section discussed current research into intelligent user interfaces. The remainder of this chapter will provide the reader with useful weather domain-specific information relevant to this project.

2.4.1 The TAF Verification Program

TAF Verification (TAFVER) was a program implemented by Air Force Weather, formerly Air Weather Service (AWS), as a first attempt to automatically monitor its own performance. The basic implementation of TAFVER was the establishment of a database in which specific forecast and observation data would be collected and compared. The forecast and observation data, continuously reported by weather stations worldwide, was decoded, and specific fields were stored in the TAFVER database. At regular intervals (monthly, quarterly, etc.), batch computer jobs were performed to compare the forecast weather parameters to the actual observed weather. The output from these jobs was statistical data indicating how accurate the forecasts were. The hope was that the statistics generated would divulge trends in inaccurate forecasts that could potentially be corrected through forecaster training.

The first generation of TAFVER was developed from 1972 to 1975. Several later versions of TAFVER were employed or prototyped, but in 1998, funding was cut and the program was scrapped.

The following sections provide an overview of the three versions of TAFVER, as well as some insight as to why the program was cancelled. Though this research does play only a supporting role in the establishment of a viable automated metrics system, in the development of an interface that performs pre-submission, real-time verification and validation of TAFs, it's useful to examine previous efforts to apply post-submission, automated TAF verification. In fact, the proposal to perform format and content QC before submitting a TAF is very

similar to applying a TAFVER program at the *front* end, with the goal of producing a better product, rather than collecting statistics on product quality.

2.4.1.1 TAFVER I

Under the initial program (simply referred to as TAFVER), ceiling and visibility were the only forecast parameters that were compared to observations [22, 26]. (Ceiling is defined as the lowest cloud base that covers more than 50% of the sky, while visibility is the greatest horizontal distance that can be observed around at least 180 degrees of the observer [8]). Additionally, forecasts were only verified based on hourly observations, and not "special observations"—observations taken when a significant change in some weather parameter occurred—for instance if the cloud ceiling dropped below a predetermined threshold [26]. As of 1986, the evolved TAFVER program had three main functions:

1. Retrieve stored and validated observations and TAFs and store selected elements in a separate database.
2. Send a receipt message back to the issuing weather stations.
3. Run verification algorithms and prepare end-of-month-reports for higher headquarters.

As TAFs and observations were ingested, they were decoded and stored in a database. Receipt messages were then sent back to the issuing weather stations on an hourly basis.[v]

Assuming each report was properly ingested without errors, algorithms were run against the data to compare the forecast ceiling and visibility to that of the actual observations. First the values were converted to a category system (as in the example shown in table 2-1).

[v] Actually, a list of all stations for which TAF or observation reports had been received was sent to all stations. It was the responsibility of the duty forecaster and observer to verify that reports had been properly ingested by the TAFVER database, and resend corrected reports if they had not.

Category	Value
Ceiling CAT A	< 200 ft
Ceiling CAT B	>= 200 feet, < 1000 ft
Ceiling CAT C	>= 1000 feet, < 3000 ft
Ceiling CAT D	>= 3000 ft
Visibility CAT A	< ½ mile
Visibility CAT B	>= ½ mile, < 2 miles
Visibility CAT C	>= 2 miles, < 3 miles
Visibility CAT D	>= 3 miles

Table 2-1: TAFVER ceiling and visibility categories [26]

Validation results were tallied based on the number of correct forecasts (where the forecast category was the same as the observed category). Table 2-2 illustrates the sample output of a TAFVER run. As an example, in the first line of the table, there was a total of 8 observations of Category A weather. Five times, Category A weather was correctly forecast. Twice Category B weather was forecast, and once Category D was forecast. From this information, statistics such as optimistic forecasts (where the forecast category was higher than the observed category), and pessimistic forecasts (where the forecast category was lower than the observed category) were derived. Similar comparisons were made between observed weather and "persistence"—that is as if the forecaster had based his entire 24-hour forecast on the previous observation[vi].

[vi] Forecasters are taught not to forecast the weather from scratch, but to start with persistence, and determine how the weather will change from what it currently is. Thus, a weather forecast is really just a prediction of how the weather will deviate from persistence—its current state. Because of this, it is sometimes useful to compare an actual forecast to a persistence forecast as a baseline. Since changes in the weather are typically incrementally small, persistence can do surprisingly well.

Terminal Forecast						
Observed		A	B	C	D	Sum
	A	5	2	0	1	8
	B	8	1	0	0	9
	C	25	4	2	1	32
	D	4	1	0	0	5
	Sum	42	8	2	2	54

Table 2-2: TAFVER Sample Output [26]

Conceptually, the initial TAFVER program fulfilled its intended goals. It was an automated process by which Air Force Weather could measure its ability to forecast specific weather parameters. Unfortunately, in practice, the program didn't live up to its expectations. Shortcomings of the TAFVER program are discussed below in §2.4.1.4.

2.4.1.2 TAFVER II

TAFVER II emerged in the early 90s. Under the new program, the verified forecast parameters increased to include wind speed, wind gusts and present weather (codes which indicate actual weather, such as rain, thunderstorms, etc.), altimeter setting (station pressure), and ceiling/visibility combinations. Additionally, TAFVER II incorporated special rather than just hourly observations into the verification algorithms, and provided limited verification of amended forecasts (forecasts re-issued to account for unforeseen weather changes) [10].

The process that TAFVER II followed was very similar to the initial TAFVER program. A separate TAFVER database was maintained to compare forecasts and observations, and statistics were generated based on the data comparisons. Because of the number of parameters tracked by the TAFVER II program, however, the output given was tailored specifically to a user's request.

For instance, as in Table 2-3, forecasts for a certain parameter, at a specific time, and for a particular category were computed separately. From this information, many complex statistical analyses and skill scoring measures were derived [10].

		Terminal Forecast	
Observed		Yes	No
	Yes	10	2
	No	2	27

Table 2-3: TAFVER II Sample Output (Ceiling < 200 feet at 3-hr point) [10]

One feature that TAFVER II offered was the ability to establish different category thresholds for different Air Force Major Commands (MAJCOMs). This was a first step toward tailoring forecast verification to the requirements of the forecast consumer, namely the flying units. For instance, because of special operations in the European theatre, U.S. Air Forces in Europe (USAFE) required detailed cloud information, and so forecast verification of ceiling height was divided into more stringent 9 categories instead of the 4 categories depicted in Tables 2-1 and 2-2 [10].

2.4.1.3 TAFVER IV

The next generation of the TAF Verification program was TAFVER IV[vii]. Though primarily just an interface enhancement to the TAFVER II program, TAFVER IV did offer much more flexibility as far as information selection and output was concerned. TAFVER IV was built using Oracle® design tools that allowed for analysis of specific segments of information. For instance, a user could initiate forecast verification runs over cloud forecasts for all bases that supported a specific airframe. Additionally, the maintenance of the TAFs,

[vii] Initially, the successor to TAFVER II was TAFVER III. As the requirements for TAFVER III grew, however, the program was re-christened TAFVER IV. The only references to TAFVER III on record are

observations, category thresholds, and station information could be easily manipulated through graphical interfaces placed on top of the database [12].

A prototype version of TAFVER IV was delivered in 1998. Shortly thereafter, however, AFW cut funding for TAFVER IV, and the program was cancelled.

2.4.1.4 Problems With TAFVER

Though the automated forecast verification programs developed and employed by AFW over the years were conceptually appealing, in practice, they had several problems that were never overcome.

Some of the problems were related to the complexity of the TAF code. For instance, short term, unforecast changes in the weather (lasting less than 20 minutes) don't require an amendment. If these changes occurred at the top of a verification hour, they would be counted as "unforecast," even though they were technically allowed. Similarly, free-text remarks such as those indicating a future wind shift (allowed in the METAR code) were not parsed, and therefore were not accounted for by TAFVER [10].

A more serious problem with the programs, however, was missing data. TAF and observation reports would not be used if they were rejected because of coding errors. In fact, because typographical errors were common in TAF and observation reports, the TAFVER II Users Manual [10] mentions specifically that TAFVER could not guarantee that a given TAF would be decoded successfully. Additionally, though the generated results would still be statistically valid if the missing reports were random[viii], the manual pointed out that "if a missing

---

monthly progress reports from the contractor to the Air Force Combat Climatology Center. TAFVER III was never implemented or prototyped.

[viii] No formal study of error trends has been documented, it seems, but an unsubstantiated report indicated that during inclement weather, forecasters are very busy, and tend to make more errors. If this were true, the statistical analysis might indeed be biased, since the occurrences of errors and therefore rejected (or missing) reports would not be random. Though plausible, no conclusive evidence to corroborate the report could be found.

observation is the one that verifies (or busts) a rare event forecast it will show up as a discrepancy [between the automated statistics and manually generated statistics]."

Because of the problems associated with the TAFVER implementations, a centrally driven forecast verification program has not been in place in Air Force Weather since 1996 (after TAFVER II was abandoned). Forecast verification was left to the individual major commands or to the WFs themselves.

2.4.2 The METAR Weather Code

METAR is an internationally accepted standard code used for the dissemination of weather forecast and observation reports. It is defined in the World Meteorological Organization's (WMO) Publication No. 306 on Meteorological Codes. The United States standard for the Aviation Routine Weather Report/Aviation Selected Special Weather Report (METAR/SPECI) code formats are defined in the Federal Meteorological Handbook (FMH)-1. This document covers all of the specifications enumerated in the WMO Publication No. 306, as well as the U.S. exceptions to the international standard [7]. The Air Force's implementation of the FMH-1 document is covered under the Air Force Manual (AFMAN) 15-111 [8].

For a complete definition of the code, refer to the FMH-1 or AFMAN 15-111. An on-line version of the FMH-1 document is available through the National Weather Service's home page [7]. For a sample forecast and observation, and a quick-reference breakdown of the codes, see Appendix A.

2.4.3 Weather System Integration

As mentioned in Chapter 1, the architecture of the reengineered Air Force Weather has been well thought out and documented in the System Requirements Document (SRD) [15]. The previous systems are based on old technology. The SRD calls for the establishment of a modernized and modular system, with the

intention of bringing AFW collection, production, and dissemination systems up to speed with current technology.

2.4.3.1 Problem Scope

The SRD states the rationale for the reengineering effort. The primary reason for the change has been the diminishing trust that operational commanders have had in Air Force weather forecasts. In some cases, pilots simply don't use AFW forecasts to make decisions, but rather "use the observations, then just go and look for [themselves]" [15].

Part of the reason for the decline in forecaster ability has been AFW's inability to keep pace with rapidly evolving technology. As stated in the SRD:

> [AFW's] technology base is vintage late 1970's and unable to leverage current commercial standards and technologies. The weather systems are a collection of stovepipes, perform redundant operations, and are labor intensive. The software is not modular and is difficult to change. The data formats are unique to AFW and do not comply with the latest available national and commercial standards. Data exchange and interoperability between systems is difficult. The end-to-end communications for both in-garrison and tactical operations is totally inadequate. Indigenous data feeds are part of local operations plans with no feedback to the strategic centers. The observational data and weather warning distribution is poor and often late [15].

The reengineering effort seeks to rectify these problems through a comprehensive overhaul of the entire production and communication system. At the heart of the system is the OWS Production System (OPS), alluded to in §1.3.5.

2.4.3.2 The Production Systems

The OPS is a collection of individual computers that make up the central system (hardware and software suite) upon which meteorologists in the OWSs will accomplish their mission. It is the central point for incoming weather

reports (surface observations, upper air observations, satellite data, radar data, forecasts from other OWSs, etc.). Additionally, it is the system forecasters will use to analyze weather data, produce forecasts, and disseminate these forecasts to their customers [15].

Also covered under the SRD is a requirement for the upgrade of the Weather Information Processing System (WIPS) currently employed at AFWA. The WIPS Upgrade (WIPS-U) will be the system used to perform the routine ingest of external data, analysis of data, forecasting of weather data, tailoring of products, and dissemination of the products and data required to fulfill AFWA's unique mission as the central weather support agency for all of AFW [17].

The basic requirements and implementation plan for the OPS and WIPS-U systems specified in the SRD are covered in more detail in the "Program Development Plan for the Weather Information Processing System – Upgrade" (WIPS-U PDP) [17] and the "Program Development Plan for the Operational Weather Squadron Production System II (OPS II) Program" (OPSII PDP) [16]. These comprehensive plans not only detail specific functional requirements of the OPS and WIPS-U, but establish architectural design requirements as well.

The major change from previous, pieced-together proprietary systems employed throughout AFW is the general requirement for each system to comply with Department of Defense standards. The important implication of this requirement is that applications will be developed to run on a common environment so that hardware and software will no longer be proprietary (as they were, for instance with AWDS, introduced in §1.2.1.1), and will fully integrate into systems already deployed throughout the Department of Defense. Additionally, the plans specify that product and message formats will adhere to national and international standards, addressing the most fundamental architectural problems identified in the SRD, discussed above in §2.4.3.1 [16, 17].

2.4.3.3 Architecture

The reengineered architecture as specified by the SRD will be entirely component based. Each (software) component (which may actually be made up of separate smaller components) will communicate with other components through clearly defined interfaces. Each component will provide a specific, advertised service to the rest of the system [15].

This important characteristic ensures compliance with the stated goal of system modularity, and makes system upgrades and changes easier and less expensive. Though the primary concern of this thesis is not the specific methods of communication and integration with the weather system architecture, it is important that any application design will adhere to these standards as specified in the SRD.

2.5 Summary

This review of relevant issues provided an explanation of the current work in the area of CHI (and specifically intelligent user interfaces), weather domain-specific information regarding previous verification implementations, and background on the METAR weather code, as well as an overview of the current and future architectural environment in which any interface application is to be deployed. The intent is to give the reader a clear understanding of the information used as a baseline for the development of the interface application.

The next chapter details the interface design specification. It explains the design methodology, providing a high-level description of system components, as well as a detailed description of object-interaction for a system that employs the JESS inference engine to achieve the goals set forth in §1.3.

3 Design Methodology

3.1 Overview

Chapter 1 discusses the need for a system that provides automated quality control of weather products before they are submitted into the Worldwide Weather Network. It presents the concept of a knowledge-based system that could not only perform format QC, but also use its weather domain knowledge to advise the forecaster of possible enhancements to the accuracy of the forecast.

In Chapter 2 information regarding the application of such a "critiquing" system is presented. It points out that critic systems have been applied to complex problems for which optimal solutions cannot easily be found. Critics are shown not only to have aided users in producing quality products, but even to have served as a teaching tool, highlighting domain-specific details for the user during the product generation process.

In this Chapter, an object-oriented design for a critic application to aid in Air Force Weather (AFW) product generation, using the Java Expert System Shell (JESS) inference engine introduced in §2.3.3, is presented. Of course, as for any application, there can be many design solutions. The aim of the design presented here is three-fold: Firstly, the design should be dynamic—that is, to the greatest extent possible, it should provide mechanisms to allow, at runtime, the modification of report templates, report properties, field properties, and rules. Secondly, though the focus of this research is the computer-assisted production and publication of TAFs and observations, any system developed should be flexible enough to be easily adapted for other surface and upper-air weather products. Finally, the system should be implemented in such a way that the data can be easily manipulated and transformed for use in an automated metrics program. See [27] for more information regarding data warehousing and its role in automated metrics.

3.2 User Interface Design

This section details the design methodology that satisfies the requirements for a system laid out in the previous chapters. Included are a brief description of design tools used, a summary of the overall design concept, and a detailed explanation of the application components and their purposes.

3.2.1 Design Tools and Language Paradigm

The design specification for this application was developed using Rational Rose®, an object-oriented Computer Aided Software Engineering (CASE) modeling tool which employs the Unified Modeling Language (UML) [31]. As mentioned previously, in keeping with the specifications outlined in the SRD [15], the design is completely object-oriented. Although the portions of the design that were implemented for this thesis were written in the Java language (see Chapter 4), the design is language-independent, and is applicable with any language that follows the object-oriented paradigm (such as C++).

It should be mentioned, however, that the design specification is written with the intent of employing JESS as the core system inference engine. As mentioned in §2.3.3.4, interaction between JESS and native Java code is facilitated by the use of JavaBeans. As such, changes to the JavaBean attributes are automatically registered in their corresponding JESS facts. If all or part of this design were to be implemented in a language other than Java, and a different expert system shell used, modifications to the specification to account for implementation-specific inter-object communication could be needed. Any alterations to the specification for this reason, however, would need only to be in the form of additional methods or attributes in the weather data objects themselves, or of an additional interface object for the purpose of establishing communication links with the expert system shell. No changes in the overall model structure should be required.

3.2.2 Design Concept and Focus

The focus of the overall system is, of course, the user interface—the on-screen interactive report form that the forecaster or observer will complete during weather product generation. However, in order to implement a critic system with a JESS inference engine, both user input data and external data must be made available to the system. Furthermore, data from these various sources must be linked, so comparisons can be made and feedback presented to the user. The design, therefore, will center on data object formatting and interaction with JESS to achieve the goals described in the overview above and in §1.3.

The data itself may be any information upon which the system can base meteorologically sound rules. For instance, if any TAF-specific critic rules require parameter values from a local surface observation and/or a local upper-air observation, those reports must be not only available to the system, but also be recognized as inference data for the specific report being generated, as well as put into a format from which the parameter information may be readily extracted.

Note that since the weather products being generated through this interface could be TAFs or other forecast reports, or METAR observations or other observation reports such as a Pilot Reports (PIREPs), all products generated by and ingested into this application will hereafter be referred to as "reports" or "weather reports." When important, a distinction will be made between forecast reports and observation reports. When necessary, information specific to either TAFs or METAR observations will be addressed.

3.2.3 Report Generation Process

The high-level conceptual process a forecaster uses to produce a forecast report with this application is illustrated in Figure 3-1. As depicted in the figure, the forecaster fills in a form that has been generated from a set of templates stored in a local database. The templates can be generated from scratch, or, more

likely, if a valid report of the same type for the same location exists in the active archive—in this case, a TAF valid for Offutt AFB (identifier KOFF), the old report will be superseded by the new when the new report is finalized.

When the initial template is invoked by the user (again, a specific template type for a specific location), a weather report (a set of *surface condition objects*) containing the weather information associated with the data entered by the user into the report template must also be created.

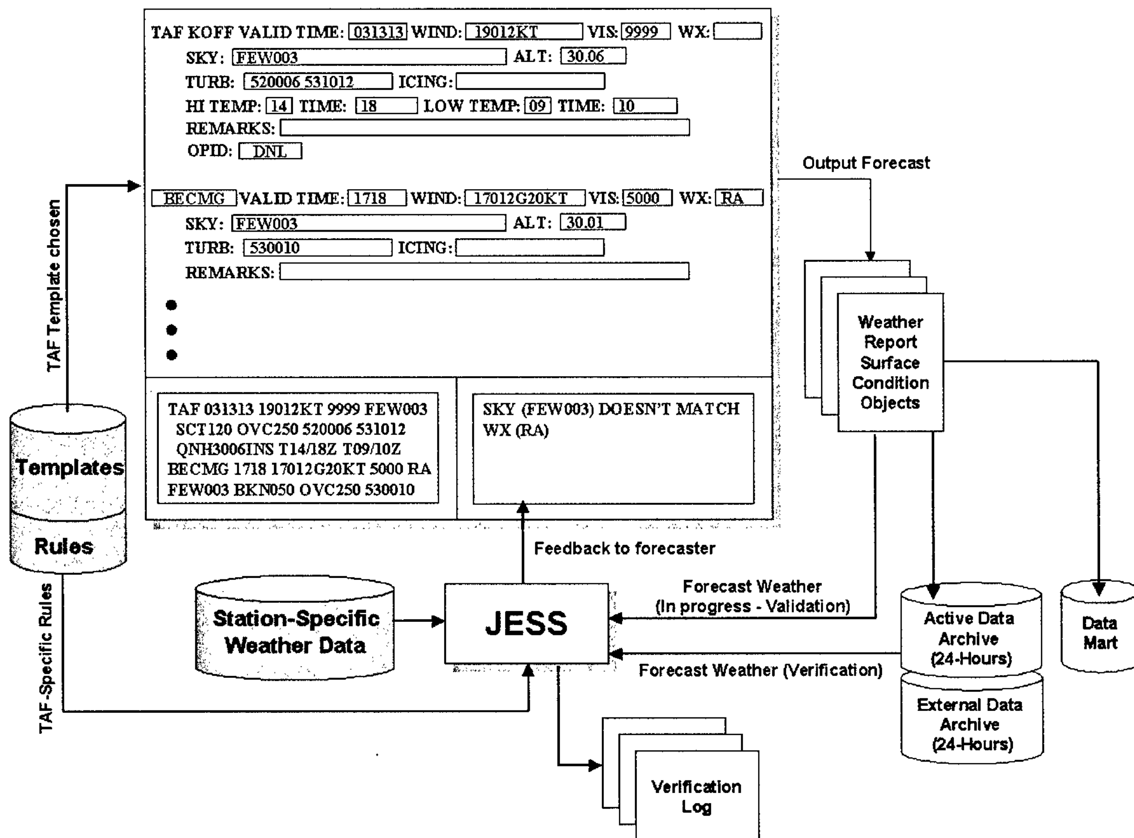


Figure 3-1: User Interface Design Concept

3.2.3.1 Report Templates

The concept of a *surface condition object* (from Figure 3-1) is new. This design is based on the premise that every *line* of a report describes the *condition* of the atmosphere for a given period of time (the valid time for the *line*). In the

case of the TAF in Figure 3-1, the first line describes the forecast surface weather starting at 13Z on the issue date[i] and ending at the start valid time (18Z) of the subsequent report *line*.

It is important to note the distinction between the *report template* and its associated *template lines*, and the *weather report* and its associated *surface conditions*. As will be more fully described later (see §3.2.4.3), templates (and template lines) are solely for the purpose of collecting user input, while the job of the weather report (and its associated weather conditions) is to compactly store the weather information extracted from the template (see Figure 3-2).

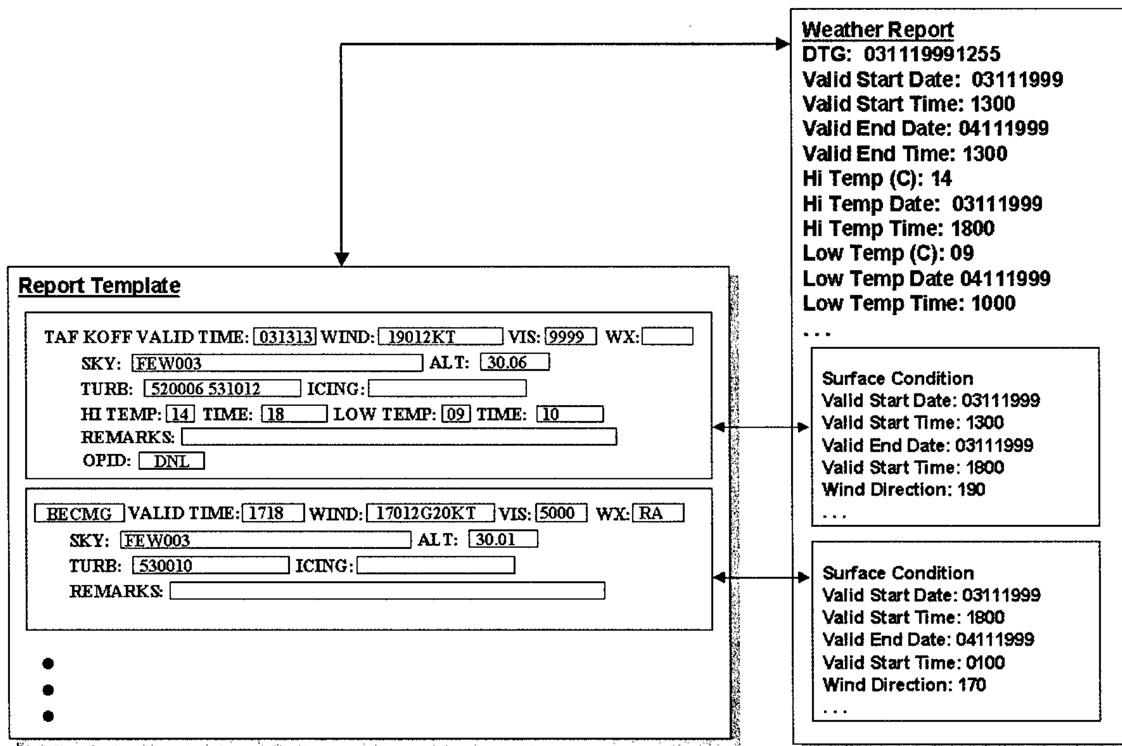


Figure 3-2: Report Template-Weather Report Relationship

As illustrated in the figure, there is a one-to-one correspondence between report templates and weather reports, as well as between template lines and weather conditions. The difference is that templates are made up of *report*

[i] See Appendix A for a sample TAF and METAR code definition

elements—each labeled entry block in the template depicted in the figure. As will be clarified later, from a programming standpoint, each of these elements has the ability not only to display itself, but also to collect user input and ensure it is in the correct format (that is, perform format QC). Furthermore, as the user fills in the individual element interfaces, weather information is extracted and the template line's corresponding surface condition is updated.

Note that there is *not* a one-to-one correspondence between weather elements in a template line and the fields that make up the corresponding surface conditions. Typically the information entered by the user is expanded so all the useful and *verifiable* information is extracted. For instance, though the user might enter "17012G20KT" in the template's "Wind" element, the fields in the surface condition object affected by this entry would be:

Wind Direction (170 degrees)

Wind Speed (12 knots)

Wind Gust (20 knots)

Additionally, the units of measurement in which the user enters the data into the template elements do not have to match the units in which the data is stored in the surface condition. For instance, visibility information is typically entered in meters in a TAF, but is entered in statute miles in a METAR observation. It would be redundant to provide separate attributes in a *WeatherCondition* object to maintain the same visibility data in two different units. It is the responsibility of the element in the report template to ensure that the data is passed to the corresponding weather condition object in the proper units and format.

3.2.3.2 Report Validation and the JESS Object

When the system is first initialized, an instance of the JESS Rete engine is established. Furthermore, JESS is initialized with the weather-domain rules

pertaining to the types of reports that will be produced[ii]. As *WeatherReport* objects and their associated *WeatherCondition* objects are instantiated, they must be registered with the JESS inference engine interface (hereafter referred to simply as JESS), as depicted in Figure 3-1 and expanded in Figure 3-3. Additionally, the JESS object is initialized with other station-specific weather information. The type of information (or facts) that may be passed to the JESS inference engine is limitless: category information, climatological data for any weather parameter, elevation, station longitude and latitude, climactic region, or other data upon which a meteorological rule might be based. (Compare Figure 3-3 with Figure 2-2, from §2.3.2).

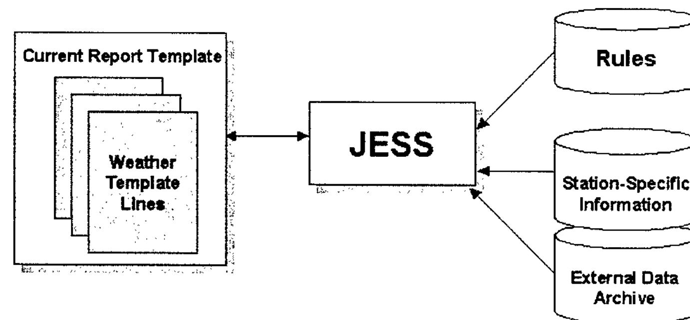


Figure 3-3: JESS Inference Engine Design

The only requirement for this information is that it must be stored as a JavaBean object and properly registered with JESS. As the user edits the report, adding, deleting, and changing data in the template, "property-change listeners" registered by JESS with each of the JavaBean objects are notified of the "property-change events" caused by the data value changes, and any affected rules are run (refer to §2.3.3.4 for information regarding JavaBeans). Feedback messages

[ii] In the implementation described in Chapter 4, TAF and METAR observation report templates are demonstrated. Rules pertaining to TAFs and observations are batch-fed into JESS at instantiation. To improve performance and reduce the required size for JESS' working memory, if it is known that only observations will be produced on a specific machine, for instance, only observation rules need be loaded. If it is later determined that a TAF report needs to be generated (for instance, in the event that an OWS requires backup from its Weather Flights), TAF rules can be batched when required.

generated by rules are sent back to the user via a message display area on the screen (see §3.2.4.2 for more information on rules and §3.2.3.4 for user feedback).

3.2.3.3 Report Activation and Verification

Once a report has been completed, the user disseminates it by marking it "finalized." At this point, if there are no errors that violate report syntax[iii], the system marks it "active." The JESS object is notified of the new pending report, and runs final validation rules and initial verification rules against it[iv]. If there is any further feedback, the user is notified. Otherwise the report is placed in the active archive, and, if applicable, supersedes the old report of the same type from the same location for verification purposes (see Figure 3-4).

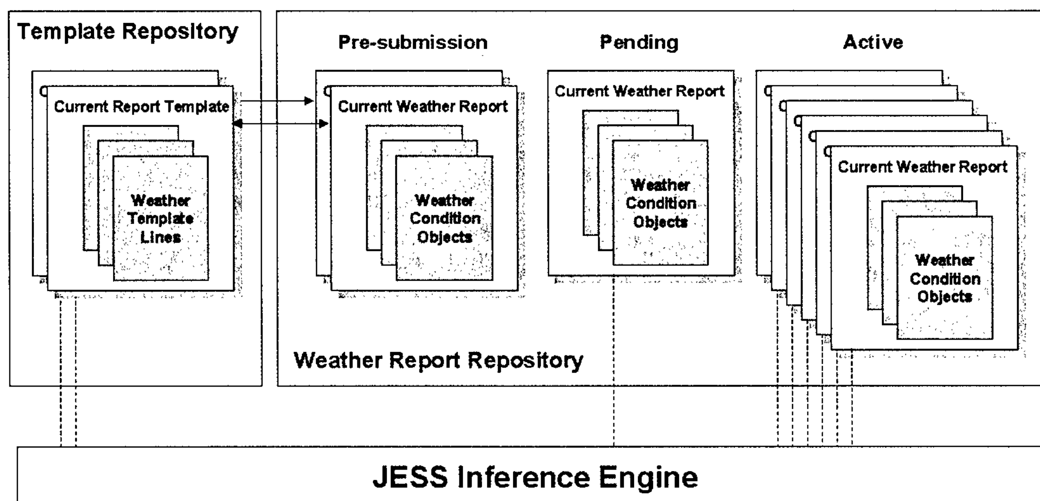


Figure 3-4: System Template and Report Repositories

From this time until the report is no longer valid or is superseded, the JESS inference engine will continue to apply verification rules to the report. Rules will be rerun any time applicable data in the external data archive is updated (see §3.2.4.9 for information on concurrent weather sets.) When

[iii] Reports with format errors may not be disseminated.

[iv] It is necessary to run final validation rules since in some circumstances during product generation, rules for missing data will not fire because it is assumed that the user has simply not yet entered the information. Once the report is marked for activation, these missing-data rules are allowed to fire.

feedback messages are generated based on a specific rule or set of rules, the user is notified. Note that as a report is upgraded from pre-submission to pending to active, the JESS applies different rules depending on the report's status.[v]

3.2.3.4 Feedback Messages

Feedback messages to the user may come in various forms, and it is not the intent of this design to specify any particular convention. Possible implementation options are color-highlighted entry blocks with corresponding messages in a separate message window, a report-status window indicating verification and validation status for all reports (pre-submission and active) currently being monitored, or pop-up dialogues and/or audible warnings when important messages (such as amendment or correction required) are generated. The implementation example discussed in Chapter 4 employs highlighted fields and textual message feedback.

Regardless of message-display format, a mechanism to transmit messages from the originating source is designed into the system. Messages are generated from two sources: The report template elements generate format QC messages, while JESS (based on applicable rules) generates "content QC[vi]" validation and verification messages.

Mechanisms to cancel issued messages are designed into the system as well. Because messages may be generated from similar elements of different weather report templates (or based on the same *WeatherCondition* attributes of different *WeatherReport* objects), messages must be uniquely identified by report type, base, element or attribute name, message type, and line number in the report[vii]. In this design, it is the responsibility of the report elements to uniquely

[v] Since observations are not "verified" once submitted, no verification rules are run on an observation. Only forecast reports will be verified.

[vi] The concept of "content QC" as introduced in Chapter 1 (§1.3.2) has been incorporated into the validation process, handled by the JESS inference engine.

[vii] Messages regarding report-level attributes do not need to be identified by line number.

name and keep track of the format messages they send. Since JESS is capable of accessing only the methods of the JavaBeans registered with it, messages generated by JESS are directly relayed to the *WeatherCondition* or *WeatherReport* objects whose attributes triggered the message generation. For this reason, in this design, *WeatherReport* and *WeatherCondition* objects must also keep track of identifiers for JESS messages generated on their behalf, so that cancellations can be made when messages are retracted.

3.2.4 System Components

The previous section describes a high-level representation of *what* the system does as the user goes through the process of choosing a report template, editing the information in the report template, and activating the associated report. This section details the design specification that indicates *how* the system performs these processes.

The complete UML design model diagram is shown in Appendix C, and the associated data dictionary, automatically generated by Rational Rose® from the object, method, and attribute descriptions is in Appendix D. Although pertinent pieces of the model object hierarchy are illustrated in figures throughout the text, it will be useful to refer to the full model diagram and data dictionary to see the complete picture of how objects interact with each other.

3.2.4.1 Station Information and Category Sets

Since the JESS knowledge base includes station-specific information, a method to retrieve and store this information must be included in the system. Station-specific information can include station identifiers, such as the four-letter International Civil Aviation Organization (ICAO) code and the six-digit block-station number; location information such as latitude and longitude, climactic region, and elevation; and weather category information—the threshold values for various parameters used to verify forecasts (see §2.4.1.1).

Note that though the category information is not *station*-specific—in the various implementations of the TAFVER program, recall that category schemas have been *Major Command* (MAJCOM)-specific—it must still be determined, based on station, what category system will be used. Furthermore, since bases are now most logically grouped by their responsible Operational Weather Squadrons (OWSs), and to provide more flexibility than the old TAFVER programs allowed, the new concept of *base-type* has been incorporated into this design. With this new additional layer of refinement, bases under the same OWS may employ different category schemas depending on their missions. Figure 3-5 illustrates the relationship between base type and category set.

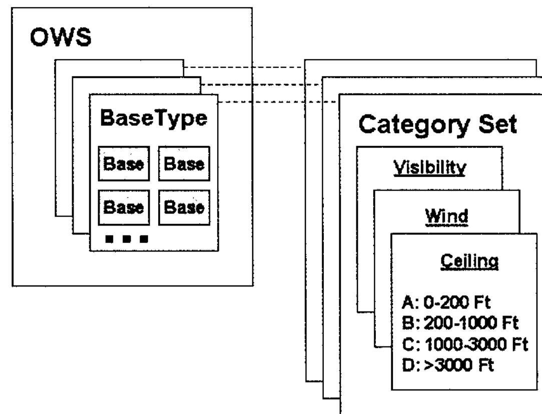


Figure 3-5: Base-Type and Category Sets

The bases themselves must have *properties* associated with them. Data stored in the *BaseWeatherStation* objects is identification and location information. Data in the *BaseProperties* object is weather-specific information—as general as the climactic region, and as specific as "Average high temperature for January," for example. Additionally, the *BaseProperties* object contains identifiers to the station(s) whose reports would be used to verify a forecast report. For instance, the *verificationSurfaceStationSourceIdentifier* would most likely be the same identifier as the base itself. The *verificationUAStationSourceIdentifier* would be the identifier for the nearest Upper Air station associated with the base, and so on.

Figure 3-6 depicts the UML classes associated with bases and categories. Note the hierarchy of container classes associated with *WeatherCategorySet* objects. This hierarchy is designed in such a way that categories for particular elements may be added or removed, and category threshold values may be changed at runtime.

This design provides only a few examples of the types of data that could be stored in these objects. The decision to include specific data depends solely on the types of rules employed. If there are no rules based on a particular datum, there is no reason to maintain it.

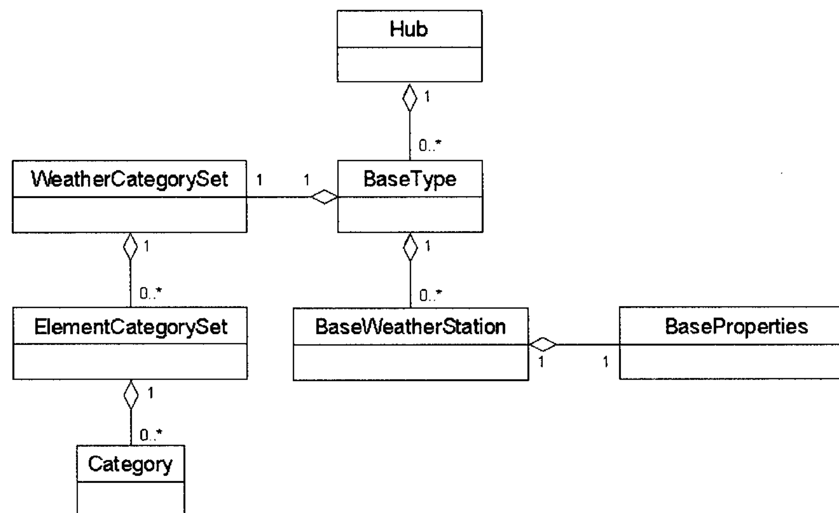


Figure 3-6: Base and Category UML diagram segment

This design is not concerned with *how* the external data is maintained, nor even *what* data should be maintained. Some of the data required for this application would come from the Master Station Catalog—a database that maintains base identification and description information. Ideally, a mechanism to add and associate new data with the application at runtime would be preferred.

The inclusion of such a data maintenance component in the design (the *SystemMaintenanceInterface* illustrated in Figure 3-7) is solely to show a requirement for a maintenance function that facilitates runtime manipulation of

base, category, rule, and template information. The essential aspect of external and station-specific data maintenance in this system is that the data must be stored in a form that enables JESS to extract it. In this design, JESS recognizes the *BaseProperties* and *Category* objects, and rules can be based on these objects' attributes. Furthermore, every *ReportTemplate* and associated *WeatherReport* instantiated is necessarily associated with a *BaseWeatherStation* object (and hence its associated *BaseProperties* and *Category* objects).

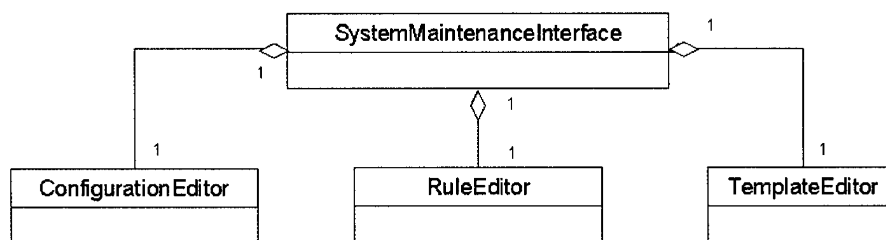


Figure 3-7: SystemMaintenanceInterface UML diagram segment

3.2.4.2 Rule Sets

The rules used by JESS are report-type specific—that is, the set of rules used to validate and verify one report would not necessarily be used to validate and verify another type of report.[viii] Because the system is designed with the use of templates, the type of report is unknown until the user specifies it at runtime. As mentioned previously (§3.2.3.2), JESS can be initialized with full rule sets for each report type at instantiation, or rule sets pertinent to specific report types can be batched-defined in JESS at runtime, as they are needed.

As designed, sets of *Rule* objects (*RuleSets*) earmarked for a particular report type are stored in a *RuleSetRepository* object (see Figure 3-8). Report

[viii] In fact, since observations obviously cannot be verified, JESS rules for observations would be solely for the purpose of report validation—primarily ensuring that the combinations of data elements entered have been logical and complete.

templates (described in the next section) are associated with a default *RuleSet* object.

The format for the individual rules as implemented for this research is the one recognized by the JESS inference engine. If this design is implemented in C++ or other language, CLIPS syntax or other inference engine syntax should be used, as appropriate.

Additionally, as with the *ConfigurationEditor* maintenance function for categories and bases, a *RuleEditor* is designed into the system (refer to Figure 3-7). This rule editor should provide the ability to add, remove, or change rules so they may be batch-defined in JESS at runtime. This editor component, combined with the JESS function call methods included in the core JESS package [19], provides a very flexible capability of modifying report rule sets without the need for recompiling code, or for that matter, halting program execution at all.

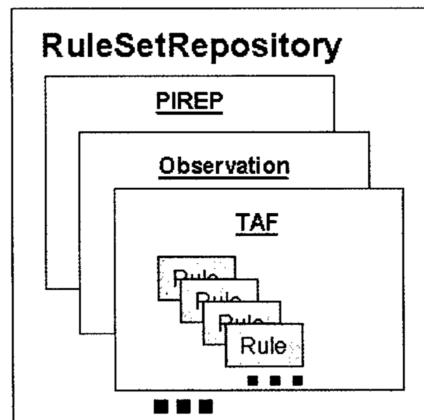


Figure 3-8: Rule Set Repository

It is important to point out that the rules themselves are the heart of the system. To be effective, rules must be designed and tuned by a domain knowledge expert (a meteorologist) to ensure that when they are applied, they provide the forecaster or observer with technically accurate and practically applicable information without becoming an annoyance, as can be the case with critic systems (see §2.2.5.3).

Since the implementation discussed in Chapter 4 is simply to show proof of concept, the rules implemented for this research are limited in scope and specific in nature. In reality, some rules might involve complex combinations of external surface and upper air elements, and some might simply involve the relationship between a couple of elements of the report currently being generated. See Chapter 5, §5.2.3 for a description of how new rules can be created so that the results of research into such areas as fog forecasting, rain or snowfall potentials, and other forecast indicators can immediately benefit operational forecasting efforts.

3.2.4.3 Report Templates

In this design, forecasters and observers employ *Template*s to input weather report data. At runtime, when a specific type of forecast report blank is requested, a *Template* object matching the desired report type is loaded and cloned for data-entry use.

Like the station-specific information and rule sets, the *Template* objects themselves must be stored persistently for runtime access. Though this design does not specify a method of persistent storage for templates, the function of accessing the stored template information is handled by the *TemplateRepository* (see Figure 3-9).

As is depicted in the figure, templates for various reports are maintained in the repository. Each template has a *DistinctTemplateLine*[ix] associated with it. A *DistinctTemplateLine* is a set of *ElementIdentifiers*—names—that are directly related to the *ReportElement* objects to be placed in the template.

[ix] When a report template is instantiated, lines may be inserted and deleted as necessary. When new lines are inserted, they are constructed in the image of the *DistinctTemplateLine*. Though a report (such as a TAF) may have only one *DistinctTemplateLine* associated with it, the template line may be repeated so that the actual report will have many lines of the same kind—that is with the same sequence of elements. A template property attribute may be set to indicate the maximum number of lines allowed in a report.

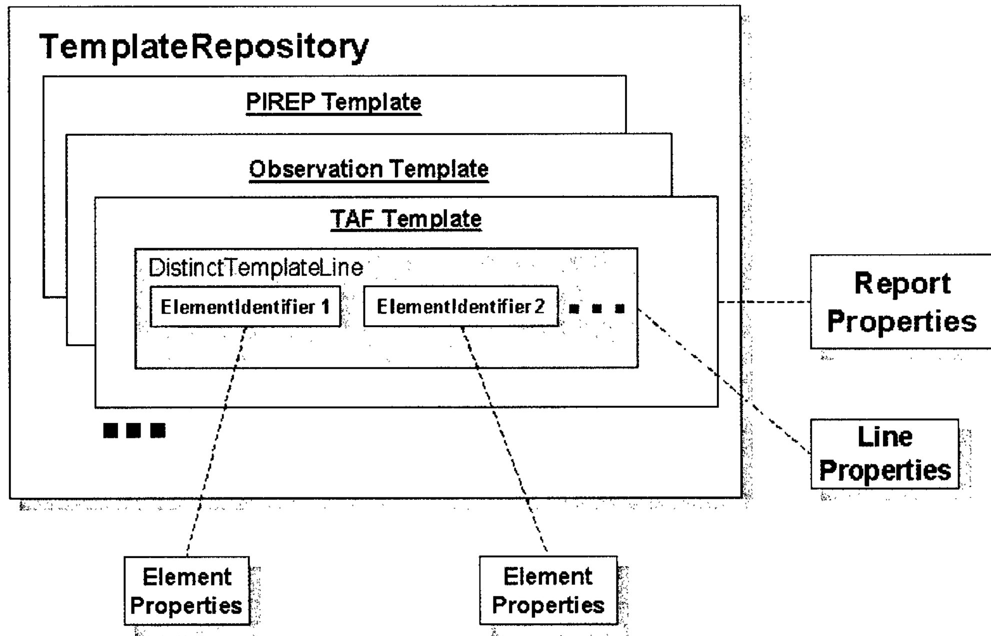


Figure 3-9: Template Repository

Each level of the Template—the *Template* object itself, the *DistinctTemplateLine* and each *ElementIdentifier* object—has a property object associated with it (see Figure 3-10). These properties contain descriptive information that can be referenced at various stages of report production. For instance, attributes of the *ReportProperties* objects indicate the type of report (forecast or observation), the duration of time the report is valid, category and rule set object references, the number of hours of previous reports to keep for verification and validation purposes (see §3.2.4.9 concerning concurrent weather sets), and other information. *ElementProperties* object attributes indicate, for example, whether the element is a required field or if the element is to appear only in the first or last line of a report (as TAF forecast high and low temperature elements do). As new *WeatherReportTemplate* objects are created in the image of the *Template* objects, the associated properties objects are cloned as well.

3.2.4.4 Weather Report Template Instantiation

Because of the interrelationship among objects, the process of report instantiation is fairly complex. At runtime, when a forecaster invokes a report template through the *ReportManager* object, the *ReportManager* directs the instantiation of a new *WeatherReportTemplate* and a related *WeatherReport* object (see Figure 3-10). The *WeatherReportTemplate* object, in turn, creates a *WeatherReportTemplateLine* object, and passes it a reference to the *TemplateRepository*. The *WeatherReportTemplateLine* is then responsible for creating a new *WeatherCondition* object that will hold all of the meteorological information associated with the data input by the forecaster through the template (refer back to Figure 3-2). The *WeatherReportTemplateLine* object also creates an instance of a *ReportElementRegistrationManager*. It is the responsibility of this object to register each *ReportElement* and identify it as the one and only element that can access and modify a particular attribute or set of attributes of the *WeatherCondition* or *WeatherReport* objects associated with the template.

Note from Figure 3-10 that the individual user interface entry elements (visibility, wind, temp, etc.) all inherit from the super class *ReportElement*. The *ReportElement* super class handles all of the registration, display functions, message tracking, and *WeatherCondition* and *WeatherReport* updates. The various subclasses of the *ReportElement* class make up the library of user interface entry elements that may be used to construct a report template. They must instantiate abstract methods inherited from the *ReportElement* class such as a *verify()* method (to ensure user-input data is formatted correctly) and get-methods for arrays containing lists of the *WeatherReport* and *WeatherCondition* attributes the element modifies. See the Data Dictionary in Appendix D for detailed method information.

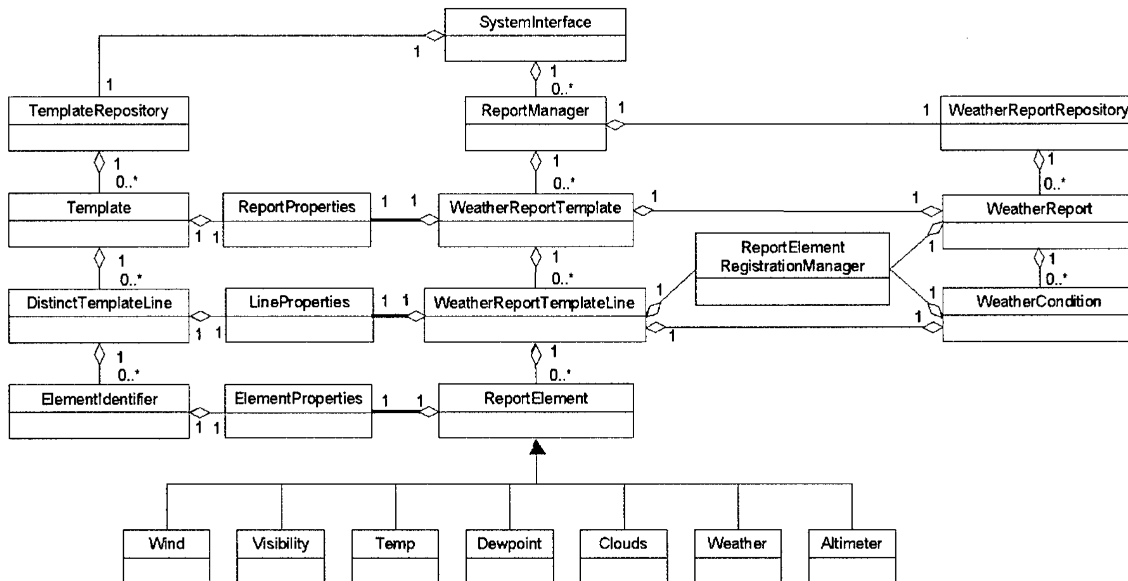


Figure 3-10: WeatherReportTemplate and TemplateRepository UML diagram segment

After creating the *WeatherCondition* and registration manager, the new *WeatherReportTemplateLine* passes a self-reference and a registration manager reference to the *TemplateRepository*. The *TemplateRepository* checks the *ElementIdentifier* and *ElementProperties* objects associated with the line to determine the names of the *ReportElement* objects it must instantiate to populate the *WeatherReportTemplateLine*. It constructs the new *ReportElement* objects, and then registers them via the registration manager. (Note that for simplification's sake, references between the *ReportElementRegistrationManager* and the *TemplateRepository* are not shown in Figure 3-10. As designed, this reference is established during element instantiation and registration, and then immediately destroyed. Refer to Appendix C for the complete model specification.)

Once all the objects associated with a weather report and weather line are constructed, references to the cloned report, line, and element properties objects are passed to the *WeatherReportTemplate*, the *WeatherReportTemplateLine*, and the *ReportElement* objects with which they are associated. In Figure 3-10, these references are depicted by the bolded aggregate relationships. (Additional

references to the *ReportProperties* and *LineProperties* objects are passed to the *WeatherReport* and *WeatherCondition* objects as well). Furthermore, a reference to the *WeatherCondition* is inserted into the *WeatherReport* object via the *ReportManager*, and the *WeatherReport* is placed in the *PreSubmissionReportRepository* (see Figure 3-4).

As the user inserts additional lines into the report, new *WeatherReportTemplateLine*, *WeatherCondition*, and *ReportElementRegistrationManager* objects are constructed as before. The *TemplateRepository* is called upon to populate the new lines, and to ensure that the *ReportElement* objects adhere to the property constraints.

Some elements might have to be moved from one line to another to ensure adherence to these constraints. For instance, in a TAF report, high and low temperature forecasts appear only in the first line of the report. If a line is inserted before the first line, these "first-line-only" elements must be moved from the *old* first line to the *new* first line[x]. The *TemplateRepository* contains methods to move elements between lines when necessary. To facilitate this, the *TemplateRepository* not only maintains sets of elements applicable to individual reports, but sets of elements representing first, last and standard lines in the reports. Because of this, the element transfers as a result of line insertions and deletions can be accomplished in O(n) computational time and cause little delay in practice.

3.2.4.5 JESS Registration and Template Display

As each new *WeatherReport* is constructed, the *ReportManager* passes a reference to the new report object and its properties object back to the *SystemInterface* so that they may be registered with the JESS Rete object. Likewise, *WeatherReportTemplateLines* pass references to *WeatherCondition* objects

---

[x] Some reporting stations require the high and low temperature elements in the last line of the report instead of the first. In this case, when a line is appended, the elements must be moved to the new *last* line of the report.

they construct back to the *ReportManager* to register with JESS. At this point, JESS, in turn, registers JavaBean property change listeners with each of the attributes of the *WeatherReport*, the *ReportProperties*, the *BaseProperties*, the *WeatherCategorySet*, and the *WeatherCondition* objects associated with the report. (Base and category objects are referenced through the *ReportProperties* object).

After all objects associated with the new *WeatherReportTemplate* have been constructed and initialized, and the *WeatherReport* and associated objects have been registered with JESS, the template's *display()* method is called, and the template field is displayed on the screen[xi]. Once displayed, the user may enter data in the element fields. As data is entered and the user moves from field to field, the *ReportElement* objects will automatically perform data entry format QC via their *verify()* methods. Additionally, they will update their associated *WeatherCondition* and *WeatherReport* objects via the *ReportElementRegistrationManager.* As attributes of these objects are updated, applicable JESS rules will fire and advise the forecaster of alternatives through feedback messages.

3.2.4.6 Report Finalization

Once the user indicates the report is finished and ready for dissemination, the *WeatherReportTemplate*'s *finalize()* method is called. The *WeatherReport* is moved to the *PendingReportRepository* (see Figure 3-4) and the *WeatherReport*'s status is updated to indicate the report is ready for submission. This signals JESS to run final validation and initial verification rules against the report. If discrepancies are found, such as missing required element data, the report is returned to the *PreSubmissionReportRepository* and the user informed of the

[xi] Display of text box, text box labels, and other GUI components is completely implementation language dependent, and therefore not part of the design specification. In the implementation that supports this research, discussed in Chapter 4, the *WeatherReportTemplate*, *WeatherReportTemplateLine*, and all of the *ReportElements* are implemented as extensions of Java.awt.Panel components. The *ReportElement* objects also have attributes of Java.awt.Label, Java.awt.Choice and/or Java.awt.TextField components as applicable to the element's function. See [5] for information on Java.awt classes.

problem. Otherwise the report will be moved to the *ActiveReportRepository* and simultaneously disseminated. If the report supersedes another report of the same type from the same location, the old report is deleted from the active repository, and replaced by the new report.

Throughout the report's valid time, if the report is a forecast, JESS will still be actively applying verification rules. These rules compare attributes of *WeatherCondition* objects from external sources to those contained in the newly created *WeatherReport*. For instance, a TAF *WeatherCondition* object would be verified against the *WeatherCondition* object that corresponds to the observation from the same location, at the same valid time (see §3.2.4.9 for information on concurrent weather sets). The forecaster could be notified not only when the forecast goes "out of category" (requires immediate amendment), but also when negative trends are perceived. For instance, if the forecast called for a rising ceiling over the forecast period but actual observations indicated a lowering ceiling, the forecaster would be advised. The type of feedback the forecaster receives, however, is entirely based on the type of data registered with JESS and the makeup of the report type-specific rules based on the data.

3.2.4.7 Report Template Design

Because *ReportElement* objects are dynamically instantiated at runtime by name, *WeatherReportTemplates* that use various combinations of the supplied library of *ReportElement* objects can be designed, stored, and invoked for use in weather report generation at runtime[xii]. Furthermore, reports can easily be modified to accommodate location- or MAJCOM-specific requirements. For instance, if one OWS requires the forecast high and low temperature elements to appear in the first line of a TAF, while another requires them in the last line, this can easily be accomplished under this design.

[xii] In a Java implementation, this is accomplished through the Class and Constructor Java classes [4] (see §4.2.7 for implementation details).

3.2.4.8 Report Element Registration

The importance of *ReportElement* registration through the *ReportElementRegistrationManager* should now become clear. Since report templates can be designed and modified at runtime, it is vital to ensure that they don't contain more than one element that modifies a particular *WeatherCondition* or *WeatherReport* attribute. For example, a single *WeatherReportTemplateLine* could not include a *"VisibilitySM"* element (typically used in a METAR observation report, for visibility reported in statute miles) as well as a *"VisibilityMeters"* element (typically used in a TAF report), since both of these elements modify the "visibilitySM" attribute of the *WeatherCondition* object. If more than one element tries to register to change the same *WeatherCondition* or *WeatherReport* attribute, an error will be generated, and the operation would be disallowed.[xiii]

As designed, once elements register themselves with the *ReportElementRegistrationManager*, they may access *WeatherCondition* or *WeatherReport* attributes only by passing the attribute name and new value to a *requestChange* method. This method first checks with the *ReportElementRegistrationManager* to see if the element is authorized to alter the requested attribute, and then checks to see if it has a set-method for the attribute name that was passed. If so, the element value is updated.

Furthermore, all element values are passed as String objects, and are only converted to their equivalent floating point or integer value, if necessary, only by the *WeatherCondition* or *WeatherReport* attributes' set-method.

This design decision ensures consistency among all user interface entry elements. The individual elements can collect and verify the entered (String-

[xiii] In any practical implementation, a "template design builder/editor" would be used to construct a template. This designer application would not allow the construction of a template using elements that modify the same *WeatherCondition* or *WeatherReport* objects. For instance, if a new report was designed, it could employ either the *VisibilitySM* or the *VisibilityMeters* element, but not both, since both of these elements modify the same attribute of the *WeatherCondition* object. See §4.2.4 for more information.

formatted) information in any way necessary. Intermediate conversions of user-entered data to floating point or integer values are usually required in *verify()* methods. In every instance, however, the information must be converted to a String object to update the *WeatherReport* or *WeatherCondition* objects through the *ReportElement*'s *requestChange()* method.

3.2.4.9 Weather Reports and Concurrent Weather Sets

As mentioned previously, and as depicted in Figure 3-2, *WeatherReportTemplates* are the interface through which the user enters weather information and the *WeatherReport* object is the object format recognized by the system for storing and maintaining weather data. All weather data, whether generated internally through a template, or input and parsed from an external source, is placed in a *WeatherReport* object.

Each *WeatherReport* object contains identification information (identifiers, status, references to properties, valid times, etc.), report-level weather information (for instance report high and low temperature forecasts in the case of a TAF), and a set of *WeatherCondition* objects. As described in §3.2.3.1, the *WeatherCondition* objects contain all of the information that describes the observed or forecast state of the atmosphere for a given time period[xiv]. A *WeatherCondition* may be a *SurfaceCondition*, as would be associated with a surface observation report, or, in the case of an *UpperAirReport*, an *UpperAirCondition*, a *SoundingCondition*, or an *ObservedCondition*. Figure 3-11 illustrates this hierarchy.

[xiv] The *WeatherCondition* may be a permanent or temporary condition. The valid times for a temporary condition are specified in the *ReportProperties* object.

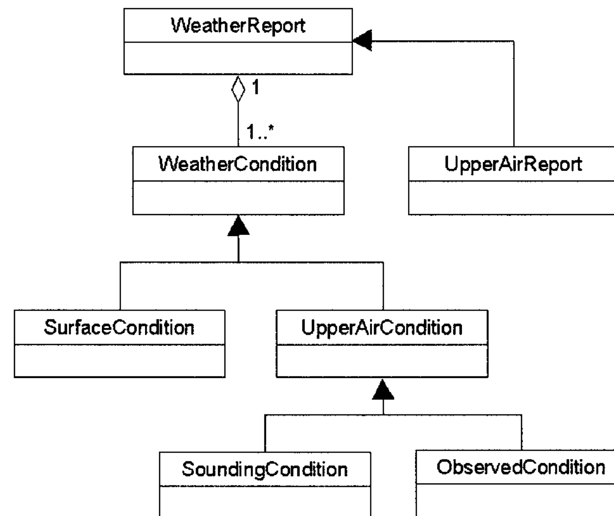


Figure 3-11: ***WeatherReport*** **Hierarchy**

Note that the *UpperAirReport* class inherits from *WeatherReport*, and contains additional report-level, but upper-air specific, information. For instance, this object might contain attributes, such as Lifting Condensation Level, which apply to the entire report. The *ObservedCondition* and *SoundingCondition* object subclasses of the *UpperAirCondition* object are included because of the distinction between an upper air report generated from a radiosonde (weather balloon) and that observed by aircraft and reported to the ground via a PIREP.

When a *WeatherReport* is instantiated by the *ReportManager*, it in turn instantiates a *ConcurrentWeatherSet* object. The *ConcurrentWeatherSet* is the set of all external weather data that will be used to verify and validate the report. In calling the *ConcurrentWeatherSet* object's constructor, the *WeatherReport* passes a reference to the *BaseProperties* object of the *BaseWeatherStation* associated with the report and selected by the user when the *Template* was first invoked. From this reference, the *ConcurrentWeatherSet* can obtain the identifiers of the

verifSurfaceStationSourceID and the *verifUAStationSourceID* attributes, as well as a set of any other alternate stations.[xv]

When the *ConcurrentWeatherSet* is first instantiated, it registers itself with the *ExternalSourceReportRepository*, which maintains a set of all *WeatherReport*s required for validation and verification use[xvi]. After the *ConcurrentWeatherSet* registers with the repository, it accesses its parent *WeatherReport* object's *ReportProperties* object to determine how many reports, or what duration of time reports are to be kept for validation or verification purposes. Since some forecast indicators are based on observed changes in weather parameters over time, JESS requires access to more than just the current observations. The *ReportProperties'* *hoursOfObsToKeep* and *numberOfSoundingsToKeep* attributes indicate the number of surface observation and upper air reports to keep on hand. These report references are retrieved from the *ExternalSourceReportRepository* and kept in First-In-First-Out (FIFO) containers within the *ConcurrentWeatherSet* objects. The references are also passed to the *SystemInterface* to be registered with JESS.

Meanwhile, the *ExternalSourceReportRepository* continuously monitors the input stream for updates to the reports (see Figure 3-12). As updates are received, the attributes of the *WeatherReport* objects are updated to reflect the changes. Simultaneously, the JavaBeans property change listeners associated with each of the external *WeatherReport* and associated *WeatherCondition* objects registered with JESS update the corresponding JESS facts. Any verification or

---

[xv] Although in this design a sample "*alternateSourceStationSet*" attribute of *BaseProperties* has been included, any references to other sources must be generic and apply to any station generating a TAF. For instance, a set of "upstream stations" might be used to satisfy specific JESS rules. If a generic source or set of sources does not apply to a specific base (i.e., a U.S. West Coast station might not have "upstream stations" in the sense that those stations might be used in JESS rules), the set reference is simply left "NULL," and those rules will be ignored. Likewise, if a station for which a forecast report is valid has no corresponding observation report, the *verifSfcStationSourceID* would be left NULL and no verification rules would fire.

[xvi] It is assumed that any reports made available to this system will be passed in the form of a *WeatherReport* and associated *WeatherCondition* objects. External data not in this form must be properly parsed and formatted for inclusion in the *ExternalSourceReportRepository*.

validation rules involving attributes of these external data sources will fire, and feedback will be displayed to the user as before.

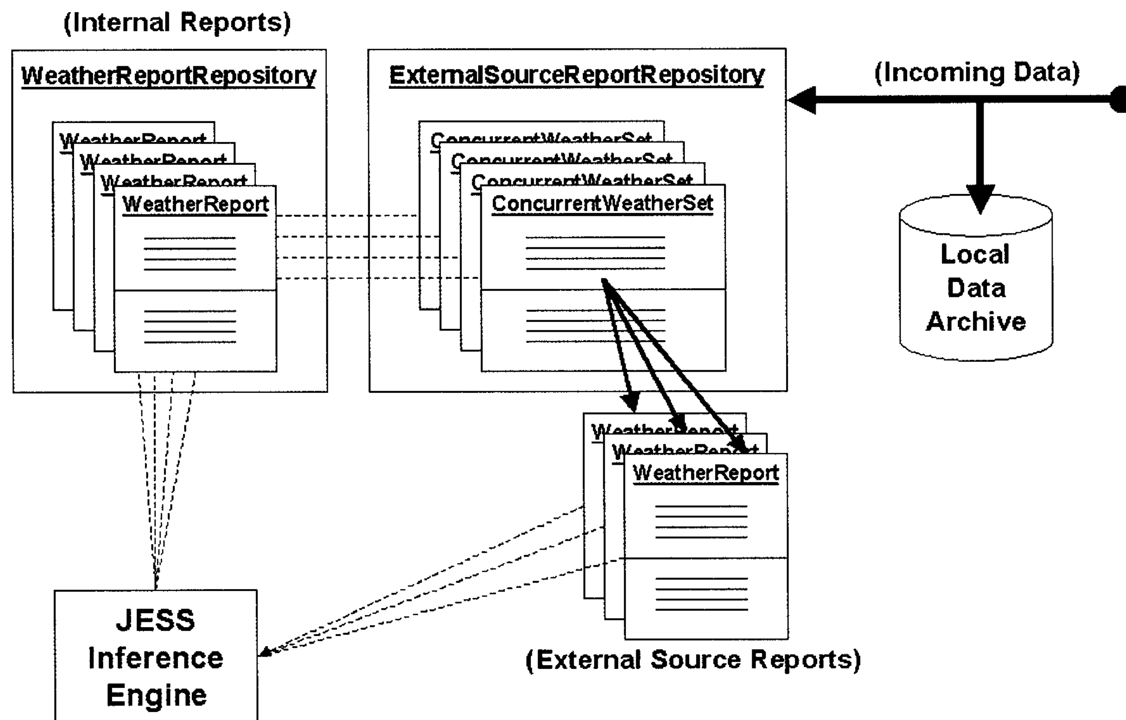


Figure 3-12: Concurrent Weather Sets and the External Source Repository

When a *WeatherReport* is superseded, the reference to the old report's *ConcurrentWeatherSet* object is copied to the new report. In this way, *ConcurrentWeatherReport* objects must be created only once for any given report type valid for any given base. As long as reports are continually superseded (as required by weather forecasting and observing business rules), the associated *ConcurrentWeatherSet* objects will be maintained in the repository, and re-initialization will not be necessary.

3.2.4.10 JESS Agent and the Verification/Advice Log

With the exception of the actual user input template, the bulk of this application places weather report information in a format useable with the JESS agent (as well as any automated metrics program). Once the agent is able to

match specific data with its generic rules and facts, it can use the JESS inference engine to provide useful feedback to the user.

While generating a report, rules whose fact-patterns have been satisfied could indicate that, based on available data, the system has determined a better alternative to the user's report.[xvii] Since the system is only as good as its rules, after further study, some rules may be found to be in error or in need of adjustment. To aid in adjustment, this design calls for the implementation of a log file to store the circumstances surrounding each rule that fires. The log will store such information as the advice given, the applicable forecast parameter(s), whether or not the forecaster heeded the advice, and, when the report becomes active (and is verified), the actual observed parameter value(s).

Through analysis of this log (one for each report type valid for each base), it will be possible to determine how well or how poorly a particular rule is performing.

3.3 Summary

This chapter has detailed the user interface application design based on the data requirements for implementing a critic system. If implemented as designed, this system would provide a system whereby weather reports could be designed, modified, and invoked at runtime, and would provide METAR-standard format QC, as well as content QC for Air Force Weather products. If given a suitable set of meteorologically sound rules, it would provide the forecaster or observer with feedback if favorable alternatives to input data exist.

In Chapter 4, a partial design implementation to show proof of concept is discussed. Issues regarding system performance and rule tuning are addressed.

[xvii] Of course, the forecaster may choose to ignore system advice for non-format related suggestions.

4 Implementation and Functional Analysis

4.1 Overview

In Chapter 1, the need for automated quality control (QC) of Air Force Weather (AFW) text-based weather products is discussed. A new product generation process that employs format QC as well as domain-specific content QC is proposed. Chapter 2 presents background information on previous implementations of AFW metrics programs (TAFVER) and introduces the concept of a critic system to aid forecasters and observers in product generation (§2.2.5). It is suggested that such a system would lend itself to the report generation process proposed in §1.3.5

In Chapter 3, a specification for the design of such a system is presented. The various components of the system are explained in detail. It is suggested that with the implementation of such a system, the proposed new process would effectively be realized.

The implication of the previous chapters is that with format QC, the syntactic quality of AFW products would improve. More significantly, it is further suggested that, through the use of rules developed by domain experts, forecast accuracy could be improved.

In support of this research, to show "proof-of-concept" of the design specification presented in Chapter 3, and to give some insight as to the extent to which AFW product quality would be improved by such a system, a portion of the design was implemented.

This partial implementation concentrated specifically on the interaction between *WeatherCondition*, *WeatherReport*, and *ReportProperties* objects, and the Java Expert System Shell (JESS) inference engine. It was intended to demonstrate, by example, that the dynamic nature of the system is feasible and

that the JESS inference engine can serve as a viable core for the implementation of a critic system.

More importantly, through the development of a small library of self-QC'ing TAF and METAR observation-specific *ReportElement* objects and rules, this demonstration is intended to show how errors in current AFW forecast and observation reports could be significantly reduced or eliminated, paving the way for data analysis including, but not limited to, an automated metrics program [27].

Finally, the demonstration is intended to suggest that, given the nature of a critic system and depending on the makeup of the rules, operational fielding of such a system could improve forecast accuracy and perhaps even forecaster ability (see §2.2.5.4).

4.2 Interface Implementation

It is important to note that the goals of this implementation were only those stated in §4.1 above. It was not it intended to be a finished product ready for operational use. Even the portions of the design that *have* been implemented have not been fully implemented. Specifically, most of the runtime configuration capabilities of the system have been simulated.

Suitable data objects (*WeatherReport*, *WeatherCondition*, *ReportElement*, and other objects) consistent with the design of this application and formatted for use with any data warehousing or data mart effort detailed in [27] have been constructed only to demonstrate their interaction with the JESS agent and to draw conclusions about their performance. Additionally, a few, simple rules have been developed to manipulate the data objects and return useful feedback based on user input.

In the next sections, specific, important aspects of the design and implementation are highlighted to add insight into the design philosophy, and to pinpoint the system's abilities and limitations.

4.2.1 Language

The design specification of Chapter 3 is language independent. As discussed in §2.3.3.4 and in §3.2.1, using JESS and JavaBeans provides streamlined mechanisms for communication between the weather data objects and the JESS inference engine. Because of the selection of JESS as the critic's core, the language used was Java.

For simplicity, the graphical user interface (GUI) widgets employed in the interface were Abstract Windowing Toolkit (AWT) components shipped with Java 1.1, rather than the newer Swing set of GUI components included in Java 2. Many of the extended library features of Java 2 were employed, however, in other aspects of the implementation. All of the container objects—sets, maps, linked lists, etc. come from the Java.util package shipped with Java 2.

4.2.2 JESS

The implementation employs JESS version 4.4. Less than a month before publication of this document, JESS 5.0 was released, adding new functions and features, and correcting bugs. In JESS 5.0, the syntax for defining a Rete engine output stream—providing the ability to watch JESS fact assertions and rule firings, and to query the fact and rule lists—was changed. Given the time constraints for this research, the code conversion to JESS 5.0 was not feasible.

One particular bug fix directly impacted this implementation. In version 4.4, when JavaBeans are "de-registered" from JESS through a method of the JESS *Funcall* object and the JESS function *"undefinstance"* (see [19] for more information regarding JESS syntax) their corresponding facts are not removed from the fact list as they should be. Since the JESS facts maintain a reference to

the JavaBean objects even after they've supposedly been de-registered, Java's automatic garbage collection routines are not run, and the old JavaBean objects maintain their state.

This presents a serious problem if old, de-registered JavaBeans contain the same identification information as newly created ones. For instance, if a user inputs data through a template into "line 2" of a TAF from Offutt AFB (KOFF), and then the line is subsequently deleted, the reference to this old line still remains. If a new "line 2" for the same TAF from KOFF is later created, there will be two JavaBean facts on the facts list with the same identification information. If the information conflicts (for instance, if the data entered in one of these lines satisfies a JESS rule, but the data in the other line does not), JESS will endlessly loop, alternately issuing and retracting warning messages regarding the data in question.

As a workaround, in the implementation, the identifiers (line numbers, report types, and base International Civil Aviation Organization (ICAO) codes) are given "null" values. Though this alleviates the looping problem, the facts corresponding to the old JavaBeans remain on the fact list, and so these objects are never actually removed from working memory.

According to Ernest J. Friedman-Hill, author of JESS, this problem has been fixed in JESS version 5.0.

4.2.3 Weather Data Object Attributes

In §3.2.2 it is pointed out that the majority of the application design is report data formatting and maintenance such that the JESS inference engine can access and manipulate weather information provided by the user and from external sources.

Though the design offers a very specific solution as to how the data should be formatted and maintained, object composition could conceivably vary

substantially within requirement guidelines given actual implementation and integration into existing weather systems. The *WeatherCondition* and *WeatherReport* objects are intended to capture all basic (that is, not derived) characteristics of the atmosphere, either forecast or observed, during a given time period. This is in keeping with the philosophy of a TAF or a METAR observation—a small number of sensible weather elements is observed or forecast in reports that have been deemed sufficient to describe the "state of the atmosphere" for practical purposes.

Given the design detailed in Chapter 3, in any operational implementation of this system, it is important to carefully select the specific attributes of the *WeatherCondition* and *WeatherReport* objects. Though new *ReportElement*s that can modify these attributes can be constructed and added to the library of GUI elements without affecting the rest of the system, changes to or deletions of the *WeatherCondition* or *WeatherReport* object attributes *will* have a great impact. The reason for this is that all *ReportElement*s in the library and all the report-specific JESS rules are based on these attributes. They address them by name, and expect them to hold values consistent with their advertised type.

As an example, if the convention were established to store visibility values in statute miles, and then later this convention were changed to meters, each of the *ReportElement* objects designed to modify the *visibilitySM* attribute of the *WeatherCondition* object would have to be modified to comply with the change in standard. In addition, all of the visibility-based rules designed for reports that employ *ReportElements* that modify the *visibilitySM* attribute would have to be revised to correspond with the change in units.

For these reasons, the attributes of the *WeatherCondition* and *WeatherReport* objects (and their subclasses) must be designed to be complete from the start. If this system were to be implemented for operational use, it would be important to

obtain the advice of domain-knowledge (weather) experts in the construction of these objects.

For this implementation, only the *WeatherReport* and *WeatherCondition* objects from the object hierarchy depicted in Figure 3-11 in §3.2.4.9 have been employed. It should be noted, however, that even in the complete design specification of Appendices C and D, the attributes associated with all of the weather data objects in this hierarchy are provided as examples only, since their presence in those objects are affected by the same problems listed above. The simplified object hierarchy and the set of attributes implemented in support of this research are depicted in Figure 4-1.

Given the discussion above, the attributes implemented are not intended to be complete. Rather they are intended as examples of the types of attributes necessary to describe the state of the atmosphere, and to demonstrate system capability. The attributes included are sufficient to describe most weather phenomena that can be forecast in a TAF. Attributes such as wind shear, runway visual range, and observed snow and rainfall amounts, for example, are not included.

Furthermore, with the exception of time attributes,[i] the data types associated with the information has been limited to basic, native types—character String, integer (int), and floating-point number (float). In a complete system implementation, it might be worthwhile to store cloud layers in an array instead of a String, so that the individual cloud layer heights might be more readily extracted for analysis and application to rules.

It should also be noted that a certain amount of *metadata* needs to be carried along with the weather information. Fields such as *reportValidStartTime* (the full start date and time this report is valid) and *supplementalReportModifier*

[i] Time elements are implemented as a Java.util.Date object so that the precise time and date of the valid time for any line in any report can be readily extracted.

(AMD or COR modifiers that indicate whether a report is a correction or amendment of an old report) are necessary to further describe the data contained in the report, and so must be included as attributes in these objects.

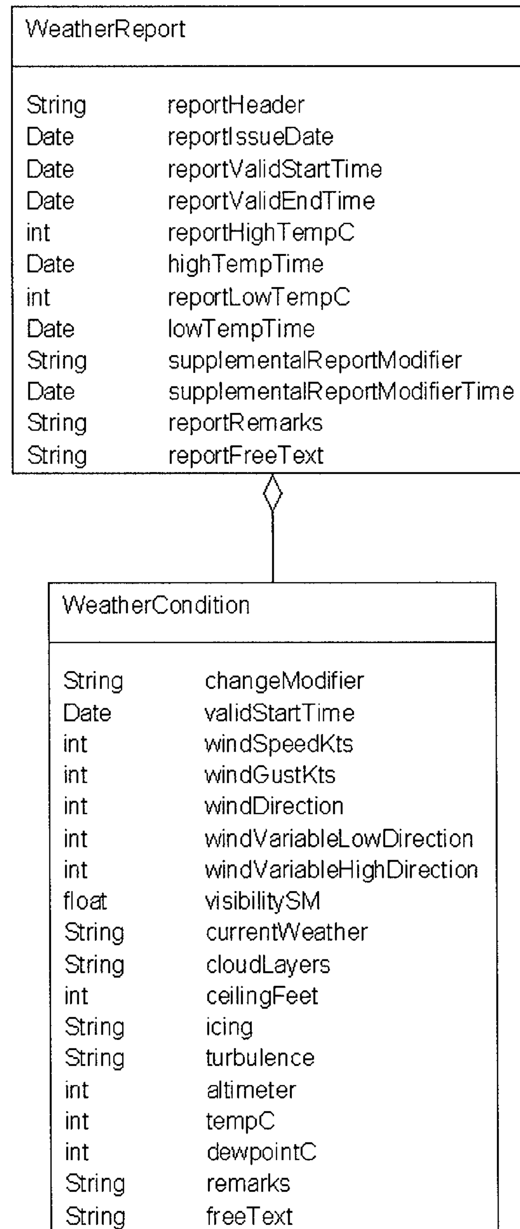


Figure 4-1: Simplified *WeatherReport- WeatherCondition* object hierarchy

The important thing to realize is that in the process of report generation, *all* of these attribute values are set externally through *ReportElement*s that are

registered to modify them. In some cases, only a few of these attributes will contain values. Only those attributes necessary to describe the "state of the atmosphere" as applicable to a particular report will be used.

For instance, a city weather summary report template may store only the narrative *freeText* attribute and perhaps the high and low temperature attributes. The rest of the attributes would be left with "null" values. It is up to the report template designer (or general AFW policy) to determine what particular attributes are important to maintain.

4.2.4 Report Elements

All report element objects—the GUI components that make up a template line—are designed as subclasses of the *ReportElement* class. The *ReportElement* object is designed as an abstract, generic object capable of supporting required registration and display functions of the individual elements in the interface.

As designed, the report elements can be "label only," "label and a text box" for user input, or "label and a choice box" for user selection of a small number of choices, depending on the element's function.

4.2.4.1 *ReportElement* Object Implementation

For this implementation, the *ReportElement* object is implemented as a subclass of the Java.awt.Panel class. As such, display, layout control, sizing, component color, etc., can all be controlled through methods of the *Panel* superclass. Furthermore, depending on the superclass constructor called by individual report elements that implement it, the *ReportElement* abstract class maintains attributes of "theLabel," (a Java.awt.Label object), "theField" (a Java.awt.TextBox), and "theChoice," (a Java.awt.Choice component).

The *ReportElement* object also contains Boolean indicator attributes that identify the element as "label-only," or "usesChoice." A "label-only" element is

one in which the value of the label is the value of the element itself. It is not editable, and is automatically set by the report element object upon instantiation.

The *SupplementalReportIdentifier* element (which modifies the *supplementalReportIdentifier* attribute of the *WeatherReport* object from Figure 4-1) is a label-only element that is included in the last line of every TAF report. When a TAF is an amendment and/or correction to an existing report, the label (and hence the element value) is automatically set with the value "AMD," "COR," or "AMD COR." Otherwise, the label is left blank. Upon instantiation of a report template that includes the *SupplementalReportIdentifier* object, the corresponding *WeatherReport* object's *supplementalReportIdentifier* attribute is automatically updated with this automatically generated value.

Similarly, "choice" elements contain predefined values. The difference is that the user can decide which one is applicable for a given line in the report. For instance, the *ChangeModifier* report element is an example of a choice element (see Figure 4-2 for an example of a *ChangeModifier* used in a TAF template). The *ChangeModifier* element is designed for user input of a change-group in a TAF line. The choices available to the user are "BECMG," "FM," and "TEMPO," each carrying distinct semantics in the METAR code (see Appendix A).

Figure 4-2: Sample report element GUI components

Note that some elements are necessarily dependent upon one another. For instance, the *ChangeModifier* element described above goes hand in hand with a *ChangeModifierTime* element. The *ChangeModifierTime* element is a standard labeled text box entry field in which the forecaster enters the four-digit change

time for the forecast line. If the *ChangeModifier* is set to "BECMG" or "TEMPO," the *ChangeModifierTime* must be in the form of $H_1H_1H_2H_2$, where H_1H_1 is the two-digit hour in which the change in weather state begins to occur (or is valid, in the case of "TEMPO"), and H_2H_2 is the two-digit hour that the change becomes effective (or the end valid time in the case of a "TEMPO"). The two-digit hours in both cases indicate the next occurrence of that hour (Greenwich Mean Time) during the valid time of the report. When the *ChangeModifier* element is set to "FM," however, the time reported in *ChangeModifierTime* is a four-digit time in the form of HHmm, the hour and minutes when the weather change is expected to occur.

Because of this difference, the *verify* method of the *ChangeModifierTime* element must take into account the value of the *ChangeModifier* element to properly QC user input. Furthermore, though the *verify* method of each element is typically called when the user inserts, deletes, or changes information in the text box associated with a particular element, in the case of elements such as *ChangeModifier* and *ChangeModifierTime*, the *verify* method of *ChangeModifierTime* must be called when changes to *ChangeModifier* occur. (Note that 0530 is a valid format for the *ChangeModifierTime* input when the *ChangeModifier* is set to "FM," but not if the second two digits represent a two-digit hour, as when *ChangeModifier* is set to "BECMG" or "TEMPO." Each of these distinct types of elements have been implemented as examples.

It should be noted that though the format QC functions (*verify* methods) of each of the elements implemented for this demonstration is designed to accept METAR-formatted weather data, the format QC for any given element is not necessarily complete. The purpose of this implementation is to show proof of design concept, and not to fully implement a TAF or METAR observation template.

4.2.4.2 ReportElement Subclasses

As mentioned in §3.2.4.4, the *ReportElement* object contains specific abstract methods that individual elements must implement. Among these are the *verify* method discussed in §4.2.4.1 above. The elements must also "advertise," in two separate String-type arrays—*modCondAttribs* and *modRepAttribs*, the names of the attributes of the *WeatherCondition* or *ReportCondition* objects they modify. As implemented, the names of the attributes modified by individual elements are hard-coded into these arrays in the individual element classes.

Though this works well (and would even suffice for an operational implementation) ideally an "element-builder" application should be built to graphically define existing generic report elements. The accepted user input data format characteristics (and therefore the functionality of the *verify* method) of these elements could be defined through a meta-language in much the same way field characteristics are defined in a Microsoft Access database table. With this type of an application built upon the system, report elements could be constructed and added to the library without compilation. An application to fulfill this function is, however, beyond the scope of this research.

4.2.4.3 ReportElement Services

Just as the report elements that inherit from the *ReportElement* superclass have responsibilities specific to the type of data they handle, the *ReportElement* class has responsibilities of its own. As mentioned in §3.2.4.4, it provides its subclasses with methods that enable them to register themselves with the *ReportElementRegistrationManager*, and to request changes to the attributes of the *WeatherCondition* and *WeatherReport* objects they modify.

Furthermore, the *ReportElement* maintains references to the *WeatherReportTemplateLine* in which the element resides, as well as the *ReportElementRegistrationManager* with which the element is registered. The

report element also provides methods so that individual elements may issue and retract warning messages based on user input (see §4.2.5 for more information regarding message handling).

Since all of these functions are vital to the operation of even the simplest implementation of the design specification, they have been fully implemented in support of the "proof-of-concept" demonstration.

4.2.5 Warning Messages and Report Status

As specified in the design, reports and report elements can have three different status levels. As a convention, they are "green," "yellow," and "red."

Elements (and reports) that are "green" have not caused the generation of any warning messages. It is not to say that those reports are ready for submission—a newly instantiated blank template is "green." It just means that any data that *has* been entered is properly formatted, and no fact-patterns have been matched causing a JESS rule to fire (and generate a warning message).

Elements (and reports) that are "yellow" contain correctly *formatted* data, but a JESS advice message has been received that suggests an alternate data value. Reports with "yellow" status can be finalized and submitted: they contain no syntactical format discrepancies. Though the capability exists for *ReportElement* objects to generate a "yellow" message should the need ever arise, "yellow" messages are at present always generated by JESS.

Elements (and reports) that are "red" contain syntactic format errors. "Red" format error messages are generated by the individual *ReportElement* objects when input data is formatted incorrectly, and by JESS when the values of two unusually unrelated *ReportElement* objects causes a violation of the syntax of the report format. See §4.3.2 for a practical example of such a case.

When a warning is generated by either a report element or by JESS[ii], a *Message* object is instantiated with a unique identifier, the message status (color), and the message text, and is passed via the object hierarchy references (refer to Figure 3-10 in §3.2.4.4) from the *ReportElement* object that generated the message to the *System Interface* that maintains the *InterfaceOutputFrame* and *InterfaceOutputPanel* objects (see §4.2.6).

Additionally, in this implementation, when element status is changed (internally by a report element's *verify* method in response to incorrectly formatted user input, or externally by a JESS rule firing), the background color of the offending element's text field is changed to the color corresponding to its status level. See Figure 4-3 for examples of these types of visual feedback mechanisms.

4.2.6 Interface Display

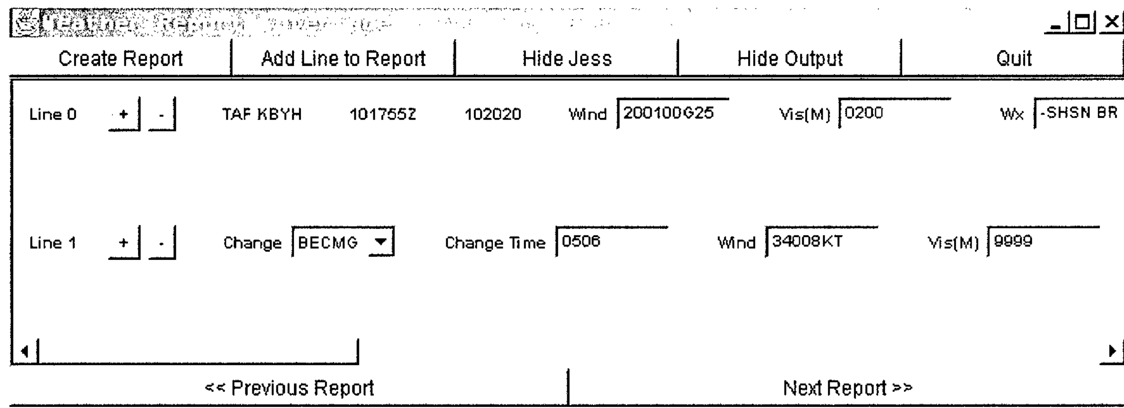
In order to produce rudimentary on-screen feedback to the user, simple display objects have been implemented. Figure 4-3 depicts the presentation of the display interface. As implemented, the interface consists of three separate frames (or windows). The main frame (Figure 4-3a) is the *SystemInterface* object itself—a subclass of Java.awt.Frame. It contains a Java.awt.ScrollPane for report template display. Inserted into the scroll pane is a single Java.awt.Panel instantiated with a Java.awt.CardLayout layout object. Each instantiated template can be individually viewed by iterating through the card layout (see [5] for more information on CardLayout functionality).

[ii] Messages generated by JESS are communicated to and generated by the weather data object (such as the *WeatherCondition* or *WeatherReport*) that caused the rule to fire. JESS accesses the *issueWarning* method of these objects to send the message. The status of individual offending elements can also be changed by JESS by accessing the weather data objects' *setElementStatus* methods. In each case, the *retractWarning* and *retractElementStatus* methods are used to retract the message when input data is corrected.

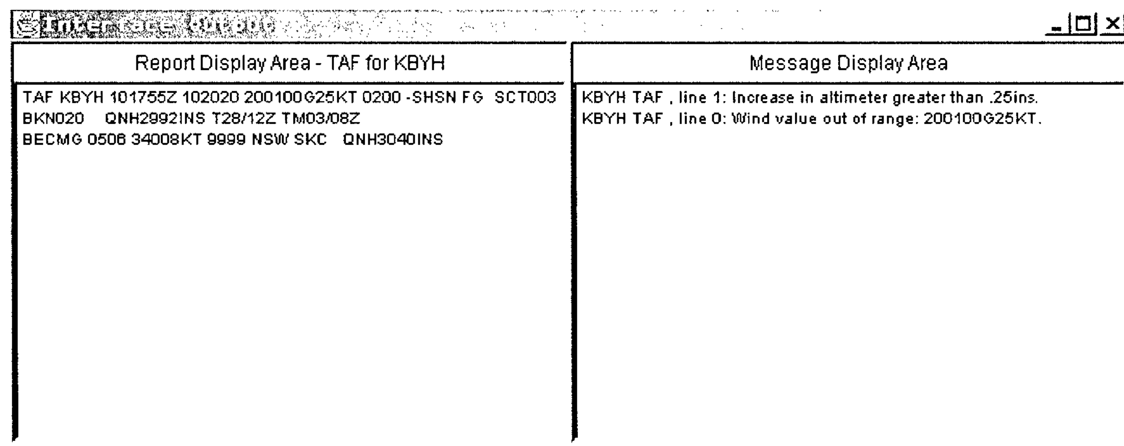
On the top and bottom of the main frame are buttons labeled "Create Report" and "Add Line to Report," as well as other control buttons to hide or display the other frames and flip through any active report templates.

The other two frames are attributes of the *SystemInterface* object, and are instantiated when the *SystemInterface* is initialized. The first of these two frames, the *InterfaceOutputFrame* object (Figure 4-3b), contains a single panel—the *InterfaceOutputPanel* object. This panel (a subclass of Java.awt.Panel) contains two Java.awt.TextArea components. In one text area, a fully formatted version of the currently selected template's report-in-progress is displayed. As data is entered by the user, it is formatted and saved as a text String by the *ReportProperties* object (see the *ReportProperties* object in the Data Dictionary of Appendix C). As updates are made to the template, the *SystemInterface* is notified (again, through the reference hierarchy depicted in Figure 3-10). The *SystemInterface,* in turn, informs the *InterfaceOutputPanel* via an *updateTextOutputArea* method.

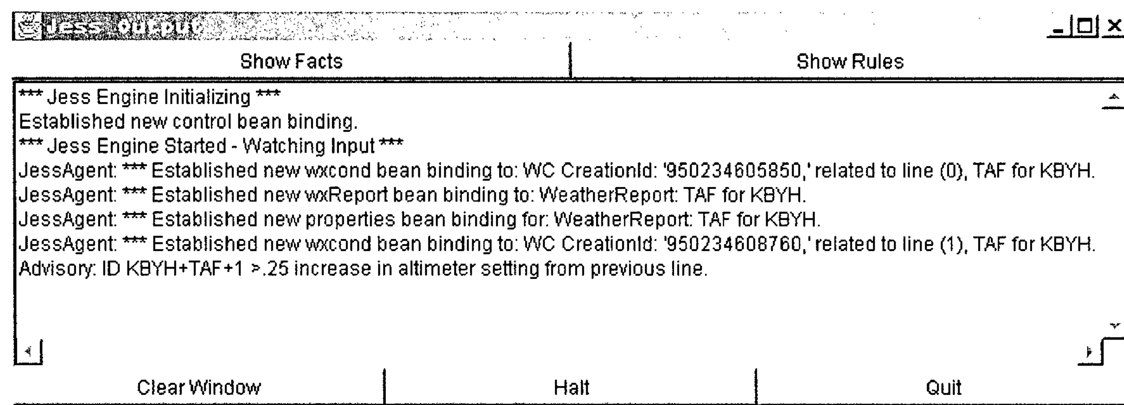
The other text area of the *InterfaceOutputPanel* contains all the messages generated by the elements and rules from all the active reports. For convenience, they are reported alphabetically by station ICAO, then by report type, and then numerically by line number. It is the responsibility of the *InterfaceOutputPanel* object to maintain a set of active *Message* objects, and provide methods for adding and retracting messages by identifier.



(a)



(b)



(c)

Figure 4-3: Screen-capture depicting the display for the user interface

The third frame (Figure 4-2c) is a simplified version of the JESS *ConsoleDisplay.java* class that ships with JESS 4.4 [19]. It contains buttons that trigger the JESS *(facts)* and *(rules)* commands to display any currently active rules and facts. The frame, *JessDisplayFrame* contains a Java.awt.Panel called *JessDisplayPanel*. This panel contains a single Java.awt.TextArea component that continually displays the JESS output stream[iii]. The inclusion of this third frame is simply for system analysis. This third frame would not be included for any operational implementation.

4.2.7 Implemented Functions

The system functions implemented in support of this research are a primarily a critical core, with enough illustrative examples needed to demonstrate proof of concept. Report template functions are "create"—which instantiates a template—and "delete" which removes one from the display. Template line functions include "insert," "delete," and "append" functions. Of course, user interaction with the instantiated template is also supported, and is in fact the basis for the implementation.

The support for runtime template design is simulated by two String arrays in the *TemplateRepository* class that simply list in order, by name, the elements of TAF and METAR observation reports. At instantiation, methods of the Java Class and Constructor classes are employed to search for, call the constructor of, and instantiate these element objects by name at runtime, in the order they appear in the arrays [4]. If the named element objects (subclasses of *ReportElement*) appear in the system package, they are instantiated and added to the template line. If not, the element name is ignored. Any change (deletion,

---

[iii] Another bug identified in Jess version 4.4 and fixed in 5.0 is that the Java.String.StringBuffer object used to hold the output JESS control messages can fill up, and the display will freeze. In this implementation, clicking a provided "Clear Display" button will clear and un-freeze the display. In version 5.0, older display lines are removed from the buffer before it can fill. This problem has a negligible effect on the implementation of this user interface system.

addition, or order change) to these arrayed names effects a change in the corresponding GUI components in the template.

The attribute values of the *ReportProperties* objects for TAFs and Observation reports are stored in two dummy classes, called TAF.java and METAR.java. Before the *ReportManager* directs the construction of a new *WeatherReportTemplate,* it instantiates one of these two objects (depending on whether the user selected TAF or METAR as a report type), as well as a new *ReportProperties* object (refer to §3.2.4.3). It then passes the report property values from the dummy object to the *ReportProperties* object. *BaseWeatherStation* objects are handled in much the same way. They are instantiated by the *SystemInterface,* and passed to the *ReportManager* for inclusion in the *ReportProperties* object. When the *ReportManager* finally instantiates the *WeatherReportTemplate,* it passes the *ReportProperties* object to the template's constructor.

The rationale for this implementation is that since the *ReportProperties* and *BaseWeatherStation* objects are created externally to the system and passed through the *ReportManager* to the *WeatherReportTemplate* at instantiation, it's easy to see how they might just as easily be dynamically constructed and persistently stored via a *ConfigurationEditor* object in a maintenance function as suggested in §3.2.4.1. Though *WeatherCategorySets* are not implemented, they would be handled in a similar manner.

More importantly, since elements are instantiated by name at runtime (as described above), it is easy to see how a *TemplateEditor* could be constructed to design new report templates using the library of report elements in various combinations, without the need to recompile system components.

4.2.8 JESS Rules

JESS rules are defined through the JESS (*defrule*) function, as illustrated in §2.3.3.3. Rules can be defined one at a time or, more conveniently, all at once by

including them in an ASCII text file and issuing a JESS (*batch*) command. The latter option is employed in this implementation.

The batch file name (by convention with extension '.clp,' held over from JESS' roots in CLIPS), is passed as an argument to the system's *main* method that calls the *SystemInterface* constructor. After all of the JESS-related objects are initialized (the *Rete* engine object, the *JessDisplayFrame* and *JessDisplayPanel* output objects discussed earlier in §4.2.6), a batch function call is made using the *Rete* object's *executeCommand* method.

In an operational implementation, a rule editor to define specific rules based on a menu of *WeatherCondition* and *WeatherReport* attributes would be useful. As with report elements, modifications to JESS rules can be accomplished externally, and then batch-fed to JESS without recompiling any system components.

Four rules were implemented initially. A fifth was added after informal analysis of TAF's and METAR observations collected from various stations during times of inclement weather (discussed in §4.3.2). They are:

4. Forecast low temperature higher than forecast high temperature
5. Observed temperature lower than the dewpoint temperature
6. >.25 inch decrease in altimeter setting between subsequent TAF lines
7. >.25 inch increase in altimeter setting between subsequent TAF lines
8. (Added after report analysis) Altimeter setting included in a TEMPO group

The forecast temperature rule (1) acts upon attributes of the *WeatherReport* object, while the other four act upon attributes of the *WeatherCondition* object. The altimeter setting rules (3 and 4) act upon two separate *WeatherCondition* objects that are consecutive in the report.

Note that in the case of all of these rules, the input data must have been formatted properly for these rules to fire. In the case of rules 1, 2, and 5, though

the data format was correct, the type of data or the combination of data types entered caused a violation of the report syntax, and so these rules cause a "red" status warning. In rules 3 and 4, though the format is correct in both altimeter fields, the change in altimeter between report lines is dramatic, and therefore questionable. It is conceivable that a rare weather event such as the passage of a deep low-pressure system could cause such a drastic drop in pressure, so the entry is allowed, but questioned. The text field is highlighted in yellow, and feedback advising the forecaster of a *possible* error is displayed.

Each of the rules asserts a 'warning' fact associated with its corresponding *WeatherCondition* or *WeatherReport* object. This 'warning' fact remains active until the situation is remedied. In this implementation, the only way to retract a message is to change the data input. Of course, in any operational implementation, there should be much more user control over messages, allowing the forecaster to acknowledge and clear advice messages with which he or she disagrees with.

In all cases of forecast advice, though not implemented, the design specification calls for an advice log to record all rules that fired. The log would include such information as the name of the valid time for the report, the rule that fired, whether or not the rule was heeded, the data input, and the actual observed weather at the time the rule fired. In this way, rules may be analyzed for their effectiveness, and adjusted as necessary.

4.2.9 System Process

After examining the individual components of the system, it may be helpful for the reader to take a step back from the details of object interaction to see the overall system process, from template design to weather report dissemination. The steps involved in the process are listed here and elaborated below:

1. Create element objects (subclasses of *ReportElement*).
2. Create and save the new template (using the library of elements).
3. Build any desired rules to aid in report generation (note that this is not a *necessary* step, in that the report template may be designed and invoked without any rules).
4. Invoke a saved template—fill it out and submit it.

As designed, the first step in creating a new report template is to build the *ReportElement* objects that will accept the coded input from the user. As discussed in §4.2.4.2, building the elements could be the function of a programmer or, more effectively, of a separate application designed for that purpose. Regardless of method, the elements must be able to accept, parse, and verify user input, and, if necessary, modify the appropriate attribute(s) of the weather data objects *WeatherCondition* and *WeatherReport* (or subclasses thereof—see §3.2.4.9). Their properties must also be defined to indicate the element's position in the report (first line only, last line only, etc.), whether the element is required in every report, etc.

If, of course, the "library of *ReportElement* objects" already created is sufficient to design the new template, this step can be skipped. Given the elements implemented for a TAF and a METAR observation, for instance, to create a PIREP template it would be necessary to create elements such as *AircraftType*, *AircraftCallSign*, etc. These elements would be registered to modify the attributes of the *ObservedCondition* (subclass of the *UpperAirCondition*) object.

After all the required elements have been constructed, the next step in the process is to design the template. Once again, ideally, this function would be fulfilled by a separate application to perform the task. Essentially all that is involved in the process of template design is listing the order of the elements as they would appear in a report and specifying line and report properties—maximum number of lines, report type, report text formatting, etc. Once this is

accomplished, the information is named and saved. The report is now ready for use.

At any time during or after report template design, rules may also be designed and associated with the template's rule set. The rules should be written based on the attributes of the weather data objects the template elements will modify. (If they aren't, the report will still work, but the rules will never fire since the attributes upon which they are based will never change). The rule sets specific to one type of report should also be named and saved.

Once these tasks are accomplished, the report template is ready for use. In the interface application, the user simply requests the new report type by name. The template loads and its associated weather data objects are automatically created and registered with JESS. As the user enters data, it will be automatically QC'd based on the *verify()* methods of the individual elements and any JESS rules associated with the attributes they modify.

4.3 Implementation Testing and Analysis

After partially implementing and using the interface, it became clear that a system meeting the requirements and providing the capabilities described in the previous chapters can be constructed. This section sums up the capabilities realized in the implementation of such a system.

Additionally, to get some sense of the operational worth of such a system, a series of TAFs and METAR observation reports from AFW reporting stations was collected and analyzed to determine what effect, if any, a system such as the one presented here would have if fielded operationally. The results of this analysis are also addressed here.

Finally, issues concerning system integration with existing AFW systems are discussed.

4.3.1 System Analysis

As discussed in the previous sections regarding the individual system components, the system implementation has shown that a completely dynamic system could be built such that:

(a) Individual user interface elements can be designed at runtime through the use of an element editor application. (Refer to §4.2.4.2).

(b) Weather report templates can be designed, created, and stored with an appropriate editor tool and a library of report elements. (Elements could either be constructed with the application mentioned in (a) or coded and compiled prior to use).

(c) Designed and stored templates can be invoked by a forecaster or observer and automatically perform format QC based on the data input into the individual interface elements.

(d) Rules specific to any designed and stored template can be written by a meteorologist to provide guidance to the forecaster during report generation.

(e) Rules can continue to be applied after the forecast has been disseminated, effectively providing automated verification. Beyond advising the forecaster when a report requires amendment (after the fact), however, rules can be applied to advise of negative trends to catch and amend forecasts "before the fact" (see §1.3.4)

(f) New rules can be written and fielded, new report elements can be designed and fielded, and new report templates can be designed and fielded, *without interrupting program execution.*

In all, the implementation demonstrated that the system, as designed, provides the capabilities described in §1.3. If fielded operationally, it would

provide the services necessary to meet the requirements of the conceptual report generation process proposed in §1.3.5.

Furthermore, since the data input is QC'd and stored in readily accessible and conveniently packaged *WeatherCondition* and *WeatherReport* objects, the system is capable of easily exporting this data for use in analytical processing in support of an automated metrics program or other analysis or research.

4.3.2 AFW TAF and METAR Observation Analysis

Over a period of seven (non-consecutive) days, TAF and METAR observation reports were collected from AFW reporting stations (see the data in Table 4-1). Stations were selected by observing the National Weather Service's national Doppler (WSR-88D) composite radar product, and picking stations that appeared to be experiencing inclement weather[iv]. Once the stations were chosen, TAF and METAR reports were collected for a 48-hour period encompassing the inclement weather.

On seven different days (one in January and six in February, TAF and METAR observation data from a total of 10 48-hour periods from seven stations were collected. If data were unavailable for either TAFs or METAR observations, the report type that *was* available was still used.

[iv] During periods of inclement weather, the duty forecaster and observer are busy and prone to make more errors.

Station	Report	Date	Error	Type
KFFO	TAF	19-JAN-00/19Z	Invalid wind format (2010G18KT)	X
KFFO	METAR	19-JAN-00/20Z	Wrong sea-level pressure value	X
KFFO	TAF	20-JAN-00/05Z	Unknown	
KFFO	TAF	20-JAN-00/13Z	Invalid cloud group (0VC008 – used 'zero' instead of letter 'O')	X
KFFO	METAR	20-JAN-00/13Z	Change in summary data	S
KFFO	METAR	20-JAN-00/1509Z	Change in summary data	S
KFFO	METAR	20-JAN-00/16Z	Change in summary data	S
KFFO	METAR	20-JAN-00/1735Z	Unknown	
KFFO	TAF	04-FEB-00/01Z	Unknown	
KFFO	TAF	04-FEB-00/07Z	High-low temp/time (T02/01Z TM03/12Z) COR to: TM02/12Z T02/20Z	?
KFFO	TAF	04-FEB-00/07Z	Unknown	
KFFO	TAF	04-FEB-00/13Z	Inconsistent icing/Turbulence Groups	X
KFFO	TAF	04-FEB-00/13Z	Unknown	
KFFO	TAF	04-FEB-00/16Z	Unknown	
KFFO	TAF	04-FEB-00/19Z	Missing high/low temperatures	X
KFFO	METAR	05-FEB-00/12Z	Omitted summary data (60000)	S
KOFF	METAR	04-FEB-00/1221Z	Missing gusts (Non-format)	
KOFF	TAF	04-FEB-00/18Z	**Altimeter included in TEMPO group**	X
KBLV	METAR	04-FEB-00/18Z	Unknown	
KBLV	METAR	05-FEB-00/00Z	Unknown	
KBLV	TAF	05-FEB-00/10Z	**Altimeter included in TEMPO group**	X
KSKA	METAR	06-FEB-00/07Z	Invalid cloud format (SCT0095)	X
KBIX	METAR	07-FEB-00/07Z	Change in summary data	S
KBIX	METAR	07-FEB-00/1013Z	BR code (vice FG) used for fog with visibility of 1/2SM	X
KBIX	METAR	08-FEB-00/1025Z	Change in summary data	S
KBIX	METAR	08-FEB-00/1059Z	Change in summary data	S
KMIB	METAR	08-FEB-00/06Z	Change in summary data	S
KMIB	METAR	08-FEB-00/1210Z	Unknown	
KMIB	METAR	08-FEB-00/20Z	Invalid wind gusts (29023G24KT)	X
KMIB	METAR	09-FEB-00/1248Z	Omitted runway remarks during snow event	S
KMIB	METAR	09-FEB-00/17Z	BKN ceiling changed to OVC (Non-format)	
KSKA	METAR	08-FEB-00/0649Z	Missing RVR information during fog event	X
KSKA	METAR	08-FEB-00/0652Z	Missing RVR information during fog event	X
KSKA	TAF	08-FEB-00/07Z	Change in visibility forecast (Non-format)	
KSKA	METAR	08-FEB-00/0922Z	Missing wet runway remark during fog event (no recent precipitation – Non-format)	
KSKA	METAR	08-FEB-00/13Z	Unknown	
KSKA	METAR	08-FEB-00/15Z	Missing wet runway remark during rain event	S
KSUU	METAR	09-FEB-00/00Z	Omitted summary data (60000)	S
KSUU	METAR	09-FEB-00/01Z	Change in summary data	S
KSUU	METAR	09-FEB-00/02Z	Invalid cloud format (BKN20)	X
KSUU	TAF	09-FEB-00/23Z	Rime icing forecast below level of clouds	X
KSUU	METAR	09-FEB-00/2342Z	Wind variable remark changed to VRB wind group	X
KSUU	METAR	10-FEB-00/12Z	Missing wet runway remark during rain event	S
KSUU	METAR	10-FEB-00/1625Z	Invalid cloud group (BKN0640)	X
KSUU	TAF	10-FEB-00/1845Z	Unknown	

Table 4-1: TAF and METAR observation report corrections

In each 48-hour period, reports with COR modifiers were extracted along with the preceding report to determine the cause for the correction, if available. In the table, the alternating shaded and un-shaded fields delineate the 48-hour period boundaries. There is "unevenness" in the time blocks because, as noted

above, the data was not collected based on time, but rather on inclement weather. So, for instance, in the case of Travis AFB (KSUU), data was collected from 8-Feb-00/02Z through 10-FEB-00/02Z, and then again from 10-Feb-00/02Z through 12-Feb-00/02Z. Only instances when corrections were issued are reported in the table.

Upon analysis of the collected data, it is worth noting first of all that in *every* 48-hour period of reports collected, there was at least one corrected report in either the collection of TAFs or the collection of METAR observations.

Table 4-2 summarizes the TAF corrections collected. Of the 8 corrections issued for known reasons, 6 were issued because of format errors that could have been caught with a system such as the one specified in Chapter 3. (Some of them, in fact, would have been caught by the limited demonstration implementation described in this chapter).

Error Type	Number
Bad Format	6
Questionable	1
Non-Format	1
Unknown Errors	6
Total TAF CORs	**15**
TAF Errors/48hrs	**1.5**

Table 4-2: TAF Correction Totals

One particular error the demonstration implementation did *not* account for (and occurred twice in this data collection) was the disallowed occurrence of an altimeter setting in a temporary weather condition (TEMPO) group (bold text in Table 4-1). Since both of these elements—the *Altimeter* and *LineChangeModifier*—were implemented for the demonstration, it seemed that a rule could be applied that would account for this error. In fact, it took less than five minutes to write and implement a rule that would identify the error, notify the forecaster of the problem, and prevent the submission of reports that contain the error. If the system had been fielded operationally, the addition of this rule

would have effectively eliminated the possibility of that type of error ever occurring again. Furthermore, to add this rule, only the ASCII rules batch file was modified—no Java code or object compilation whatsoever was required.

Similar rules could be developed to ensure cloud layers exist where rime icing is forecast, that icing and turbulence groups are consistent throughout a forecast period, etc.

One particular "questionable" error (marked with a question mark in Table 4-1) was a mix-up between high and low temperature times. In this instance, had a critic system been employed to advise the forecaster, if the input a low-temperature had been higher than the high-temperature, the forecast would have been disallowed. On the other hand, if the low temperature had been forecast to occur in the afternoon and the high temperature to occur in the morning, through the use of a rule, the forecaster could have been alerted to ensure it was not a mistake (as it would have been in this case). Either way, the forecaster would have been given feedback so that his or her attention would have been focused on the suspect data, and most likely this error would not have been issued.

Table 4-3 lists the summary data for METAR observations. Of the data collected, more than half of the known errors were for incorrect 3, 6, 12, or 24-hour summary weather data (labeled as type "S" in Table 4-1). Because of the explicit rules defining the format of METAR summary data (rain and snowfall totals, cloud types, cloud amounts, runway conditions, pressure changes, etc—see [8] for more information on surface observation coding), default values for much of these code groups could have been generated automatically. For instance, rules based on earlier-reported rain amounts or on the presence of rain in the current or previous observations could advise the observer of a missing rain amount group if, by METAR standards, the group were required.

Similar rules could be written to advise for the need of such code groups as a "wet runway" code when it's raining, "snow-depth" codes when it's snowing, and to ensure that no cloud types or amounts were reported above a low overcast deck. In fact, in every case concerning METAR summary information, rules could be devised that would disallow contradictory input and warn when the observer may have overlooked required input.

The rest of the errors—invalid code groups, using the "BR" code instead of the "FG" code for fog with reported visibility less than ½ statute mile, etc., are purely format errors, and could be automatically caught via the individual report elements' *verify* method. Note that "wrong sea-level pressure" is considered to be a format error—in fact, since the sea-level pressure is a value computed based on a mathematical formula (involving station pressure, temperature and station elevation), it could be calculated and formatted into the report automatically.

Error Type	**Number**
Bad Format	12
Summary Data	13
Unknown Errors	5
Total METAR CORs	**30**
TAF Errors/48hrs	**3**

Table 4-3: METAR Observation Correction Totals

Though an extremely small number of reports was collected in this set, an analysis of the results makes it is easy to see that an operationally fielded implementation of the design specification presented in Chapter 3 would virtually eliminate the need for TAF and METAR observation corrections due to format errors. Furthermore, it would considerably reduce the occurrence of other common errors by calling the forecasters' and observers' attention to questionable or contradictory input.

A final note regarding this analysis: The error totals given here are for errors the forecaster or observer *caught* and for which they issued a correction.

There is no practical way to determine how many other errors go unnoticed or are not caught in time to issue a correction. Though these reports were intentionally taken from stations experiencing inclement weather, and weather personnel were busy, the error rates encountered are significant. Consequentially, any post-production analysis conducted on such data would be at risk of drawing faulty conclusions.

4.3.3 AFW System Integration

As the ongoing reengineering effort continues throughout Air Force Weather, and forecasting responsibility is assumed by regional Operational Weather Squadrons (OWSs), forecasters will be responsible for generating reports for multiple stations.

The current product generation interface component of the OWS Production System (OPS) introduced in §2.4.3.2 is the Advanced Meteorological Information System (AMIS) described in §1.2.1.2. This interface provides GUI input templates for TAFs, METAR observations, Pilot Reports (PIREPs), and a number of other report types. As mentioned, AMIS is more than just a user interface—it also provides a mechanism to securely disseminate reports. As mentioned, however, the user interface front-end to AMIS provides no QC of outgoing products. Furthermore, the report information collected via AMIS is simply collected, formatted, and concatenated into an ASCII character string. The individual data elements associated with the user's input are not collected. In the Wind field example given in §3.2.3.1, for example, the wind group is collected and stored in the form in which the user entered it. The data this METAR-coded field represents—namely the wind direction, wind speed, and wind gusts speed—is ignored.

Given these circumstances, a continued positive TAF report error rate—especially during periods of inclement weather—is inevitable. And given the

forecasters' increased responsibility to produce multiple base forecasts from regional OWSs, it is likely to increase.

Not only that, but any metrics program or other data analysis effort will still require report parsing as an initial (and typically problematic) step. As it stands, therefore, the same problems that plagued the cancelled TAFVER programs (§2.4.1) will persist: data will be incomplete and/or inaccurate, and any conclusions drawn from the data analysis will be unreliable.

If an interface designed in accordance with the specifications detailed in Chapter 3 (and Appendices C and D) were to replace AMIS' GUI interface front end, these problems would be alleviated. Reports generated using the interface would be all but error free, and reliable data could easily be exported for ingestion into any post-production analytical processing program, including automated metrics.

4.4 Summary

In this chapter, a partial implementation of the user interface design presented in Chapter 3 is discussed. Through implementation of specific pieces of the design, it is readily apparent that, if implemented, the design specification achieves the goals and fulfills the requirements set forth in previous Chapters. Namely, a system such as this will not only ensure that weather reports are formatted correctly, but it will use its weather domain knowledge base to advise the forecaster or observer of possible enhancements to his or her product *before* it is submitted. Through an analysis of a random collection of TAFs and METAR observations, it was shown how such a system could virtually eliminate format errors in weather reports.

Chapter 5 summarizes this research effort, highlighting the operational benefits of fielding a system such as the one proposed. It is suggested that a critic system could not only improve forecast accuracy, but could even serve to

better forecasters' ability. Furthermore, Chapter 5 discusses how the implementation of such a system provides the opportunity for research in several different areas, all of which could directly lead to higher quality, more accurate weather products.

5 Summary and Conclusions

5.1 Overview

If Air Force Weather (AFW) is to implement a successful automated metrics program (or any post-production weather data analysis program, for that matter), it is imperative that the raw data be error free [27]. This research sought to explore user interface options to solve the problem of inconsistent, incomplete, and erroneous weather data in weather products. In Chapter 3, a design specification solution to the problem is presented, and in Chapter 4 it is shown that such a system would eliminate the need for report corrections due to format errors and dramatically reduce occurrences of other types of errors.

In addition to these immediate advantages, other less obvious yet equally important benefits could be realized through implementation of a system such as the one proposed in this research. This Chapter highlights some of those benefits if such a system were implemented operationally.

5.2 System Benefits

Though a near-zero AFW weather report error rate is the most obvious and immediate benefit of the operational implementation of a critic system, the flexibility inherent in the design provides opportunities for data analysis and research that would not otherwise be feasible. Furthermore, the nature of a critic system such as the one proposed offers the possibility of other benefits as well, such as improved forecast accuracy and improved forecaster ability. Additionally, it offers easy access to error-free weather data for research and analysis, and a convenient platform for testing, analyzing, and fielding the results of research involving correlations in weather data. This section details these and other benefits of such a system.

5.2.1 Improved Forecast Accuracy

Through the development of rules based on weather data from a variety of sources (surface observations, upper air observations, climatology, model data, etc.), this system could provide forecasters with advice based on the past experience of weather-domain knowledge experts. The advice would offer the forecaster "educated alternatives" to his or her forecast.

Because of the dynamic nature of the weather, even after implementing and tuning these rules, forecast amendments would not become obsolete. However, many generalizations and cross-parameter correlations have been tested over the years and proven to produce positive results. In fact, the field of Numerical Weather Prediction (NWP) is dedicated to employing such correlations on a broad scale for the production of models.

One industrious weather student, George Elliot, researched and published many of these guidelines and procedures for using previous and current surface and upper air observation parameters to predict high and low temperatures, precipitation events, fog, and other weather events [18]. Each of these procedures is based on a correlation that had been either formally or informally found to produce positive results.

Given a critic system such as the one presented here, tried-and-true correlations such as the ones in Elliot's book could easily be implemented as rules that would serve to advise the forecaster of the possibility of weather phenomena he or she might not have considered. And using an advice log (§3.2.4.10), these rules could be analyzed and tuned for best performance.

It is impossible to precisely determine the impact a critic system would have on forecast accuracy without building and fielding a viable system and studying the weather products produced with it. Past research has indicated, however, that critics have improved the quality products in other complex domains where optimal solutions are not readily apparent (see §2.2.5). It is

logical to assume, therefore, that forecasters (and observers) would benefit from their use, and that product quality and accuracy would improve.

5.2.2 Improved Forecaster Ability

As with a presumed improvement in forecast accuracy, it is impossible to prove that a critic advice system would improve forecasters' ability to forecast without operationally fielding one and analyzing its impact. It has been shown, however, that critic systems providing useful advice serve to educate the user as well (§2.2.5.4).

The fact that this happens is not difficult to imagine. When presented with a solution to a problem given a particular set of circumstances, it is human nature to recall the solution if the same set of circumstances arises again.

In a familiar example, people who use Microsoft Word's "auto-correct" feature (that highlights and corrects misspelled words) are, perhaps, more likely to spell these word correctly the next time they have to type them.[i]

In a critic system, alternative solutions to portions of a particular problem are provided based on certain identified circumstances. It is not unreasonable to assume that, should the advice be correct, the next time the user faced similar circumstances, he or she would recognize the solution to the problem based on his or her previous experience.

Unlike the design of kitchens (as in the *Janus* critic example presented in §2.2.5), weather patterns are chaotic and defined by so many variables that it is quite possible that even though logic might be successfully applied today, tomorrow the same reasoning could yield unsatisfactory results. Even so, there is enough consistency in weather patterns to develop sound reasoning that can be taught and applied with positive results.

[i] Though no formal study of this tendency was conducted for this research, this is certainly consistent with the experience of *this* author in drafting this document.

Much of this reasoning involves procedural steps and the analysis of correlations between various weather data, such as described in §5.2.1. If the rules of a critic system were tuned to provide reliable advice based on proven weather correlations, when the forecasters using the system were given this advice, they would inevitably learn from their experiences.

5.2.3 Operational Use of Laboratory Research

Correlations in weather data that are found during academic research, though useful as a stepping-stone to a grander solution, are sometimes difficult to immediately put to operational use. One such research effort (concurrent with this research) involves the use of various surface weather and upper air parameters as a fog formation predictor [34].

Though research of this type is extremely useful in areas such as the development of computer models, a critic system gives an opportunity for this sort of correlative data to be used operationally, during forecast report generation. If the research indicated that such a correlation of parameters was strong, for example, in an effort such as the fog predictor described above, a JESS rule could be written, and such information relayed to the forecaster had he or she not considered fog in the forecast.

5.2.4 Analysis of Research Results

The argument presented in the preceding section can be taken a step further. If the results of such research were condensed into a JESS rule and used operationally, when weather parameters matching the rule pattern occurred, the rule would fire, and the forecaster would be advised of the correlation. In those cases, the forecaster may or may not have heeded the advice of the system, and the system may or may not have been right. Regardless, the circumstances of the rule's firing would have been recorded in the advice log.

Over time, when enough data had been collected, the advice log could be analyzed to determine how the rule was performing. This would provide the unique ability for researchers to reexamine the results of specific research efforts and fine-tune the applicable correlative parameters. This cycle of research and operational use of the results could continue in regard to all the rules applied to the system. It would greatly facilitate researchers' data collection efforts and, in the process, improve the quality of operational AFW forecast products.

5.3 Future Research

After conducting research into the application of a critic system to the weather product generation problem, it is clear that other research efforts involving the fields of Computer Science and Meteorology would be beneficial. This section elaborates some areas for potential study.

5.3.1 Critic Research

Critics have been widely studied and proven to be beneficial to the application of various complex problems (§2.2.5). Though the present research applies a critic to a specific domain, it would be useful to research the potential of a *generically* defined intelligent user interface application.

Though report generation and formatting is a by-product of the interface applied to the weather problem in this research, the primary goal is the *quality* and *accuracy* of the report, rather than production and dissemination of the report itself. The quality and accuracy of the report are realized through the application of a specific validation process to ensure data is correct (format QC) and an interactive, collaborative effort between the system and the user to produce a quality product (content QC).

In the design presented in Chapter 3, the interaction between user and system to realize this collaborative effort is *domain specific*. Even the objects that

communicate with the expert system shell themselves—the *WeatherCondition* and *WeatherReport*—are designed for weather data processing.

In the conduct of various research and operational activities, it would be useful to employ a "critic shell," much in the same way it was useful to this research to employ an "expert system shell." The "critic-shell" would provide a core library of interface components designed to interact with an expert system shell such as JESS, but be domain independent, such that domain-specific features of the system (parameter names and values, rule definitions, format criteria, etc.) could be easily defined and implemented.

Such a powerful and flexible system would have great potential impact when applied to problems with complex domains such as weather forecasting, stock-market analysis, complex scheduling problems, and others.

5.3.2 Meteorological Research

In §5.2.3 and §5.2.4, the potential for weather-domain correlative analysis research given a critic system is limitless. Because in the design presented in Chapter 3, rules can be applied, modified, and reapplied without mission impact (see, for example, the implementation of the new rule in §4.3.2), it would be easy to use them operationally. Furthermore, with the inclusion of an advice log that records instances of and circumstances surrounding rule firings (to include rules fired during report generation as well as during post-production verification), precise data indicating rule performance (and therefore an operational measure of weather parameter correlations) is readily available.

Additionally, because of the strict automated quality control measures employed during product generation, the raw data—not just the text report output, but the weather data itself—is accurately stored and available for processing [27].

This wealth of data is a potential lode of correlative data never before accessible in the weather field. Properly extracted and archived, it could give rise to uncountable research efforts into a variety of aspects of meteorology, such as climatology, synoptic meteorology, and numerical weather prediction. And as pointed out in §5.2.4, the results of such studies could be applied operationally, and would immediately benefit forecasters and improve forecast quality—and, as mentioned, pave the way for *more* future, related research.

5.3.3 Other Areas for Research

In this user interface, every piece of information a user enters is parsed and analyzed for correctness of format. Though the dissemination of products with typographical and other format errors would not be permitted in such a system, the occurrences of such errors could potentially be recorded.

With the other types of data collected under the design of Chapter 3—"clean" raw data and the results of rule firings—further data mining research could be conduct to examine correlations between these data and intermediate format errors.

Along the same lines, correlations between format errors and physical interface configuration could be conducted, to determine more efficient interface components and layouts. This type of research is already conducted in the Computer-Human Interaction (CHI) arena, and the data produced by a system such as the one proposed would offer new avenues to explore.

5.4 Summary

The automated measurement of Air Force Weather product accuracy as well as the application of other data analysis efforts *is* an attainable goal. For that goal to be realized, however, steps must be taken to ensure the data used by such efforts is reliable, consistent, and accurate.

This research has shown that the application of an intelligent user interface "critic" system would not only correct format errors on the spot, but also advise forecasters and observers of potential enhancements to report accuracy. The system, effectively a metrics program applied before the weather products have been submitted, has the potential to eliminate format errors from future reports, dramatically reduce the instances of other errors, improve the meteorological content quality (and therefore accuracy) of weather products, and even serve as a teaching tool for forecasters and observers alike. It has been shown how the results of system-produced data could be studied and reapplied, to further improve the quality of Air Force Weather products.

Appendix A – METAR Coding Example/Explanation

The following sample forecast and observation reports were taken from the National Weather Service home page [7]. Each group of the reports is extracted and elaborated. Note that there are some key differences between military and civilian code format—for instance, among other things, the Air Force doesn't indicate the probability of weather phenomenon with a 'PROB' group appended to a line. The concept for the code is the same, however, and serves to illustrate the kind of typographical rules a forecaster or observer must follow to properly format a weather report.

KEY to AERODROME FORECAST (TAF) and AVIATION ROUTINE WEATHER REPORT (METAR)

```
TAF KPIT 091730Z 091818 15005KT 5SM HZ FEW020 WS010/31022KT
     FM1930 30015G25KT 3SM SHRA OVC015 TEMPO 2022 1/2SM +TSRA
      OVC008CB
     FM0100 27008KT 5SM SHRA BKN020 OVC040 PROB40 0407 1SM -RA BR
     FM1015 18005KT 6SM -SHRA OVC020 BECMG 1315 P6SM NSW SKC

METAR KPIT 091955Z COR 22015G25KT 3/4SM R28L/2600FT TSRA OVC010CB
                 18/16 A2992 RMK SLP045 T01820159
```

Forecast	Explanation	Report
TAF	Message type: **TAF**-routine or **TAF AMD**-amended forecast, **METAR**-hourly, **SPECI**-special or **TESTM**-non-commissioned ASOS report	**METAR**
KPIT	ICAO location indicator	**KPIT**
091730Z	Issuance time: ALL times in UTC "**Z**", 2-digit date, 4-digit time	**091955Z**
091818	Valid period: 2-digit date, 2-digit beginning, 2-digit ending times	

	In U.S. **METAR**: **COR**rected ob; or **AUTO**mated ob for automated report with no human intervention; omitted when observer logs on	**COR**
15005KT	Wind: 3 digit true-north direction, nearest 10 degrees (or **V**a**R**ia**B**le); next 2-3 digits for speed and unit, **KT** (KMH or MPS); as needed, **G**ust and maximum speed; 00000KT for calm; for **METAR**, if direction varies 60 degrees or more, **V**ariability appended, e.g. 180**V**260	**22015G25KT**
5SM	Prevailing visibility: in U.S., **S**tatute **M**iles & fractions; above 6 miles in **TAF** **P**lus**6SM**. (Or, 4-digit minimum visibility in meters and as required, lowest value with direction)	**3/4SM**
	Runway Visual Range: **R**; 2-digit runway designator **L**eft, **C**enter, or **R**ight as needed; "**/**"; **M**inus or **P**lus in U.S, 4-digit value, **F**ee**T** in U.S. (usually meters elsewhere); 4-digit value **V**arialbility 4-digit value (and tendency **D**own, **U**p or **N**o change)	**R28L/2600FT**
HZ	Significant present, forecast and recent weather: see table (below)	**TSRA**
FEW020	Cloud amount, height and type: **SK**y **C**lear 0/8, **FEW** >0/8-2/8, **SC**a**T**tered 3/8-4/8, **B**ro**K**e**N** 5/8-7/8, **OV**er**C**ast 8/8; 3-digit height in hundreds of ft; **T**owering **CU**mulus or **C**umulonim**B**us in **METAR**; in **TAF**, only **CB**. **V**ertical **V**isibility for obscured sky and height "VV004". More than 1 layer may be reported or forecast. In automated **METAR** reports only, **CL**ea**R** for "clear below 12,000 feet"	**OVC010CB**
	Temperature: degrees Celsius; first 2 digits, temperature "**/**" last 2 digits, dew-point temperature; **M**inus for below zero, e.g., M06	**18/16**

	Altimeter setting: indicator and 4 digits; in U.S., **A**-inches and hundredths; (**Q**-hectoPascals, e.g. Q1013)	**A2992**
WS010/31022KT	In U.S. **TAF**, non-convective low-level (<=2,000 ft)**W**ind **S**hear; 3-digit height (hundreds of ft); "/", 3-digit wind direction and 2-3 digit wind speed above the indicated height, and unit, **KT**	
	In **METAR**, **R**e**M**ar**K** indicator & remarks. For example:**S**ea-**L**evel **P**ressure in hectoPascals & tenths, as shown: 1004.5 hPa; **T**emp/dew-point in tenths °C, as shown: temp 18.2°C, dew-point 15.9°C	**RMK SLP045 T01820159**
FM1930	**F**ro**M** and 2-digit hour and 2-digit minute **beginning** time: indicates significant change. Each FM starts on a new line, indented 5 spaces.	
TEMPO 2022	**TEMPO**rary: changes expected for < 1 hour and in total, < half of 2-digit hour **beginning** and 2-digit hour **ending** time period	
PROB40 0407	**PROB**ability and 2-digit percent (30 or 40): probable condition during 2-digit hour **beginning** and 2-digit hour **ending** time period	
BECMG 1315	**BEC**o**M**in**G**: change expected during 2-digit hour **beginning** and 2-digit hour **ending** time period	

The following describes the various codes allowed that indicate present, forecast or recent weather. In the table above, the forecast weather code HZ means haze, as defined below. In the absence of any significant weather to report, the acronym NSW (No Significant Weather) is used. Refer to the forecast and observations above for the use of these codes and the intensity qualifiers. For instance, in the second line of the sample TAF, +TSRA means heavy thunderstorms with rain showers, and in the second line of the TAF, -RA BR means light rain and mist.

The following information, extracted from [7] has been elaborated, and usage examples have been provided.

QUALIFIER
Intensity or Proximity.

Qualifier	Meaning	Example
- (minus)	Light	-RA means light rain
"no sign"	Moderate	SHRA means moderate rain showers
+ (plus)	Heavy	+SHSN means heavy snow showers
VC	Vicinity	VCTS means vicinity thunderstorms

Vicinity: In U.S. **METAR**, between 5 and 10SM of the point(s) of observation; in U.S. **TAF**, 5 to 10SM from center of runway complex. (Elsewhere within 8000m)

Descriptor. Only one descriptor may be used with a weather phenomena code, but more than one weather phenomena code can be matched with a descriptor. Some descriptors can't be matched with certain weather codes. (For instance, you can't have BLDZ–blowing drizzle makes no sense.

Descriptor	Meaning	Example
MI	Shallow	MIFG (Shallow fog)
BC	Patches	BCBR (Patchy mist)
PR	Partial	PRFU (Partial obscuration due to smoke)
TS	Thunderstorm	TSRA (Thunderstorms with rain showers)
BL	Blowing	BLSN (Blowing snow)
SH	Showers	SHRA (Rain showers)
DR	Drifting	DRSN (Drifting snow)
FZ	Freezing	FZDZ (Freezing drizzle)

WEATHER PHENOMENA. Indicates type(s) of weather occurring.
Precipitation

Code	Meaning
DZ	Drizzle
RA	Rain
SN	Snow
SG	Snow grains
IC	Ice crystals
PE	Ice pellets
GR	Hail
GS	Small hail/snow pellets
UP	Unknown precipitation in automated observations

Obscuration

Code	Meaning
BR	Mist(>= 5/8SM)
FG	Fog(< 5/8SM)
FU	Smoke
VA	Volcanic Ash
SA	Sand
HZ	Haze
PY	Spray
DU	Widespread dust

Other Code	Meaning
SQ	Squall
SS	Sandstorm
DS	Duststorm
PO	Well developed dust/sand whirls
FC	Funnel cloud
+FC	tornado/waterspout

A more complete set of rules for the METAR code can be found in [7 and 8].

Appendix B – JESS Rules Implementation

The batch rule file implemented in support of this research is presented here. Note that at the top of the file, the JESS (*defclass*) command is used to give JESS visibility to object instances of those types of classes. The full package name associated with each class type is provided and associated with a variable "handle" (for instance, "wxcond" is used to refer to the jess.examples.interface.WeatherCondition object).

The one and only fact defined in this file is the "idle-fact." This fact will be asserted and retracted (see the "sleep-if-bored" rule at the bottom of the file) when the system is idle. This rule (and fact) is here for two reasons. First, we always want to provide a rule that will match some pattern and fire. If no rules match any patterns, the system will halt. Since during product generation rules may or may not match any patterns, we must provide a "default rule" that keeps JESS working. Second, we allow JESS to sleep during idle time so that processor time is not uselessly wasted asserting and retracting idle facts more than necessary. There is a trade off between system response time and the amount of time the system is allowed to sleep. Since user input of data is slow compared to the computational power of a processor, the sleep time is set to the relatively high value of 5000 milliseconds. If this JESS engine were interacting solely with another computer process, the idle time could be reduced in favor of response time.

All other facts are established when instances of the *WeatherCondition*, *WeatherReport*, *ReportProperties* and *Control* objects are registered with JESS. These facts are used in pattern matching for the rest of the rules.

Each of the rules asserts a particular "warning" fact (based on the rule name and an identifier). Note that each of these rules has a counterpart "retract-

warning" fact that retracts any messages and resets element status if the attribute values are corrected.

Finally, the system control rules at the end of the file demonstrate how alternately setting and resetting JavaBean attributes (the bean is defined as "control" in this file) can allow external manipulation of JESS commands while the engine is running. Changing the attributes of the control bean effectively halts the JESS engine, or issues a JESS "(*rules*)" or "(*facts*)" command.

The batch file follows. Comments in the file are delimited by semi-colons.

```
;; -*- clips -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Register Java classes for matching like deftemplates.
;; Bean-like properties become slots. Classes must support
;; addPropertyChangeListener. First argument is the 'deftemplate name'.

(defclass wxcond jess.examples.interface.WeatherCondition)
(defclass wxReport jess.examples.interface.WeatherReport)
(defclass control jess.examples.interface.ExternalControl)
(defclass props jess.examples.interface.ReportProperties)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This fact will be used to sleep when idle

(deffacts idle-fact
  (idle))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Observation-only rules
;; Note that calls to WeatherCondition object method can be made
;; directly from JESS as in 'setElementStatus' and 'issueWarning'

(defrule high-dp
  "Dewpoint higher than temperature - severity red"
  (wxcond (id ?id) (intInitVal ?initVal) (tempC ?tempC) (dewpointC
?dewpointC)(myWeatherReport ?report)(OBJECT ?wxcond))
  (props (baseId ?icao)(reportType ?type)(OBJECT ?props))
  (test (eq ?props (?report getMyProperties)))
  (test (<> ?initVal ?tempC))
  (test (<> ?initVal ?dewpointC))
  (not (warning dp ?id))
  (test (< ?tempC ?dewpointC))
  =>
  (assert (warning high-dp ?id))
  (printout t "WARNING: ID " ?id  " dewpoint higher than temp." crlf)
  (?wxcond issueWarning "Dewpoint higher than temp." "red" "high-dp")
  (?wxcond setElementStatus "dewpointC" "red")
  (?wxcond setElementStatus "tempC" "red"))

(defrule dp-reset
  "Dewpoint no longer higher than temperature - cancel severity red"
  ?warning <- (warning high-dp ?id)
  (wxcond (id ?id) (tempC ?tempC) (dewpointC ?dewpointC) (OBJECT
?wxcond))
  (test (>= ?tempC ?dewpointC))
  =>
  (printout t "Notification: " ?id " dewpoint/temperature now
acceptable." crlf)
  (?wxcond retractElementStatus "dewpointC" "red")
  (?wxcond retractElementStatus "tempC" "red")
  (?wxcond issueRetraction "Dewpoint/temperature now acceptable." "red"
"high-dp")
  (retract ?warning))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Forecast-only rules

(defrule temp-mismatch
  "Low temp higher than high temp"
  (wxReport (id ?id) (intInitVal ?initVal) (reportHighTempC ?highTempC)
(reportLowTempC ?lowTempC)(OBJECT ?myReport))
  (test (<> ?initVal ?lowTempC))
  (test (<> ?initVal ?highTempC))
  (not (warning hi-low-temp ?id))
  (test (< ?highTempC ?lowTempC))
  =>
  (assert (warning hi-low-temp ?id))
  (printout t "WARNING: ID " ?id " low temperature higher than high
temperature." crlf)
  (?myReport issueWarning "Low temp higher than high temp." "red"
"temp-mismatch")
  (?myReport setElementStatus "reportHighTempC" "red")
  (?myReport setElementStatus "reportLowTempC" "red"))

(defrule hi-low-temp-reset
  "Low temp no longer higher than high temp"
  ?warning <- (warning hi-low-temp ?id)
  (wxReport (id ?id) (reportHighTempC ?highTempC) (reportLowTempC
?lowTempC)(OBJECT ?myReport))
  (test (>= ?highTempC ?lowTempC))
  =>
  (printout t "Notification: " ?id " high/low temperature now
acceptable." crlf)
  (?myReport retractElementStatus "reportHighTempC" "red")
  (?myReport retractElementStatus "reportLowTempC" "red")
  (?myReport issueRetraction "High/Low temperatures now acceptable."
"red" "temp-mismatch")
  (retract ?warning))

(defrule altimeter-increase
  "Drastic change in altimeter setting"
  (wxcond (id ?id1)(altimeter ?alt1)(intInitVal ?init)(OBJECT ?cond1))
  (wxcond (id ?id2)(altimeter ?alt2)(previousCondition ?cond1)(OBJECT
?cond2))
  (test (and (<> ?alt2 ?init) (<> ?alt1 ?init)))
  (test (> (- ?alt2 ?alt1) 25))
  (not (warning alt-increase ?id2))
  =>
  (assert (warning alt-increase ?id2))
  (printout t "Advisory: ID " ?id2 " 25+ drop difference in altimeter
setting from previous line." crlf)
  (?cond2 issueWarning "Increase in altimeter greater than 25ins."
"yellow" "alt-increase")
  (?cond2 setElementStatus "altimeter" "yellow"))
```

```
(defrule altimeter-increase-reset
  "Altimeter change ok"
  ?warning <- (warning alt-increase ?id)
  (wxcond (id ?id1)(altimeter ?alt1)(intInitVal ?init)(OBJECT ?cond1))
  (wxcond (id ?id2)(altimeter ?alt2)(previousCondition ?cond1)(OBJECT
?cond2))
  (test (or (= ?alt2 ?init) (= ?alt1 ?init) (<= (- ?alt2 ?alt1) 25)))
  =>
  (printout t "Notice: ID " ?id2 " altimeter change now ok." crlf)
  (?cond2 issueRetraction "Altimeter change now acceptable." "yellow"
"alt-increase")
  (?cond2 retractElementStatus "altimeter" "yellow")
  (retract ?warning))

;;;;;

(defrule altimeter-decrease
  "Drastic change in altimeter setting"
  (wxcond (id ?id1)(altimeter ?alt1)(intInitVal ?init)(OBJECT ?cond1))
  (wxcond (id ?id2)(altimeter ?alt2)(previousCondition ?cond1)(OBJECT
?cond2))
  (test (and (<> ?alt2 ?init) (<> ?alt1 ?init)))
  (test (> (- ?alt1 ?alt2) 25))
  (not (warning alt-decrease ?id2))
  =>
  (assert (warning alt-decrease ?id2))
  (printout t "Advisory: ID " ?id2 " decrease in altimeter greater than
25ins." crlf)
  (?cond2 issueWarning "Decrease in altimeter greater than 25ins."
"yellow" "alt-decrease")
  (?cond2 setElementStatus "altimeter" "yellow"))

(defrule altimeter-decrease-reset
  "Altimeter change ok"
  ?warning <- (warning alt-decrease ?id)
  (wxcond (id ?id1)(altimeter ?alt1)(intInitVal ?init)(OBJECT ?cond1))
  (wxcond (id ?id2)(altimeter ?alt2)(previousCondition ?cond1)(OBJECT
?cond2))
  (test (or (= ?alt2 ?init) (= ?alt1 ?init) (<= (- ?alt1 ?alt2) 25)))
  =>
  (printout t "Notice: ID " ?id2 " altimeter change now ok." crlf)
  (?cond2 issueRetraction "Altimeter change now acceptable." "yellow"
"alt-decrease")
  (?cond2 retractElementStatus "altimeter" "yellow")
  (retract ?warning))
```

```
(defrule tempo-altimeter
  "Altimeter setting included in tempo group"
  (wxcond (id ?id) (intInitVal ?initVal) (altimeter ?alt)
(changeModifier ?change) (OBJECT ?myCond))
  (test (eq ?change "TEMPO"))
  (test (<> ?initVal ?alt))
  (not (warning tempo-alt ?id))
  =>
  (assert (warning tempo-alt ?id))
  (printout t "WARNING: ID " ?id " altimeter included in a TEMPO
group." crlf)
  (?myCond issueWarning "Altimeter issued in a TEMPO group." "red"
"tempo-alt")
  (?myCond setElementStatus "altimeter" "red")
  (?myCond setElementStatus "changeModifier" "red"))

(defrule retract-tempo-altimeter
  "Altimeter/change group relationship normal"
  ?warning <- (warning tempo-alt ?id)
  (wxcond (id ?id) (intInitVal ?initVal) (altimeter ?alt)
(changeModifier ?change) (OBJECT ?myCond))
  (test (or (neq ?change "TEMPO") (= ?initVal ?alt)))
  =>
  (printout t "Notification: " ?id " change group/altimeter setting
normal." crlf)
  (?myCond retractElementStatus "altimeter" "red")
  (?myCond retractElementStatus "changeModifier" "red")
  (?myCond issueRetraction "Change group/altimeter setting now normal."
"red" "tempo-alt")
  (retract ?warning))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; System control rules

(defrule halt-me
  "Halt the Jess engine"
  (control (stop ?stop) (OBJECT ?control))
  (test (<> ?stop 0))
  =>
  (set ?control stop 0)
  (halt))

(defrule rules-command
  "Prints out current rules"
  (control (rules ?rules) (OBJECT ?control))
  (test (<> ?rules 0))
  =>
  (set ?control rules 0)
  (rules))
```

```
(defrule facts-command
  "Prints out current facts"
  (control (facts ?facts) (OBJECT ?control))
  (test (<> ?facts 0))
  =>
  (set ?control facts 0)
  (facts))

(defrule remove-weatherCondition
 "Removes the binding to the given WeatherCondition object"
  (wxcond (id ?id) (jessBinding ?binding)(OBJECT ?myCondition))
  (test (= ?binding 0))
  =>
  (undefinstance (?myCondition)))

(defrule retractWarning
 "Retracts warning for the given wxcond"
  ?warning <- (warning ?name ?id)
  (control (retractCommand ?retract)(id ?id)(OBJECT ?control))
  (test (<> ?retract 0))
  =>
  (retract ?warning)
  (printout t "Retracted Warning " ?name " for " ?id crlf)
  (?control retractionComplete))

(defrule sleep-if-bored
  "Wait for 100 and then wake up--just don't stop!"
  (declare (salience -100))
  ?idle <- (idle)
  =>
  (retract ?idle)
  (call java.lang.Thread sleep 100)
  (assert(idle)))

;; Actual creation of objects, as well as (reset) and (run) calls,
;; must be made form Java code that runs this batch file.
```

Appendix C – Design Specification Model

On the following pages is the object oriented Unified Modeling Code (UML) design specification for the intelligent user interface described in Chapter 3. This diagram shows the relationships between objects in the system.

The design is intended to be generic—that is, it can be implemented in any language, and employ any expert system shell. The interaction between system objects and expert system shell is largely implementation dependent, however, so modifications may have to be made to accommodate specific shells.

As an example, the design calls for an *ExpertSystemShellAgent* object to facilitate communications between the shell and the system objects (see Figure B2). As noted in §3.2.1, however, because a Java Expert System Shell (JESS) is employed as the core inference engine for this research, JavaBeans property listeners obviate the need for such explicit communications.

Figure B1 depicts the design specification UML model for the system maintenance functions (see §3.2.4.1). Figure B2 contains the model for the rest of the system. Finally, Figure B3 contains the implemented *ReportElement* subclasses for the demonstration implementation discussed in Chapter 4.

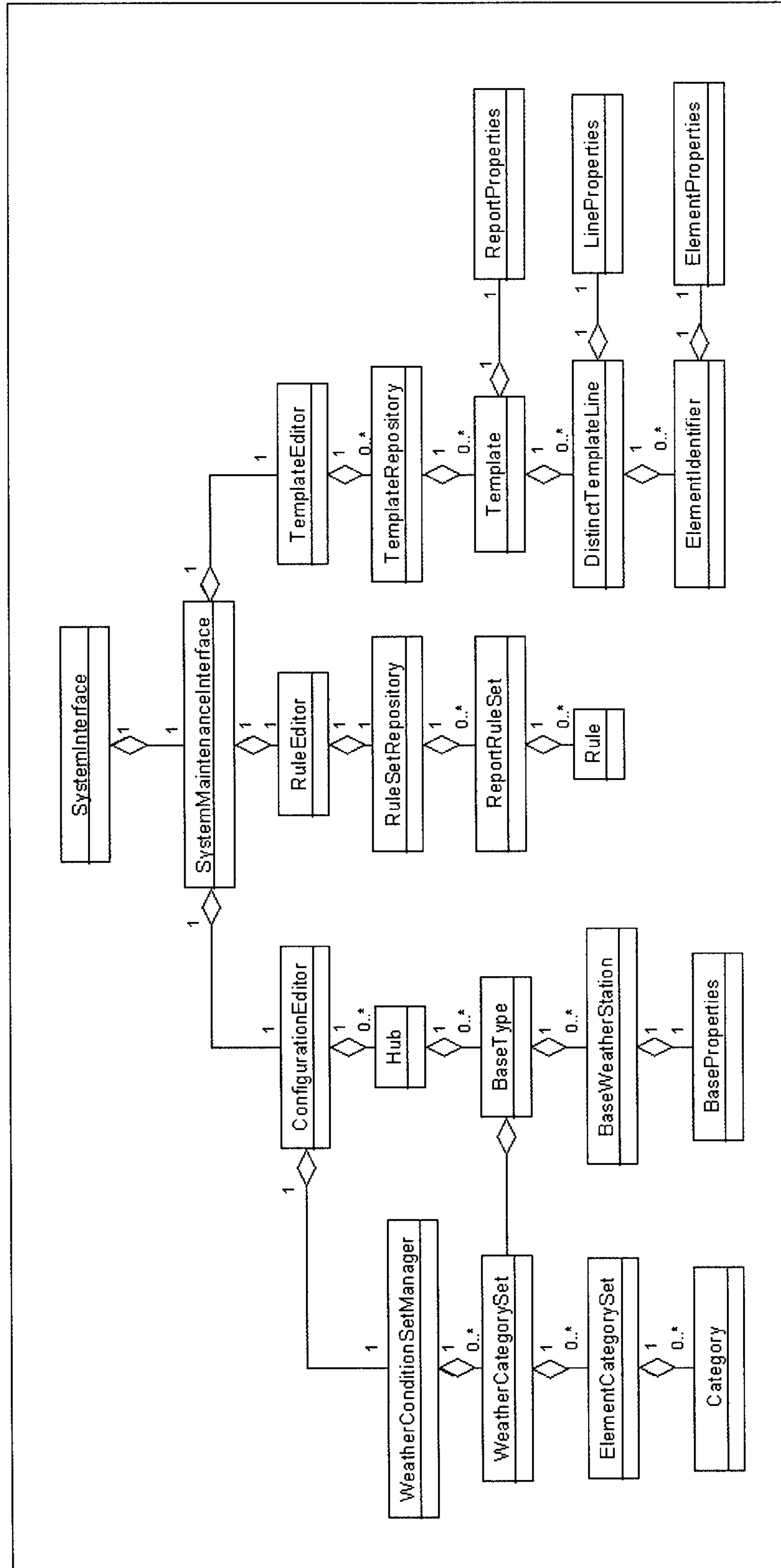


Figure C1: System Maintenance Design Specification

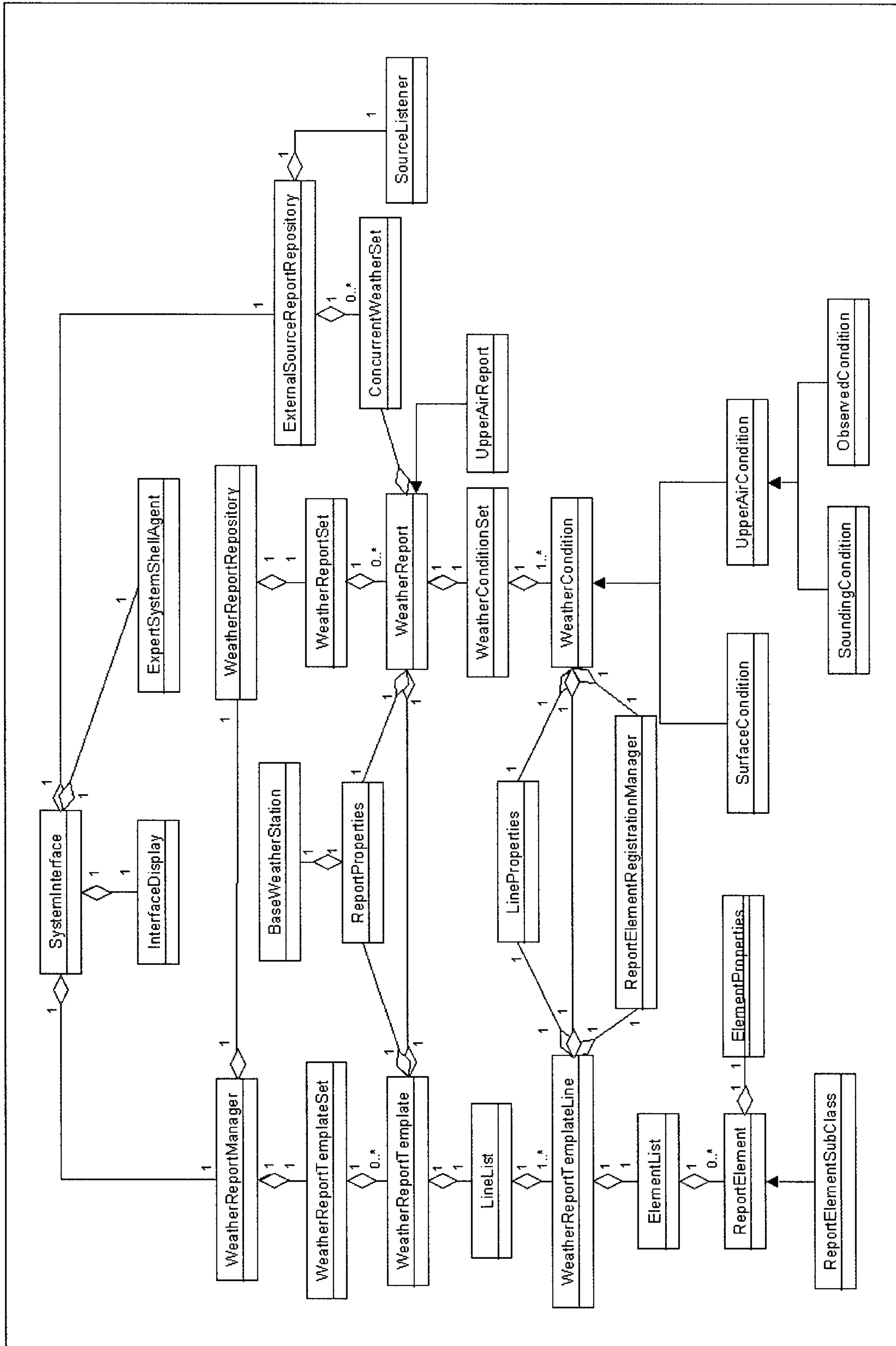


Figure C2: System Design Specification Model

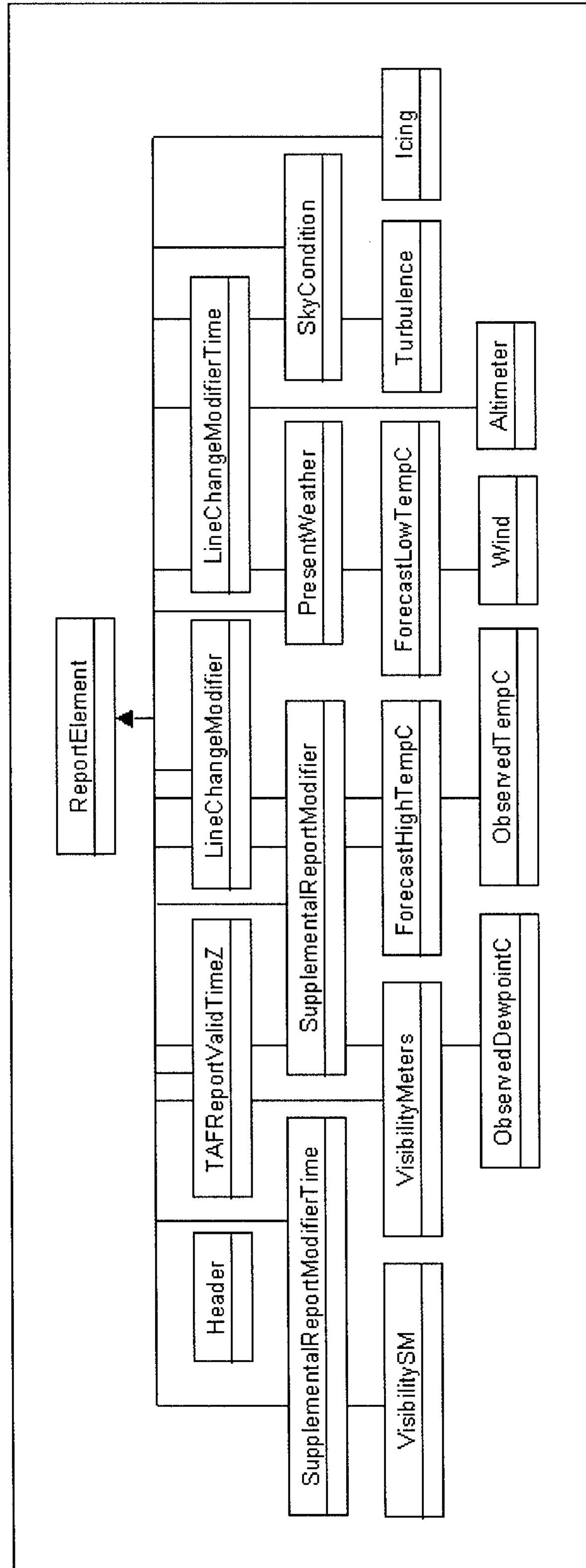


Figure C3: *ReportElement* Object Subclasses

Appendix D – Specification Data Dictionary

The following is the data dictionary generated by Rational Rose®, the tool used to build the design specification model for the critic system. It includes descriptions of all objects, methods and attributes in the design.

Data Dictionary Object Quick Reference

Logical View **126**

SystemInterface 126
SurfaceCondition 128
WeatherReportTemplateLine 130
ReportElement 132
WeatherReport 137
BaseWeatherStation 142
Hub 143
WeatherCategorySet 144
ConfigurationEditor 144
TemplateEditor 145
WeatherReportManager 145
WeatherReportTemplate 151
BaseType 156
UpperAirCondition 157
WeatherCondition 157
ConcurrentWeatherSet 161
TemplateRepository 163
Template 167
ReportRuleSet 168
Rule 170
ExternalSourceReportRepository 170
WeatherReportRepository 171
ElementIdentifier 174
InterfaceDisplay 174
Message 176
SourceListener 176
RuleSetRepository 177
TemplateRuleSet 177
ElementProperties 177
TemplateLineProperties 179
ElementCategorySet 180
Category 181
BaseProperties 181
StationSet 182
RuleEditor 182
DistinctTemplateLine 183
ReportProperties 184
LineProperties 186
ExpertSystemShellAgent 187
WeatherCategorySetManager 188
SystemMaintenanceInterface 188
SoundingCondition 188
PirepCondition 189
UpperAirReport 189
WeatherReportTemplateSet 189
LineList 190

ReportElementSubClass 190
WeatherConditionSet 192
WeatherReportSet 192
ElementList 192
ReportElementRegistrationManager 193

Logical View

SystemInterface

Called by Main, this object holds all references to the other system objects, including the WeatherReportManager, TemplateRepository, display objects, etc. A subclass of Java.awt.panel, the SystemInterface also provides the main GUI controls and template display panels for the application.

Private Attributes:

activeReportTemplate : WeatherReportTemplate

The WeatherReportTemplate object currently active (being viewed & edited).

Public Operations:

initSystem () :

Initializes system by instantiating the ReportEditor, the JessAgentManager, the ExternalSourceReportRepository, the SystemManagerFunctionInterface and the InterfaceDisplay.

SystemInterface () :

Constructor.

1. Construct WeatherReportManager.
2. Construct JESS engine objects.
3. Construct InterfaceDisplay objects.
4. Construct JESS display objects
5. Construct system maintenance objects.
6. Start JESS engine

execute () :

Method to initialize and start the JESS engine. For more information on JESS initialization, see the JESS documentation.

1. Load JESS user packages
2. Batchfile rules, if necessary.
3. Register the ExternalControl object with JESS

validateInterface () :

Validates (updates) all GUI components associated with the system.

updateTextAreaOutput () :

Calls display's report output update method.

resetTitle () :

Reset the title of the main window frame consistent with the base and report type of the active report.

addWeatherCondition (WeatherCondition wc :) :

Method to register the given WeatherCondition object with JESS.

To use JavaBeans capability, use JESS Funcall object to register an instance of the object. See JESS users manual for more information.

removeWeatherCondition (WeatherCondition wc :) :

Method to deregister the given WeatherCondition object with JESS.

See JESS users manual for more information.

addWeatherReport (WeatherReport wr :) :

Method to register the given WeatherReport object with JESS. As implemented, also registers the associated ReportProperties object.

See JESS users manual for more information.

removeWeatherReport (WeatherReport wr :) :

Method to deregister the given WeatherReport object with JESS. As implemented, also deregisters the associated ReportProperties object.

See JESS users manual for more information.

createReport () :

Method to instantiate a new report.

1. Get user input: base
2. Get user input: reportType
3. Retrieve template/ReportProperties objects associated with base/report type given
4. Call WeatherReportManager().createReport() passing ReportProperties object/template
5. Set the activeReportTemplate to the WeatherReportTemplate associated with the new report (retrieved form WeatherReportManager).
6. addWeatherReport() (to register with JESS).
7. validateInterface() (to update all GUI components).

deleteReport (WeatherReportTemplate wrt :) :

Method to remove a template from the display.

1. Remove the report from the set of activeReports.
2. removeWeatherReport() (remove binding with JESS.

3. validateInterface() (to update GUI components).

appendLine () :

Appends a line to the active report template.

1. Call WeatherReportManager.appendLine(activeReportTemplate).

receiveMessage (Message msg :) :

Method to update the display with a new feedback message.

Call display.addMessage(Message).

retractMessage (String id :) :

Method to remove a message from the display.

Call display.removeMessage(id), where id is the id of the Message object to remove.

changeMessageId (String oldKey : , String newKey : , String newLineNumber :) :

Method to update the id corresponding to a specific Message object. Also updates the line number of the Message object, so that accurate information can be presented to the user. (When Messages relate to specific ReportElements and those elements move from one line to another, the display information associated with those Messages has to be updated).

Call the display's changeMessageId method.

SurfaceCondition

A WeatherCondition that describes the state of the atmosphere as seen from a particular surface location. Associated most directly with an Observation or a line in a TAF. Fields, however, can be NULL so that other reports, which may only describe a particular subset of the surface weather condition, may be used to verify or validate reports.

The attributes provided for this design are "sample-only." A weather domain-knowedge expert would have to be consulted to determine a sufficient set of attributes for all conceivable uses.

The set of attributes depicted was implemented for the demonstration for this thesis, although a separate SurfaceCondition and WeatherCondition object was not.

Note: For purposes of the attribute descriptions, "current" means "during the valid times for this weather condition."

Derived from WeatherCondition

Private Attributes:

tempC : int

Current temperature (C).

dewPointC : int

Current dew point temperature (C).

ceilingFeet : int

Current ceiling (in feet).

rvr : String

Current runway visual range (METAR format string).

Future: Break out into specific runway/visual range information.

visibilitySM : float

Current visibility in statute miles.

presentWeather : String[]

Current weather and intensity (valid METAR codes).

Implemented as String (rather than array)

altimeter : float

Current station pressure (inches of mercury).

windSpeedKts : int

Current (predominant) wind speed in knots.

windGustKts : int

Current wind gust speed in knots.

windDirection : int

Wind direction in degrees.

windVariableLow : int

Low end of wind variablility direction. (I.E. if wind was variable from 020 to 180, this value would be set to 020).

windVariableHigh : int

Low end of wind variablility direction. (I.E. if wind was variable from 020 to 180, this value would be set to 020).

cloudLayers : String[]

List of reported cloud layers (CloudCover and AtmosphericHeight for each layer).

Implemented as String.

remarks : String

Free-flow remarks

freeText : String

Free text space.

name : type = initval

icing : String[]

Each layer represents an icing code group. For the purposes of this thesis, the icing layers are left in their METAR coded format. Future implementation could break the code group into lowerExtent, upperExtent and icingType.

turbulence : String[]

Each layer represents a turbulence code group. For the purposes of this thesis, the turbulence layers are left in their METAR coded format. Future implementation could break the code group into lowerExtent, upperExtent and turbulenceType.

WeatherReportTemplateLine

An instance of Template's DistinctTemplateLine which provides user with a set of element-specific user interfaces. Each WeatherReportTemplateLine, when data is entered by the user, serves to describe the state of the atmosphere (or WeatherCondition) at the associated BaseWeatherStation for a given valid time (either as a predominant or temporary condition, depending upon the report template configuration).

LineProperties objects are instantiated directly in the image of (copied from) the DistinctTemplateLine's LineProperties (stored persistently).

Private Attributes:

theLine : ElementList

The set of instantiated ReportElement objects for the line. (Based on the DistinctTemplateLine configuration).

c : LineControlElement

Element to exact line-specific control (i.e. insert/delete buttons and line labels)

lineProperties : LineProperties

The line properties associated with this object

weatherCondition : WeatherCondition

The WeatherCondition object associated with this line.

lineNumber : int

Line number of line in report. (Zero-based).

myReportTemplate : WeatherReportTemplate

Back-reference to the WeatherReportTemplate in which this line resides.

myManager : ReportElementRegistrationManager

The registration manager associated with the line (and its associated WeatherCondition)

tr : TemplateRepository

Local reference to the system TemplateRepository object (used for element instantiation)

lineType : String

Either "init," "insert", or "append." Passed through constructor.

Public Operations:

displayForEdit () :

Not needed for this thesis.

Displays the current line of the template for editing after all element objects have been instantiated.

WeatherReportTemplateLine (WeatherReportTemplate wrt : , TemplateRepository tr :) :

Constructor. Called when this is the first line in the report.

1. Set GUI components associated with this line.
2. Set local references to passed params.
3. Call the init() method to set up the line.

WeatherReportTemplateLine (String type : , int lineNumber; WeatherReportTemplateLine displacedLine : , TemplateRepository tr :) :

Constructor. Called for appended or inserted lines.

1. Set GUI components associated with this line.

2. Set local references to passed params.
3. Call the init() method to set up the line.

init () :

Method to initialize lines with elements as required by template.

1. String reportType = getMyTemplate().getMyProperties().getReportType() - (Get the report type from the properties object).
2. wc = new WeatherCondition(this, getMyTemplate().getMyReport()) - (Create a new WeatherCondition object).
3. ReportElementRegistrationManager rm = new ReportElementRegistrationManager(wc) - (Create a new registration manager)
4. Call appropriate method of TemplateRepository (getInitLine, getAppendLine, or getInsertedLine) to create and register the new line with the appropriate elements.
5. Perform any necessary updates required for GUI components associated with this line.

initValAllElments (DistinctTemplateLine : , WeatherReportTemplateLine :) :

Not implemented for this thesis.

1. Get old template line handle.
2. Create new line.
3. Add/register element copies of old line.

getLineNumber () : int

eraseLine () :

Not implemented for this thesis.

Iterates though elements and erases all values in template line.

sendMessage (Message : , lineNumber : , Priority :) :

Not implemented for this thesis. Message passing is direct from elements to SystemInterface via reference hierarchy.

Call the ReportTemplate's sendMessage method.

ReportElement

Partially implemented for this thesis.

Superclass for all WeatherElement and WeatherChoiceElement self-validating user-interface objects. Instantiated at run-time when a new WeatherReportTemplate is created based on the underlying Template object's DistinctTemplateLine configuration of ElementIdentifiers.

Several ReportElement subclasses have been implemented in support of this thesis. See the ReportElementSubClass object for a sample of what services these elements must provide.

Private Attributes:

typeName : String

Element name. Maps to ElementIdentifier elementSpecifier.

value : String

The value of the element (entered by user).

If the 'showLabel' attribute of the ElementProperties is set to 'true', the value of the element will include the label (for display purposes).

setOfValues : String

Values if the element in question is compound.

labelName : String

The name of the label associated with the element.

displayColor : color

Set from setValid, setInvalid, setQuestionable. Red if invalid, yellow if questionable. Standard text color (black) if valid.

validity : ElementValidity

ElementValidity can be one of {valid, invalid, questionable}

elementSpecifier : String

Unique attribute to identify this element. Corresponds to the elementSpecifier persistently stored in the ElementIdentfier object.

myLine : WeatherReportTemplateLine

Back reference to the line in which the element in question is stored. Necessary so that elements can manipulate the WeatherReport object.

myReportAttributes : HashSet(AttributeValuePairs)

A HashSet of AttributeValuePair objects that relate the name of the WeatherReport attribute this element modifies with the value.

Note that the value in the AttributeValuePair corresponding to each attribute name is set by subclass elements. These values are DERIVED from, but not necessarily EQUAL to user input (elementValue).

myConditionAttributes : HashSet(AttributeValuePairs)

A HashSet of AttributeValuePair objects that relate the name of the WeatherCondition attribute this element modifies with the element's value.

Note that the value in the AttributeValuePair corresponding to each attribute name is set by subclass elements. These values are DERIVED from, but not necessarily EQUAL to user input (elementValue).

elementName : String

Name of the element. Must match Class name of the subclass element.

elementValue : String

User input value.

REQUIRED : String = "Required"

RANGE : String = "Range"

FORMAT : String = "Format"

OTHER : String = "Other"

Public Operations:

displayForEdit () :

Not required for this thesis implementation.

Displays the element in question for editing.

ReportElement (String labelName : , int fieldWidth : , String elementProperties :) :

Constructor.

1. Set local attributes based on element subclasse's name, field width, properties.
2. Instantiate appropriate GUI components depending on whether subclass element is a regular text box, a choice element, or a label-only (non-editable text)
3. Initialize the display for the GUI components, if necessary.

verify () :

Abstract method. Each element must have a verify method which ensures the user entered text matches the correct format for the particular element. Each element included for the purposes of this thesis is derived from the standard METAR format for a TAF or Observation (METAR or SPECI) report type.

verifyBlankText () :

Method called when element value is blank.

IssueInternalWarning if the report has been "finalized()" and a required element is blank (see ElementProperties).

Finalization of WeatherReports not implemented.

runRegistrationTasks () :

Abstract method implemented by subclass elements. Performs any pre-registration tasks that need to be run (setting default values, etc.) Some elements that are automatically set require pre-registration formatting.

getModCondAttribs () : String[]

Abstract method implemented by subclass elements. Contains names of attributes of the WeatherCondition attributes this element modifies.

getModRepAttribs () : String[]

Abstract method implemented by subclass elements. Contains names of attributes of the WeatherReport attributes this element modifies.

lostFocus () :

Method to perform required operations when user leaves the entry box for this element.

1. Check to see if value has changed.
2. Call element's verify method if it has.

setValid () :

Sets the element as "containing valid information." (I.E. the information entered by the user has been QC'd and found to be valid.) Sets the displayColor as appropriate.

setInternalRedFlag (Boolean :) :

Set a red flag based on formatting.

setInternalYellowFlag (Boolean :) :

Set a yellow flag based on formatting.

setExternalRedFlag (Boolean :) :

Set a red flag from an external source (JESS)

setExternalYellowFlag (Boolean :) :

Set a yellow flag from an external source (JESS)

generateWarning (String type : , String text : , String severity :) :

Method to generate a warning associated with this element (i.e. based on bad formatting).

1. Format the message text.
2. Format the message ID based on ReportType, Base, Element, LineNumber, MessageType.
3. Create a new Message object.
4. Call the SystemInterface's receiveMessage method to send the Message object to display feedback.
5. IssueInternalWarning(severity) - (Sets internal flags)

Note: warning type is either REQUIRED, FORMAT, or OTHER.

issueInternalWarning (String severity :) :

Tracks issued Messages by severity.

1. Update the number of "severity" type message issued. (Use setInternalYellowFlag and setInternalRedFlag.
2. Paint the element background color based on highest severity of external/internal messages.

retractInternalWarning (severity :) :

Retracts a warning of the severity passed.

Update the number of "severity" warnings.
Paint the background either clear (if no other warnings exist) or based on the highest reminaing severity.

sendMessage (Message : , Priority :) :

Calls WeatherReportTemplate's sendMessage method if format error message is generated by any element. Not implemented for this thesis. Instead, message passing is done directly to the SystemInterface through the generateWarning() method.

updateWeatherCondition () :

Updates the WeatherCondition attributes associated with this element.

Iterate through myConditionAttributes and myReportAttributes extracting each AttributeValuePair. For each pair, update the WeatherCondition and/or WeatherReport object attributes with the associated value in the pair.

updateRegistrationManager () :

Method to update the registration manager when the element value changes.

register (ReportElementRegistrationManager rm : , WeatherReportTemplateLine line :) :

Registers elements as modifiers of either the WeatherCondition or WeatherReport object attributes.

1. runRegistrationTasks() - (Run any tasks, such as default formatting, that elements require before registration).
2. setUpAttributeValuePairs() - (Set up sets of objects to associate attribute names and values).
3. Iterate through all report-level attributes and call WeatherReport().registerAttribute -- Pass WeatherCondition associated with the line in which this element resides. Necessary so that WeatherReport will be aware of the location of element(s) that modify its attributes (since report-level attributes will occur only once in the report).
4. Call the ReportElementRegistrationManager's registerElement() method, passing self-reference and the modCondAttribs and modRepAttribs arrays (set by subclass elements).
5. updateRegistrationManager() - (Update the registration manager with the element's value.)
6. updateWeatherCondition() - (Update the WeatherCondition or WeatherReport objects with the newly registered values).

deregister () :

Method to deregister this element.

Iterate through the set of AttributeValuePairs for WeatherCondition and WeatherReport attributes modified (myReportAttributes, myConditionAttributes) and remove them all from the set.

WeatherReport

Object to hold all information derived from user input into the template. There is a 1-to-1 relationship between templates and reports, and a 1-to-1 relationship between template lines and weather conditions.

Weather-specific report-level attributes (such as high temperature for the report, headers, etc) are included as sample only. A Weather-domain knowledge expert should be consulted to determine the required weather attributes for this (and the WeatherCondition) objects.

Private Attributes:

reportIdentifier : String

Unique identifier for the WeatherReport.

reportIssueDate : Date

Date of issue for the report (set externally by a ReportElement).

reportHeader : String

Header for the Report (set externally by a ReportElement).

reportValidStartTime : Date

Valid start date of report (set externally by a ReportElement).

reportValidEndTime : Date

Valid end date/time of the report (set externally by a ReportElement).

reportHighTempC : int

High temperature for the report (set externally by a ReportElement).

reportLowTempC : int

Low Celcius temperature for the report (set externally by a ReportElement).

highTempTime : Date

Time of the occurence of the high temperature (set externally by a ReportElement).

lowTempTime : Date

Time of the occurence of the low temperature for the report (set externally by a ReportElement).

supplementalReportModifier : String

Modifier for the report, if required. For instance, in a TAF, this would be AMD, COR or AMD COR (or null). (Set externally by a ReportElement).

supplementalReportModifierTime : Date

Time associated with the sup. modifier attribute (set externally by a ReportElement).

reportRemarks : String

Remarks (set externally by a ReportElement).

reportFreeText : String

Text data (set externally by a ReportElement).

modifiableAttributes : String[]

A String array containing the names of all attributes that could be modified externally. The array, in this case, would be initialized with {"reportIssueDate", "reportHeader", "reportValidStartTime", "reportValidEndTime", "reportHighTempC", etc.}.

reportStatus : StatusType

{Presubmission, Pending, Active, External}

Presubmission: Still being edited (validation rules apply)
Pending: Submit report requested, in final verification (validation and verification rules apply)
Active: Submitted. In the WorldwideWeatherNetwork. Uneditable. (verification rules apply)
External: Outside source validation/verification purposes

reportProperties : ReportProperties = NULL

The ReportProperties object associated with this report.

concurrentWeatherSet : ConcurrentWeatherSet = NULL

The ConcurrentWeatherSet object assoiciated with this report.

weatherReportTemplate : WeatherReportTemplate = NULL

The WeatherReportTemplate object associated with this WeatherReport. Instantiated as a bi-directional link between report and template, since during creation, the report must be accessed from the template, but after activation, the template must be accessed from the report (to generate a new template).

weatherConditionSet : WeatherConditionSet = NULL

Contains all WeatherCondition objects (applicable to this report) corresponding (1-to-1) with WeatherTemplateLines

attributesSet : HashSet

A quick-reference HashSet initialized with the values of the modifiableAttributes array.

weatherConditionMap : HashMap

A map containing references to WeatherCondition objects indexed by attribute name. Needed so that the WeatherReport has a back reference to the WeatherCondition (and therefore the WeatherReportTemplate and ReportElements) that modify its attributes (for warnings).

yellowWarnings : HashSet

A set of Message Id's for yellow-status warnings generated by this report.

redWarnings : HashSet

A set of Message Id's for red warnings generated by this report.

sourceID : String

Identifier for the source of this report (used for externally generated reports only).

Public Operations:

weatherReport (WeatherReportTemplate : , ReportProperties :) : WeatherReport

Constructor to instantiate a blank WeatherReport.

weatherReport (WeatherReportTemplate : , ConcurrentWeatherSet : , ReportProperties :) : WeatherReport

Constructor to instantiate a blank WeatherReport, but superseded from an old report. Report station, properties and rule set objects, as well as

ConcurrentWeatherSet associated with this report will be instantiated based on the passed objects.

weatherReport (sourceID : , templateName :) : WeatherReport

Constructor returns a NULL-valued WeatherReport for the given sourceID. Only used for External reports.

registerAttribute (String attributeName : , WeatherCondition wc :) :

Method to register an attribute that will be modified by an element associated with the given WeatherCondition object.

1. Verify that the attributeName is in the attributeSet.

2. Add the WeatherCondition reference to the weatherConditionMap indexed by the attribute name.

verifyAmmendmentEligibility () : Boolean

Not implemented for this thesis.

Verifies whether the report in question is eligible for ammendment.

- Must be "activated" (in the ActiveReportSet)
- Clock time must be before the next issue time for this report (see TemplateProperties issueHours).

verifyCorrectionEligibility () :

Not implemented for this thesis.

Verifies whether the report in question is eligible for correction.

- Must be "activated" (in the ActiveReportSet)
- Clock time must be before the next issue time for this report (see TemplateProperties issueHours).
- Clock time must be within corEligibilityMaxTime (see TemplateProperties) minutes of issue time for the report.

requestChange (ReportElement re : , WeatherCondition wc : , String attributeName : , String value :) :

Method external objects (ReportElements) use to modify the attributes of this object. ReportElements must be registered with the ReportElementRegistrationManager associated with the WeatherCondition passed by reference to this method.

1. Verify (through wc reference to registration manager) that the attributeName can be modified by the ReportElement given.

2. If so, call the attribute's set-method, passing value as the parameter.

Note that for this design, for consistency, all set-methods accept String representations of the attributes they modify, regardless of actual attribute type. The conversion from String to int, float, etc., is accomplished by the set-method.

getStringReport () :

Returns a string version of the formatted report. I.E. will return a formatted METAR report (retrieved from ReportProperties object).

addWeatherCondition (WeatherCondition :) :

Adds WeatherCondition to the WeatherReport's WeatherConditionSet. Order is inferred from valid times.

removeWeatherCondition (lineNumber :) :

Removes the specified WeatherCondition from the WeatherConditionSet.

setStatus (reportStatus :) :

Sets the report status for this report. Status is either Presubmission, Pending, Active or External.

setElementStatus (String attributeName : , String color :) :

Used to externally set an individual ReportElement's status (in the event that data unput through that element caused a JESS rule to fire).

1. Get the WeatherCondition reference for the attributeName from the weatherConditionMap.

2. Get the ReportElement reference for the attributeName from the ReportElementRegistrationManager associated with the WeatherCondition reference.

3. Call the ReportElement's issueExternalWarning method with the given severity (color).

retractElementStatus (String attributeName : , String color :) :

Used to externally retract an individual ReportElement's status (in the event that data input through that element caused a JESS warning to be retracted).

1. Get the WeatherCondition reference for the attributeName from the weatherConditionMap.

2. Get the ReportElement reference for the attributeName from the ReportElementRegistrationManager associated with the WeatherCondition reference.

3. Call the ReportElement's retractExternalWarning method with the given severity (color).

issueWarning (String msg : , String severity : , String messageType :) :

Method used by JESS to issue a warning based on report attributes.

1. Uniquley format a message id based on information passed.
2. set red or yellow warning based on severity.
3. Create a new Message object.
4. Send the message to the SystemInterface via its receiveMessage() method.

issueRetraction (String msg : , String severity : , String type :) :

Method for external (JESS) objects to retract a previously issued message.

1. Format the message id based on the information passed.
2. Call the SystemInterface's retractMessage() method, passing it the formatted message id.

setRedWarning (Boolean add : , String id :) :

Method to add or remove (if Boolean 'add' is true or false, respectively) a red warning id from the set of redWarnings.

setYellowWarning () :

Method to add or remove (if Boolean 'add' is true or false, respectively) a yellow warning id from the set of yellowWarnings.

reportHasBeenDeleted () :

Mehtod to perform tasks when the WeatherReport object is pending deletion.

1. Iterate through red and yellow warnings HashSets and retract message id's found.

updateSystemInterface () :

This "dummy" method has been added to indicate the need to notify the SystemInterface (JESS) of updates to the attributes for WeatherReport objects registered with JESS.

In a Java implementation, the WeatherReport can be defined as a JavaBean, and propertyChangeEvents fired in every set method for every modifiable attribute (see demonstration code, WeatherReport object).

For other implementations, the JESS (or other expert knowledge system shell) needs to be notified of attribute value changes by some other (implementation specific) mechanism.

<u>BaseWeatherStation</u>

Partially implemented for this thesis.

A BaseWeatherStation is the location for which a forecast or observation report is being generated (via this application). For the purpose of this thesis, it is named "BaseWeatherStation," but could be renamed to simply "ReportLocation" to

extend the system to cover tactical locations, landing zones, drop zones, and other forecast/observation points as needed.

Information for this class is taken from the "Master Station Catalog," and is not limited to these attributes. Only attributes needed to demonstrate "proof-of-concept" are implemented.

Private Attributes:

ICAO : String

International Civil Aviation Organization (ICAO) code for the station.

stationName : String

Familiar name for the location.

stationElevation : int

Station elelvation in FEET

stationLocation : Coordinate

Station location in GEOGRAPHIC COORDINATE.

blockStation : String

International standard numeric identifier for the location.

stationType : BaseType

BaseType object (container class) of which this station is a member.

baseProperties : BaseProperties

The BaseProperties object associated with this station.

Public Operations:

getBaseProperties () : BaseProperties

getBaseType () :

Hub

Not implemented for this thesis.

Class to hold identification data for a particular Operational Weather Squadron (OWS) or Hub. Forecast responsibility for all bases (and other locations) is assigned to a particular Hub.

Private Attributes:

name : String

Familiary name (or designation)

hubID : String

Unique identifier for the Hub.

baseTypeSet : Set(BaseType)

Set of base "types" associated with the Hub.

WeatherCategorySet

Not implemented for this thesis.

Container to hold and manipulate weather category threshold schemas. In the past, schemas have been Command-specific. This design associates a weather category set with a "type" of base, that may be grouped arbitrarily within a Hub. A WeatherCategorySet may be chosen to be associated with a particular BaseType at run-time (Not implemented for this thesis).

Private Attributes:

catSetID : String

The category set will have a unique, user-defined name for easy recognition.

elementSet : Set(ElementCategorySet)

Container which holds the various element-specific category thresholds (modifiable at run-time).

ConfigurationEditor

Not implemented for this thesis.

Interactive interface to add, modify and delete Hub and Base-specific information (names, identifiers, properties, weather category thresholds, etc.)

Public Operations:

initEditor () :

Method to instantiate the WeatherCategorySetManage and initialize the Hub/BaseType/BaseWeatherStation/BaseProperties objects, and associate them with their persistent storage addresses.

getBaseWeatherStation (ICAO :) :

Returns the BaseWeatherStation object with the corresponding ICAO base identifier.

getWeatherCategorySet (ICAO :) :

Returns the WeatherCategorySet object associated with the particular ICAO.

TemplateEditor

Not implemented for this thesis.

Central interface for supervisor (user) management of report templates.

Private Attributes:

activeTemplate : Template

The currently active template.

Public Operations:

createTemplate () :

Create a new template type.

listTemplates () :

List available (instantiatable) template types.

deleteTemplate () :

Delete a template.

editTemplate () :

Modify a template.

saveTemplate () :

Save the current modifications to persistent storage.

loadTemplate (templateName :) : Template

WeatherReportManager

Partially implemented for this thesis.

Class to implement report editing functions. Allows creation and editing of reports based on a particular template. Maintains a repository of all report templates being edited. Provides methods to create and delete reports, as well as to append, insert, and delete lines from specified reports. Liason object between the SystemInterface (which controls the display function and the JESS inference engine) and the report templates and reports themselves.

Private Attributes:

activeReportSet : WeatherReportTemplateSet

See

The repository containing all "active" templates--that is, templates which have been instantiated as a report, but have not yet been submitted.

tr : TemplateRepository

A reference to the TemplateRepository. Maintained and passed to new templates as they are created.

myInterface : SystemInterface

Back reference to the system interface.

Public Operations:

WeatherReportManager (SystemInterface jess : , TemplateRepository _tr :) :

Constructor. Passed references to the SystemInterface (for easy access to JESS) and the TemplateRepository (passed to new templates as they are instantiated.

createNewReport (sourceID : , templateName :) : WeatherReportTemplate

Instantiates a new report based on the template configuration and a particular base. Creates a new WeatherReportTemplate object, and provides it with applicable line and element identifier information so that the object class heirarchy can be instantiated in the image of the template.

1. VerifyCreationEligibilty(base, type) - (Ensure no duplicate reports.)
2. WeatherReportTemplate wrt = new WeatherReportTemplate(rp, this) - (Instantiate a new WeatherReportTemplate object, passing it a reference to the report properties object (obtained from the template) and a self reference (to the manager)
3. appendLine(wrt) - (Append a new line to the new template.)
4. getActiveReportSet().addTemplate(wrt) - (Add the new template to the set of active templates)
5. getReportRepository().addPreSubmissionReport(wrt.getMyReport())) - (Add the report associated with the new template (created by the template upon construction) to the ReportRepository)
6. return wrt - (Return the WeatherReportTemplate object reference.

supersedeReport (WeatherReport :) :

Not implemented for this thesis.

Create a new report based on the WeatherReport in question.

1. String reportType = WeatherReport.getReportProperties().getReportType() - (Get the reportType of the old report).

2. BaseWeatherStation theStation =
WeatherReport.getReportProperties().getStation();
String sourceID = theStation.getICAO() - (Get a reference to the old BaseWeatherStation object).
3. verifyCreationEligibility(theStation, reportType)
WeatherReportTemplate wrt = new WeatherReportTemplate(WeatherReport);
4. WeatherReport wr = wrt.getWeatherReport() - (Get a handle to the WeatherReport object associated with the newly created template (see WeatherReportTemplate constructor).
5. addReportTemplate(wrt) - (Put the new report in the activeTemplateRepository).
6. WeatherReportRepository.addPreSubmissionReport(wr) - (Add the report to the repository as a pre-submission report.
7. getSystemInterface().validateInterface() - (Ensure the graphical display is updated with the new components).

deleteReportTemplate (reportTemplateID :) :

Method to delete the specified report template.

1. WeatherReport wr = wrt.getMyReport() - (Obtain a reference to the report associated with the template to be deleted).
2. getActiveReportSet().removeTemplate(wrt)) - (Remove the template from the set of active templates)
3. getSystemInterface().deleteReport(wrt) - (Inform the SystemInterface of the deletion so that it may be deregistered from JESS)
4. WeatherConditionSet = wrt.templateBeingDeleted() - (Inform the template it is being deleted so that it may perform final operations. Retrieve the set of WeatherCondition objects associated with the deleted template).
5. for each object in the set, removeWeatherCondition(wc) - (Call the (local) method to remove WeatherCondition objects individually from Reports.
6. wr.reportHasBeenDeleted() - (Inform the WeatherReport that it is going away, so that all its active messages can be cleared.
7. getRepository().removePreSubmissionReport(wr)) - (Remove the report from the repository).

createAmmendedReport (WeatherReport :) :

Not implemented for this thesis.

Create a new report based on the WeatherReport in question. Set its status to "ammended report."

1. String reportType = WeatherReport.getReportProperties().getReportType() - (Get the reportType of the old report).
2. BaseWeatherStation theStation =
WeatherReport.getReportProperties().getStation();
String sourceID = theStation.getICAO() - (Get a reference to the old BaseWeatherStation object).
3. verifyCreationEligibility(theStation, reportType)

WeatherReportTemplate wrt = new WeatherReportTemplate(WeatherReport, WeatherReportTemplate.AMMENDMENT);
4. WeatherReport wr = wrt.getWeatherReport() - (Get a handle to the WeatherReport object associated with the newly created template (see WeatherReportTemplate constructor).
5. addReportTemplate(wrt) - (Put the new report in the activeTemplateRepository).
6. WeatherReportRepository.addPreSubmissionReport(wr) - (Add the report to the repository as a pre-submission report.
7. getSystemInterface().validateInterface() - (Ensure the graphical display is updated with the new components).

createCorrectedReport (WeatherReport :) :

Not implemented for this thesis.

Create a new report based on the WeatherReport in question. Set its status to "corrected report."

1. String reportType = WeatherReport.getReportProperties().getReportType() - (Get the reportType of the old report).
2. BaseWeatherStation theStation = WeatherReport.getReportProperties().getStation();
String sourceID = theStation.getICAO() - (Get a reference to the old BaseWeatherStation object).
3. verifyCreationEligibility(theStation, reportType)
WeatherReportTemplate wrt = new WeatherReportTemplate(WeatherReport, WeatherReportTemplate.CORRECTION);
4. WeatherReport wr = wrt.getWeatherReport() - (Get a handle to the WeatherReport object associated with the newly created template (see WeatherReportTemplate constructor).
5. addReportTemplate(wrt) - (Put the new report in the activeTemplateRepository).
6. WeatherReportRepository.addPreSubmissionReport(wr) - (Add the report to the repository as a pre-submission report.
7. getSystemInterface().validateInterface() - (Ensure the graphical display is updated with the new components).

receiveMessage (Message m :) :

Not implemented for this report. Messages are sent directly to the SystemInterface from the ReportElement, WeatherCondition or WeatherReport objects via back-references.

verifyCreationEligibility (Template : , sourceID :) : Boolean

Verifies that no other similar presubmission report exists. Returns true if elgible, false if not.

return (!getActiveReportSet().containsType(base, type)) - (Query the activeReportSet object for the existence of a report matching the base and reportType passed. If one exists, eligibiliity is denied).

getWeatherReportTemplate (reportTemplateID :) :

Not implemented in this thesis.

Method to return WeatherReportTemplate specified by the template's id.

1. return activeReportSet().getTemplate(id) - (Call the getTemplate method of the WeatherReportTemplateSet.

activateReport (WeatherReportTemplate :) : Boolean

Not implemented for this thesis.

User call to activate the report currently being edited.

Sets the reports status to active and moves to pending report repository until all (if any) JESS messages are resolved. If Messages exist that will prohibit report submission, the report is moved back to the PreSubmissionReportRepository for editing. Otherwise, it is moved to the activeReportTemplate.

receiveReturnedReport (WeatherReportTemplate : , Message :) :

Method called when a pending WeatherReport fails verification and is returned for edit. Places the report back into the PreSubmissionReportRepository.

appendLine (WeatherReportTemplate :) : Boolean

Appends a line to the passed WeatherReportTemplate.

1. WeatherReport wr = wrt.getMyReport() - (Get handle to the WeatherReport associated with the template).
2. WeatherCondition wc = wrt.appendNewLine(getTemplateRepository()) - (Instruct the template to append a line and return the newly created WeatherCondition object. Pass the template a reference to the system's template repository object).
3. addWeatherCondition(wr, wc) - (Add the WeatherCondition to the WeatherReport).
4. getSystemInterface().validateInterface() - (Ensure the graphical display is updated with the new components).

insertLine (WeatherReportTemplateLine wrt : , int lineNumber :) :

Method to insert a line into the specified WeatherReportTemplate object at the specified line number.

1. WeatherReport wr = wrt.getMyReport() - (Get handle to the WeatherReport associated with the template passed).
2. WeatherCondition wc = wrt.insertLine(lineNumber, getTemplateRepository()) - (Instruct the template to insert a line at the specified line number and return the newly created WeatherCondition object. Pass the template a reference to the system's template repository).
3. addWeatherCondition(wr, wc) - (Add the new WeatherCondition object).

4. getSystemInterface().validateInterface() - (Inform the SystemInterface that components have been added).
//wr.updateWeatherConditionReferences();

deleteLine (WeatherReportTemplate wrt : , int lineNumber :) :

Method to delete a line from the given template.

1. WeatherCondition wc = wrt.deleteLine(lineNumber, getTemplateRepository()) - (Instruct the template to delete a line, returning the removed WeatherCondition object).
2. removeWeatherCondition(wc) - (Remove the WeatherCondition).
3. getSystemInterface().validateInterface() - (Inform the SystemInterface of a component change).

addWeatherCondition (WeatherReport rp : , WeatherCondition wc :) :

Method to add a WeatherCondition to a report.

1. wr.addWeatherCondition(wc) - (Tell the WeatherReport object to add the WeatherCondition).
2. getSystemInterface().addWeatherCondition(wc) - (Notify Jess of the newly-created WeatherCondition object).

removeWeatherCondition (WeatherCondition wc :) :

Method to remove a WeatherCondition object.

1. WeatherReport wr = wc.getMyWeatherReport() - (Obtain a reference to the WeatherReport associated with the WeatherCondition object).
2. getSystemInterface().removeWeatherCondition(wc) - (Tell the SystemInterface that the WeatherCondition is going away. It will be deregistered from JESS).
3. wc.weatherConditionIsBeingDeleted() - (Inform the WeatherCondition that it will be deleted. It will clear any pending messages it has generated).
4. wr.removeWeatherCondition(wc)) - (Remove the WeatherCondition from the WeatherReport).

getTemplateRepository () :

Returns a reference to tr, the system's TemplateRepository object.

Private Operations:

removeReportTemplate (WeatherReportTemplate :) :

Removes a template from the activeTemplateRepository.

addReportTemplate (WeatherReportTemplate :) :

Adds a report template to the activeTemplateRepository (reports being edited).

WeatherReportTemplate

An instantiated weather report entry form configured according to the associated "Template" object and BaseWeatherStation (specified at creation time). TemplateProperties objects are instantiated directly in the image of (copied from) the Template'sTemplateProperties (stored persistently).

Private Attributes:

templateLines : Set(WeatherReportTemplateLine)

Set of WeatherReportTemplateLines (containing ReportElement objects according to its associated DistinctTemplateLine object).

weatherReport : WeatherReport

The WeatherReport object associated with this template. Instantiated as a bi-directional reference.

reportProperties : ReportProperties

The ReportProperties object associated with this report template.

reportTemplateID : String

System-generated template Identifier for the template object.

numberOfLines : int

Indicates the number of lines in the report.

Maintainence of this variable left to coding, since some set objects (in language-specific libraries) will handle it.

AMMENDMENT : int = 0

Ease of use attributes for constructors.

CORRECTION : int = 1

Ease of use attributes for constructors.

AMMEND_COR : int = 2

Public Operations:

insertLine (WeatherReportTemplateLine : , lineNumber : , TemplateRepository :) :

Method to insert a new line into the report BEFORE the line specified.

1. WeatherReportTemplateLine newLine = null - (Establish a variable for the new line).

2. WeatherReportTemplateLine displacedLine = null - (Establish a variable for theline displaced by the insertion).
3. displacedLine = getLineList().getLineAtIndex(lineNumber) - (Get a handle to the displaced line).
4. newLine = new WeatherReportTemplateLine("insert", lineNumber, displacedLine, this, tr) - (Construct a new WeatherReportTemplateLine).
5. lineAdded() - (Perform processing required when a line is added to the list).
6. theLineList.insertLineAtIndex(lineNumber, newLine) - (Append the line to the LineList).8. getMyProperties().updateReportText() - (Update the report text with the new line's information).
7. return newLine.getWeatherCondition() - (Return a reference to the WeatherCondition created when the new line was instantiated).

deleteLine (lineNumber : , TemplateRepository tr :) : Boolean

Method to remove a line from the report. At least one line must exist. If only one line exists, it is considered to be both the first and the last line in the report.

1. String reportType = this.getMyProperties().getReportType() - (Find out the report type for this report).
2.. int lastLineNumber = this.getNextLineNumber() - 1; - (Determine the number of the last line in the report).
3. deleteLine = getLineList().getLineAtIndex(lineNumber) - (Get a reference to the line to be deleted).
4. wc = deleteLine.getWeatherCondition() - (Get a reference to the associated WeatherCondition object).

If deleting first line:
5. displacedLine = getLineList().getLineAtIndex(lineNumber+1) - (Get a handle to the displaced line)
6. tr.deleteFirstLine(deleteLine, displacedLine, lastLineNumber, reportType) - (Tell TemplateRepository about deletion. Pass displaced and deleted line references so elements can be moved, if necessary).

If deleting last line:
5. displacedLine = getLineList().getLineAtIndex(lineNumber-1) - (Get a handle to the displaced line)
6. tr.deleteLastLine(deleteLine, displacedLine, lastLineNumber, reportType) - (Tell TemplateRepository about deletion. Pass displaced and deleted line references so elements can be moved, if necessary).

If deleting a standard line, no elements will be moved (steps 5 & 6 not required).

7. getLineList().deleteLineAtIndex(lineNumber) - (Remove the list).
8. Perform any necessary operations associated with the GUI components for the deleted line.
9. lineDeleted() - (Perform delete-related operations (decrement line numbers)
10. deleteLine.lineIsBeingDeleted() - (Inform the delete line that it will be deleted.

11. getMyProperties().updateReportText() - (Update the report text associated with the report.
12. return wc - (Return the handle to the WeatherCondition associated with the template that is being deleted).

appendLine (TemplateRepository tr :) :

Method to insert a new line into the report after the last line in the report.

1. WeatherReportTemplateLine newLine = null - (Establish a variable for the new line).
2. WeatherReportTemplateLine previousLastLine = null - (Establish a variable for the old last line).
3. if (getNextLineNumber() == 0) newLine = new WeatherReportTemplateLine(this, tr) - (Appending a line, but there were no lines in the report to begin with).

Otherwise:

4. previousLastLine = getLineList().getLastLine() - (Get the last line in the report template).
5. newLine = new WeatherReportTemplateLine("append", getNextLineNumber(), previousLastLine, this, tr) - (Construct a new WeatherReportTemplateLine supplying it with the next available line number).
6. theLineList.appendLine(newLine) - (Append the line to the LineList).
7. lineAdded() - (Perform processing required when a line is added to the list).
8. getMyProperties().updateReportText() - (Update the report text with the new line's information).
9. return newLine.getWeatherCondition() - (Return a reference to the WeatherCondition created when the new line was instantiated).

lineAdded () :

Method to perform required operations when a line is added to the report. (Increment line numbers)

lineDeleted () :

Method to perform required operations when a line is added to the report. (Decrement line numbers)

displayForEdit () :

Not implemented for this thesis. Implementation was in Java, and template objects were subclasses of Java.awt components. All display methods were inherited.

Displays template for editing. Iterates all lines in LineSet and calls their display methods.

WeatherReportTemplate (ReportProperties rp : , WeatherReportManager manager :) :

Constructor. Constructs a new WeatherReportTemplate given the ReportProperties passed.

1. myManager = manager - (Set local reference to the WeatherReportManager object passed).
2. myProperties = rp - (Set the ReportProperties object reference).
3. Initialize GUI components associated with the template, if required.
   JAVA SPECIFIC: this.setLayout(new GridLayout(0,1)
4. theLineList = new LineList() - (Create a new line list object)
5. myReport = new WeatherReport(this, getMyProperties()) - (Create a new WeatherReport object, passing it a reference to the ReportProperties object and a self reference).
6. myProperties.setMyTemplate(this) - (Provide the ReportProperties object with a back (self) reference).
7. myProperties.setMyReport(myReport) - (Provide the ReportProperties object with a reference to the WeatherReport associated).
8. getMyProperties().updateReportText() - (Initialize the report text associated with the report).

WeatherReportTemplate (WeatherReport wr :) :

Not implemented for this thesis.

Constructor. Constructs a new WeatherReportTemplate in the image of the Template of the active (post-submission) WeatherReport provided. Initializes elements with valid values from the "old" WeatherReport.

1. Get a reference to the old report's WeatherReportTemplate
2. Create duplicate WeatherReportTemplateLines for all lines in the old report's WeatherReportTemplate LineList that are still applicable (given current time) - Use initValuedReport()
3. Create and set local reference to new ReportProperties object, created in the image of the old report's properties object.
4. Set local reference to the WeatherReportManager

WeatherReportTemplate (WeatherReport wr : , int type :) :

Not implemented for this thesis.

Constructor. Constructs a new WeatherReportTemplate in the image of the Template of the active (post-submission) WeatherReport provided, and sets the new Properties object to ammendment or correction depending on type (using AMMENDMENT or CORRECTION attributes). Initializes elements with valid values from the "old" WeatherReport.

1. Get a reference to the old report's WeatherReportTemplate
2. Create and set local reference to new ReportProperties object, created in the image of the old report's properties object.
3. Set ReportProperties "isAmmendment" and/or "isCorrection" attributes as appropriated.

3. Create duplicate WeatherReportTemplateLines for all lines in the old report's WeatherReportTemplate LineList that are still applicable (given current time) - (use initValuedReport()).
4. Set local reference to the WeatherReportManager

setForAmmendment () :

Not implemented in this thesis.

\
Method to flag the new WeatherReportTemplate as an ammendment to an "active" report.

setForCorrection () :

Not implemented in this thesis.

Method to flag the new WeatherReportTemplate as a correction to an "active" report.

changeBase (BaseWeatherStation :) :

Not implemented for this thesis.

Mechanism to change the base for which the report is valid. Affects:

1. Which base property set is used.
2. Which category set is used.

clearTemplate () :

Not implemented for this thesis.

Iterates list and clears contents of all lines. Does not delete any additional lines inserted via insert lines.

deleteAllLines () :

Not implemented for this thesis.

Deletes all but required initial line(s) of report. Does not clear elements in the first line if they contain values.

sendMessage (Message : , Priority :) :

For this thesis, message passing was directly from elements to manager via reference hierarchy.

Calls WeatherReportManager's receiveMessage method when format QC message generated from any element.

getWeatherReport () : WeatherReport

getReportProperties () : ReportProperties

initValuedReport (WeatherReportTemplate :) :

Not implemented for this thesis.

Initializes the WeatherReportTemplate object (this) based on an existing WeatherReportTemplate object (passed parameter). Uses current (run-time) time and initializes the new template (and WeatherReport object) with old report data that is still valid. (I.E. if a WeatherReport is valid for 24-hours from 02Z to 02Z, and at 07Z the user initiates a "supersede" command, this method will initialize (this) WeatherReportTemplate with the values in the original template from 07Z onward.

1. Time currentTime = <getClockTime> - (Get current time).
2. oldTemplate = WeatherReportTemplate.getProperties().getTemplate() - (Get reference to old template)
3. oldTemplateLines = WeatherReportTemplate.getLineList() - (Get reference to the old LineList
4. for each oldLine in oldTemplateLines, if (line.getValidStartTime >= currentTime):

Instantiate a template line based on the DistinctTemplateLine object and the oldLine.

Pass the WeatherReport associated with this WeatherReportTemplate so that the WeatherCondition object created (associated with the new line) can have a append itself to the WeatherReport object.

appendLine(createNewValuedLine(oldLine) - (Append the new line to the report)

getTemplateLines () : LineSet

BaseType

Not implemented for this thesis.

A base "type" is a new concept introduced to allow similar subsets of bases to be associated with a particular forecast category set. For instance, if a particular Hub is responsible for a base which contains only fighter aircraft as well as a base that houses only transport aircraft and paratroopers, forecast categories pertinent to the weather requirements for each of the airfields can be accounted for.

Private Attributes:

baseTypeID : String

Plain-language, user-intuitive identifier for the type.

stationSet : Set(BaseWeatherStation)

Set of BaseWeatherStations that belong to the particular "type" of installation.

categorySet : WeatherCategorySet

The WetherCategorySet associated with this object.

Public Operations:

setCategorySchema (WeatherCategorySet :) :

Associates base type with a particular category schema.

getCategorySchema () :

Returns the categorySet object associated with this base type.

UpperAirCondition

Not implemented for this thesis.

Derived from WeatherCondition

Private Attributes:

levelMil : Millibars

Level of atmosphere for which the condition is valid (in Millibars).

levelFeet : int

The level for which the condition is valid (in feet).

tempF : TempT

The temperature of the atmosphere at the given level/valid time.

dewPointF : TempF

The dew point of the atmosphere at the given level/time.

windDirection : CompassDirection

The direction of the wind at the given level/time.

windSpeed : UpperLevelWindSpeed

The speed of the wind at the given level/time.

WeatherCondition

The "state of the atmosphere." 1-to-1 relationship with a WeatherReportTemplateLine. The template line holds the user-entered information, and the "weather condition" line holds derived data that describes the atmosphere, as well as information sufficient to rebuild the complete report.

A weather condition could be a surface condition (i.e. a line in a surface weather report) or an upper air condition (i.e. a pirep or information gleened from sounding data. A full sounding would be a WeatherReport which consisted of many upper-air WeatherConditions--one for each level reported in the sounding.)

Alternate source data is placed into "WeatherCondition format," and stored in a WeatherReport object for easy comparison to other WeatherConditions (i.e. a forecast report currently being validated or verified.)

Private Attributes:

startValidTime : Time

The start valid time for the particular WeatherCondition. For a report being edited, this field is derived from the LineValidTime object.

endValidTime : Time

The end valid time for the particular WeatherCondition. For a report being edited, this field is derived from the LineValidTime object.

nextCondition : WeatherCondition

A reference to the next WeatherCondition in this WeatherReport (if it exists).

Used by JESS rules to compare attributes of consecutive WeatherCondition objects.

previoiusCondition : WeatherCondition

A reference to the previous WeatherCondition object in the WeatherReport, if it exists.

Used by JESS rules to compare attributes of consecutive WeatherCondition objects.

stringValue : String

"Value" of the weather condition formatted in accordance with the template provided. After the report is disassociated with the template, the string value of the WeatherCondition is maintained so that the report can be rebuilt for transmission. The "string value" of a TAF line, for instance, will be formatted in proper TAF format.

The string value of alternate source reports will be derived directly from the incoming reports.

isTemporary : Boolean

Variable indicating whether the condition is temporary or not. If temporary, the allowed minutes for a temporary condition are stored as part of the ReportProperties (for verification purposes).

myReport : WeatherReport

Back-reference to the report in which this condition is stored.

lineNumber : int

Line number indicating position of this condition in the report. Associated with WeatherReportTemplateLine's lineNumber attribute. (Zero based).

modifiableAttributes : String[]

The names of all of the attributes of the WeatherCondition object that are modifiable (for registration purposes).

redWarnings : HashSet

"Red" severity messages issued for this WeatherCondition, by ID.

yellowWarnings : HashSet

"Yellow" severity messages issued by this WeatherCondition, by severity.

Public Operations:

weatherCondition (WeatherReport :) :

Constructor returns a new NULL-valued WeatherCondition object.

weatherCondition (WeatherCondition : , WeatherReport :) :

Not implemented for this thesis.

Creates a WeatherCondition based on a copy of an existing WeatherCondition. (Set all attributes to the corresponding values of the old WeatherCondition)

weatherCondition (WeatherReportTemplateLine wrtl : , WeatherReport wr :) :

Constructor.

Associate WeatherCondition with the passed WeatherReportTemplateLine and WeatherReport.

requestChange (String attributeName : , String value : , ReportElement re :) :

Method to update local attributes, called by ReportElements.

1. Verify with ReportElementRegistrationManager that the ReportElement is eligible to update the attributeName attribute. (ReportElementRegistrationManager().isWeatherConditionUpdatePermitted())

2. Set the local attribute corresponding the the attributeName.

Note that for this design, for consistency, all set-methods accept String representations of the attributes they modify, regardless of actual attribute type. The conversion from String to int, float, etc., is accomplished by the set-method.

updateSystemInterface () :

This "dummy" method has been added to indicate the need to notify the SystemInterface (JESS) of updates to the attributes for WeatherCondition objects registered with JESS.

In a Java implementation, the WeatherCondition can be defined as a JavaBean, and propertyChangeEvents fired in every set method for every modifiable attribute (see demonstration code, WeatherCondition object).

For other implementations, the JESS (or other expert knowledge system shell) needs to be notified of attribute value changes by some other (implementation specific) mechanism.

weatherConditionIsBeingDeleted () :

Method to run tasks necessary when being deleted.

Iterate through redWarnings and yellowWarnings HashSets. For each entry found, issueRetraction based on the id of the message.

lineNumberHasBeenIncremented () :

Method to perform necessary tasks when another WeatherCondition has been inserted before this one in the WeatherReport.

1. Increment the line number.
2. Iterate through red and yellow warning HashSets. Issue retractions for all messages, reformat message id's based on new line number, and issueWarnings again.

lineNumberHasBeenDecremented () :

Method to perform necessary tasks when another WeatherCondition has been deleted before this one in the WeatherReport.

1. Decrement the line number.
2. Iterate through red and yellow warning HashSets. Issue retractions for all messages, reformat message id's based on new line number, and issueWarnings again.

updateReferences () :

Update references to previous and next WeatherCondition objects.

issueWarning (String message : , String severity : , String type :) :

Method to issue feedback about this WeatherCondition. (Method called as a result of a JESS rule firing).

1. Format the message ID based on the line, report type, base and message type
2. Instantiate a new Message object.
3. Send the message (via the SystemInterface().receiveMessage() method).
4. setRedWarning or setYellowWarning based on severity.

issueRetraction (String msg : , String severity : , String type :) :

Method to retract a warning.

1. Format message ID based on base, report type, message type, and line number.
2. Iterate appropriate red or yellow warning HashSets and find message with id matching formatted id.
3. Issue retraction via SystemInterface().retractMessage()

setYellowWarning (Boolean add : , String id :) :

If "add" is true, adds the String id to the HashSet or yellowWarnings IF the warning id doesn't already exist in the set.

Otherwise, removes the warning from the set if it exists.

setRedWarning (Boolean add : , String id :) :

If "add" is true, adds the String id to the HashSet or redWarnings IF the warning id doesn't already exist in the set.

Otherwise, removes the warning from the set if it exists.

ConcurrentWeatherSet

Not implemented for this thesis.

Not used for reports that are observations (that is the isForecast attribute of the ReportProperties object is "false.")

The set of weather concurrent to the weather report being verified or validated. The sourceID's for the currentReportSet, the localSurfaceOb and localUpperAirOb WeatherReport objects referenced here are extracted from BaseWeatherStation's BaseProperties object at instantiation of new WeatherReport/ConcurrentWeatherSet objects.

At instantiation, the ConcurrentWeatherSet registers itself with the ExternalSourceReportRepository. As updates to the associated weather reports are "heard" by the SourceListener, the ConcurrentWeatherSet data will be updated.

Private Attributes:

localSurfaceOb : WeatherReport

The specific weather report which holds the most recent (current) observation which would verify the report in question.

previousSfcObs : OrderedSet(WeatherReport):NULL

A set of previous observation (reports) that can be used to calculate, for example, pressure changes, temperature changes, weather changes, etc. Number of hours kept is determined by ReportProperties (hoursOfSfcObsToKeep attribute).

Ordered set, FILO.

localUpperAirOb : UpperAirReport

The specific weather report which holds the most recent (current) upper-air observation (sounding) associated with the report in question.

previousUAObs : OrderedSet(UpperAirReport):NULL

A set of previous upper-air soundings that can be used to calculate, for example, LCL changes, UA temperature changes, etc. Number of reports kept is determined by ReportProperties (numberOfSoundingsToKeep attribute).

Ordered set, FILO.

alternateSfcReportSet : Set(WeatherReport):NULL

Not implemented for this thesis

Alternate source reports could be other (nearby) station TAFs, UA reports, Obs, or any other decipherable report. In order for these reports to be valuable, a Jess rule associating the owner WeatherReport (forecast) with the alternate source report (via sourceID identification) must be written.

If the source exists but the rule doesn't, the source will be stored and updated, but will never be accessed. If the rule exists but the source does not, the rule will be ignored at runtime.

keepMaxHoursSfcObs : int

Maximum hours worth of observations to keep. (From ReportProperties)

keepMaxNumberUAReports : int

Max number of UA reports to keep. (From ReportProperties)

allCurrentReports : Set(WeatherReport)

Variable to keep a reference to all reports in the set.

myWeatherReports : Set(WeatherReport)

Set of weather reports that "own" this ConcurrentWeatherSet object.

Public Operations:

updateReport (WeatherReport : , WeatherReport :) :

Called when new data is received.

```
if (oldWeatherReport == getLocalSurfaceOb()) {
  pushSfcPreviousOb(oldWeatherReport);
  setLocalSurfaceOb(newWeatherReport);
}
else if (oldWeatherReport == getLocalUpperAirOb()) {
  pushUAPPreviousOb(oldWeatherReport);
  setLocalUpperAirOb(newWeatherReport);
```

```
}

for each report in myWeatherReports {
  report.concurrentWeatherUpdate();
}
```

concurrentWeatherSet (WeatherReport :) :

Constructor.

Set all local references (from WeatherReport's ReportProperties object)

addOwner (WeatherReport :) :

Adds an "owner" WeatherReport to the set of myWeatherReports. Provides a back-reference to all reports that might access this ConcurrentWeatherSets. Since a new (presubmission) report can supersede an active report, and both can access the same ConcurrentWeatherSet object, it is important to keep all back references.

removeOwner () :

Method to remove an owner from the set of owners. If the last owner is removed, the set is deactivated.

pushPreviousSfcOb () :

Method to maintain the previousSfcObs container.

pushPreviousUAObs () :

Method to maintain the previousUAObs container.

Private Operations:

deactivateSet () :

De-registers the concurrent weather set with the set repository. Called after all owner reports have been removed.

TemplateRepository

Container class to hold a set of (instantiatable) weather templates.

For this thesis, the TemplateRepository will be given more responsibility than it should in a full implementation. In this implementation, the persistently-stored template lines and elemets that make up a template are simulated by two arrays (TAF and Observation). Furthermore, instances of all template line types (first, last, standard--following ElementProperties attributes--are instantiated when the TemplateRepository is intialized.

In this design/implementation, when a new line is inserted into the report, based on the new line number and the line number of the line that was displaced by the insert, methods in the TemplateRepository instantiate new elements, move

elements between lines (as required by FirstLineOnly, etc., ElementProperties), register all of the newly instantiated elements, and reregister all of the moved elements with the WeatherCondition or WeatherReport objects they modify. (See the ReportElementRegistrationManager object).

For practical purposes, in a full implementation, in may be useful to provide these services in a separate object associated with the TemplateRepository.

Private Attributes:

templateSet : Set(Template)

The set of templates. Simulated in the implementation as two arrays (TAF and Observation).

allReportElements : ElementList

An element list containing all elements that appear in this template, regardless of properties.

reportInitialLine : ElementList

Elements appearing in the initial line of the active template (based on ElementProperties).

Note: The initial line is both the first and last line of a report (since it is the only one).

reportStandardLine : ElementList

Elements appearing in a standard line of the active template (based on ElementProperties).

reportLastLine : ElementList

Elements appearing in the last line of the active template (based on ElementProperties).

reportFirstLine : ElementList

Elements appearing in the first line of the active template (based on ElementProperties).

NOT_IN_FIRST : int = 3;

Ease of use variable to describe elements

NOT_IN_LAST : int = 2;

Ease of use variable to describe elements

FIRST_ONLY : int = 1;

Ease of use variable to describe elements

LAST_ONLY : int = 0;

Ease of use variable to describe elements.

packageName : String

jess : SystemInterface

Reference to SystemInterface object

Public Operations:

TemplateRepository (SystemInterface :) :

Constructor

addTemplate (Template :) :

Method to add a template to the collection. Ensures no duplicates, etc.

removeTemplate (Template :) :

Remove a specified template from the set.

saveAll () :

Saves all templates to persistent storage.

saveTemplate (Template :) :

Saves a particular template to persistent storage.

getTemplate (templateName :) :

Iterates set and returns template with the given name.

instantiateNewElementList () : ElementList

Iterates ElementList passed as an argument, instantiates a copy of each individual element in the list, and returns a newly instantiated copy of the old list.

instantiateReportElement (elementName :) : ReportElement

Method to instantiate new element based on its name.

In Java:

```
Object o = null;
try
{
  Class c = Class.forName(packageName + elementName);
  Constructor elementCons =
     c.getDeclaredConstructor(new Class[] {getJess().getClass()});
  o = elementCons.newInstance(new Object[] {getJess()});
```

```
}
catch {
  // All applicable exceptions
}

if (o == null) return null;
else return (ReportElement)o;
```

getPackageName () : String

getJess () : SystemInterface

Returns the reference to the SystemInterface (passed to the constructor on instantiation).

getInitLine (WeatherReportTemplateLine wrtl : , String reportType :) : ElementList

Method to instantiate a report's initial line. Arguments passed are a reference to the new WeatherReportTemplateLine and the reportType being instantiated. References to line number, properties, etc., can be obtained through the WeatherReportTemplateLine.

getInsertedLine (WeatherReportTemplateLine new : , WeatherReportTemplateLine displaced : , WeatherReportTemplateLine reportType :) :

Method to instantiate a report line that will be inserted. Arguments passed are a reference to the new WeatherReportTemplateLine, a reference to the WeatherReportTemplateLine that the new line will be displacing, and the reportType being instantiated. References to line number, properties, etc., can be obtained through the WeatherReportTemplateLine(s).

deleteFirstLine (WeatherReportTemplateLine line : , WeatherReportTemplateLine displaced : , int lastLine : , String reportType :) : Boolean

Method to perform appropriate element transfers and registrations when a line is being deleted. Arguments passed include references to the line being deleted, the line displaced (affected by the deletion), the lastLineNumber in the report, so that the position of the deleted/displaced lines in the report may be determined, and the report type.

Note that, for instance, if there are only two lines in the report and the last line is deleted, the remaining line becomes the first and last line, while if there are more than two lines, the second-to-last line becomes the last line.

deleteLastLine (WeatherReportTemplateLine line : , WeatherReportTemplateLine displaced : , int lastLineNumber : , String reportType :) : Boolean

Method to perform appropriate element transfers and registrations when a line is being deleted. Arguments passed include references to the line being deleted, the

line displaced (affected by the deletion), the lastLineNumber in the report, so that the position of the deleted/displaced lines in the report may be determined, and the report type.

Note that, for instance, if there are only two lines in the report and the first line is deleted, the remaining line becomes the first and last line, while if there are more than two lines, the second line becomes the first line.

removeNotInLastLineElements (ElementList list :) :

Method to remove elements whose properties indicate they should not appear in the last line of a report.

removeNotInFirstLineElements () :

Method to remove elements whose properties indicate they should not appear in the first line of a report.

moveFromOldLastToNewLast (ElementList newLast : , ReportElementRegistrationManager newRm : , WeatherReportTemplateLine newLine : , ElementList oldList :) :

Method to move elements from the old lasy line to the (newly inserted) new last line. Performs reregistering and registering of elements as required.

moveFromOldFirstToNewFirst (ElementList newFirst : , ReportElementRegistrationManager newRm : , WeatherReportTemplateLine newLine : , ElementList oldList :) :

Method to move elements from the old first line to the (newly inserted) new first line. Performs reregistering and registering of elements as required.

addLineElements (ElementList oldList : , ReportElementRegistrationManager oldRm : , WeatherReportTemplateLine oldLine : , String reportType : , int elementType :) :

Method to (instantiate and) add elements to a list based on the type of element passed (see ease of use attributes NOT_IN_FIRST, etc.)

This method performs all necessary registration.

moveLineElements (ElementList newList : , ReportElementRegistrationManager newRm : , WeatherReportTemplateLine newLine : , String reportType : , int element :) :

Method to move elements between the lists (old and new) passed as arguments based on the element type passed.

Template

Simulated for this thesis by TAF and Observation String arrays containing the names of elements appearing in the template.

The foundation container class that holds all of the line and element identifiers used to instantiate a particular weather report. Identifier (template) information is saved to persistent storage. Weather reports are "constructed" at run-time based on the configuration of element and line objects specified by template line and element identifiers and properties.

Private Attributes:

templateName : String

Unique (and user intuitive) identifier for the template. For instance, for the purposes of this thesis, a TAF and Observatoin template have been constructed to show proof of concept.

lines : Set(DistinctTemplateLine)

Set of distinct lines in a report.

See DistinctTemplateLine class description for more information.

Public Operations:

getTemplateProperties () : TemplateProperties

Returns the TemplateProperties object associated with this Template.

ReportRuleSet

Not implemented for this thesis.

A set of rules to be applied for a specific report type (i.e. associated with a particular, unique report template). Rules are used to:

1. Provide forecaster with constructive feedback based on input element values prior to forecast submission.

2. Provide forecaster feedback regarding negative trends in forecast as compared to actual observations. (I.E. ceiling decreasing faster than forecast, or decreasing vice increasing, etc.)

3. Provide forecaster feedback when forecast is or is nearing "out of category" compared to current observation.

4. <Future capability> Provide observer with feedback when automated weather-sensors (i.e. wind sensors, barometer, ceilometer, etc.) indicate "out-of-category" weather requiring issuance of a special observation

Rules are based on any of:

1. Report elements entered by the user (forecaster or observer) at run time.

2. If the report is a forecast, elements of the local, concurrent observation report defined in BaseProperties (see ConcurrentWeatherSet).

3. If the report is a forecast, elements of the local, concurrent upper-air observaton report defined in BaseProperties (see ConcurrentWeatherSet).

4. If the report is a forecast, elements of other station surface and upper-air reports defined in BaseProperties class, alternateSourceStationSets attribute. (see ConcurrentWeatherSet).

5. Base "type"-specific category thresholds for various report elements established in BaseType class.

6. Other base-specifc values for meteorological parameters established in the BaseProperties class.

Rules are employed by an expert knowledge system such as Jess or Clips. (This application implements Jess).

For the purposes of this thesis, a mininal set of rules will be generated to show proof-of-concept. Rules in support of this thesis are included in a ".clp" batchfile. The format of the batchfile is the JESS-accepted language. See the JESS documentation for proper format.

Private Attributes:

rules : Set(Rule)

A container holding the set of all rules.

ruleSetID : String

A unique identifier for the rule set.

stationProperties : BaseProperties = NULL

The properties object associated with the BaseWeatherStation that the report applies to.

categorySet : WeatherCategorySet = NULL

The WeatherCategorySet object used by this ReportRuleSet.

Public Operations:

addRule () :

Not implemented in this thesis.

Method to allow the addition of a rule.

removeRule () :

Not implemented in this thesis.

Method to allow the deletion of a rule.

listRules () :

Method to list all rules in the set.

reportRuleSet (ReportRuleSet : , BaseWeatherStation : , WeatherCategorySet :) :

Constructor operation that returns a NULL-valued ReportRuleSet object. Sets local handles to property variables, etc.

reportRuleSet (ReportRuleSet :) :

Constructor operation that returns a ReportRuleSet object based on an existing one (copy).

Rule

Not implemented for this thesis.

A rule instance. Rule syntax will be output in the syntax of Jess (a superset of CLIPS).

Private Attributes:

verificationValid : Boolean

Indicates whether the rule is valid for verification (pending and post-submission) purposes.

validationValid : Boolean

Indicates whether the rule is valid for validation (pre-submission) purposes.

theRule : String

The Jess-formatted String rule.

Public Operations:

printRule () : return

Displays the rule.

getRule () :

ExternalSourceReportRepository

The repository of all reports that are used to verify or validate all "active" weather reports instantiated in the system.

Private Attributes:

weatherSetPointers : Set(ConcurrentWeatherSet)

A set of active ConcurrentWeatherSet's. The sets are registered with the ExternasSourceReportRepository on instantiation (of the set).

Public Operations:

registerWeatherSet (ConcurrentWeatherSet :) :

Adds a ConcurrentWeatherSet pointer to the list of sets the repository will update.

deregisterWeatherSet (ConcurrentWeatherSet :) :

Removes a ConcurrentWeatherSet pointer.

receiveUpdate (WeatherReport :) :

Method to update weather in concurrent weather sets.

1. Iterate through weatherSetPointers
2. Call update methods of ConcurrentWeatherSets

WeatherReportRepository

Central class to hold sets of reports that are either being edited, are pending submission, or have been submitted and are currently being verified. Coordinates and distributes to the user messages from the ReportValidator and ReportVerifier. Provides functions to interate through the various sets, and return specific reports given criteria.

Only a very limited portion of the PreSubmissionReport functions have been implemented for this thesis.

Private Attributes:

activeReportSet : WeatherReportSet

Set of all reports that are "active" and being verified. These reports have been finalized and are un-editable.

preSubmissionReportSet : WeatherReportSet

Set of all reports in "edit mode."

pendingReportSet : WeatherReportSet

Contains all reports qeued for submission before verification. Once reports are verified they are moved to activeReportSet through the supersede method

Public Operations:

activateReport (WeatherReport :) :

Sends report to the pendingReportSet and calls ReportVerifier. If verified properly, calls supersede to move the report to the activeReportSet. If not, calls reeditReport to move it to the preSubmissionReportSet for re-edit.

1. addPendingReport(WeatherReport) - (Add report to the pending report set)
2. removePreSubmissionReport(WeatherReport) - (Remove the report from the preSubmissionReportSet)

getAllActiveReports () :

Returns a set of all active reports (reports in the activeReportSet).

getAllPendingReports () :

Returns a set of all pending reports (reports in the pendingReportSet).

getAllPreSubmissionReports () :

Returns a set of all pre submission reports (reports in the presubmissionReportSet).

receiveMessage (Message : , Priority : , WeatherReport :) :

Not implemented for thie thesis. Message passing is direct from "offending" object to the SystemInterface.

preSubmissionReportSetContainsType (sourceID : , templateName :) : Boolean

Iterates the set and returns true if the preSubmissionReportSet contains a report of the given type and station. False otherwise.

preSubmissionReportSetContains (WeatherReport :) : Boolean

Iterates the set and returns true if the preSubmissionReportSet contains the given report..

pendingReportSetContains (WeatherReport :) : Boolean

Iterates the set and returns true if the pendingReportSet contains the given report.

pendingReportSetContainsType (sourceID : , templateName :) :

Iterates the set and returns true if the pendingReportSet contains a report of the given type and station.

activeReportSetContains (WeatherReport :) :

Iterates the set and returns true if the activeReportSet contains the given report..

activeReportSetContainsType (sourceID : , templateName :) :

Iterates the set and returns true if the activeReportSet contains a report of the given type and station.

receivePendingVerification (WeatherReport : , Message : , Boolean :) :

Not implemented for this thesis.

Receives verification messages regarding the reports queued for dissemination.

sendReport (WeatherReport :) :

Not implemented for this thesis

Interface with World Wide Weather Network and/or database store for active reports. Disseminates globally.

correctActiveReport (WeatherReport :) : WeatherReport

Not implemented in this thesis.

Used to initiate a new weather template for a corrected forecast based on the selected weather report forecast. (Uses identical report values for forecast lines that are still valid at call time).

ammendActiveReport (WeatherReport :) : WeatherReport

Not implemented in this thesis.

Used to initiate a new weather template for an ammended forecast based on the selected weather report forecast. (Uses identical report values for forecast lines that are still valid at call time).

getActiveReport (sourceID : , templateName :) :

Iterates the set and returns an active report with the given sourceID (ICAO) and templateName.

Private Operations:

addActiveReport (WeatherReport :) :

Places report in the activeReportSet.

removeActiveReport (WeatherReport :) :

Removes report from the activeReportSet.

addPreSubmissionReport (WeatherReport :) :

Adds report to preSubmissionReportSet. Ensures report is removed from any other set.

removePreSubmissionReport (WeatherReport :) :

Removes report from the preSubmissionReportSet.

addPendingReport (WeatherReport :) :

Adds a WeatherReport object to the pendingReportSet.

removePendingReport (WeatherReport :) :

Removes report from the pendingReportSet.

reeditReport (WeatherReport : , Message :) :

Sends report from the PendingReportSet to the preSubmissionReportSet.

deactivateReport (WeatherReport :) :

Removes report from the active set. Called once the report is no longer valid (i.e. superseded).

supersede (WeatherReport :) :

Called when a report has been initially verified (cleared to be submitted). Replaces the old active report of the same type/base with the new report. deactivates the old report through call to "deactivateReport." Consistency must be ensured.

ElementIdentifier

Not implemented for this thesis. Instead, this object's function is simulated in the TemplateRepository object with two arrays (TAF and Observation) which contain the list of elementSpecifiers for a particular report.

The "signature" for a particular ReportElement object that will be instantiated at run-time when a WeatherReport is created from a Template. The ReportElement class (and subclasses) provide self-validating user entry forms, associated with attributes of a WeatherReport object. See the descriptions for those objects for more information.

Private Attributes:

elementSpecifier : String

The specifier (which maps to a particular ReportElement).

Public Operations:

getElementProperties () : ElementProperties

Returns the ElementProperties object assoiciated with this element.

InterfaceDisplay

Class to update the user display based on instantiated template configurations. Methods to physically display reports are maintained by the instantiated report template and associated lines and elements. Overall screen/window position, etc. is managed by this class.

Message and report text display, as well as user interface controls to create a report, append a line, quit the program, etc., are language independent. Methods included here must be implemented, but there specific implementation methodology is not elaborated for a generic design.

See the InterfaceDisplayFrame and InterfaceDisplayPanel obejcts of the demonstration associated with this thesis for a sample implementation of display functions using the Java.awt.Frame and Panel components.

Private Attributes:

messageMap : Map(Id, Message)

Set of active messages to display to the user, indexed by id. Suggested implementation: sorted by base, report type and line number. Note that all messages for all active reports should be visible at all times (rather than just the messages for the activeReportTemplate).

Public Operations:

displayMessages () :

Iterate through the collection of Messages and display them for the user.

displayWarning (Message : , WeatherReport :) :

Not implemented in this thesis.

Display a pop-up dialog message to the user.

ackMessage () :

Not implemented for this thesis.

Receive acknowledgement for a particular message so that it won't be displayed again in the future.

ackWarning () :

Not implemented in this thesis.

Receive acknowledgement for a particular pop-up message.

initDisplay () :

Initialize the display based on the template configurations.

addMessage (Message msg :) :

Adds the specified message to the collection of messages.

removeMessage (id :) :

Removes a message from the message map, if it exists.

changeMessageId (String oldKey : , String newKey : , int lineNumber :) :

Method to change the id of a message.

1. Retrieve and remove message from map (using old key).
2. Set the message's line number attribute.
3. Add the message to the map using the new key.
4. Redisplay all messages (displayMessages() method).

Message

Private Attributes:

reportID : WeatherReport

messageText : String

baseICAO : String

Identifier for the base associated with the report generating this message.

severity : String

"Red" or "Yellow"--The severity associated with this message.

lineNumber : int

The line number in the report that this message is assoicated with, if any.

reportType : String

The report type that generated this message.

SourceListener

Class to listen for updates to weather data. When an update is received, the receiveUpdate method is called. The sourceID is compared to the sourceID's of all ConcurretWeatherSet's in the weatherSetPointers set (or the ExternalSourceReportRepository, and if the set contains the ID in question, the set's updateReport() method is called, and the new data passed.

For this thesis, incoming data is assumed to be in the desired format (i.e. already a WeatherReport object). Actual implementation (i.e. database trigger, CORBA interface, etc.) is dependent upon customer (AFW) requirements.

Private Attributes:

portID : String

(Example) The port on which the listener listens for updates.

Public Operations:

receiveUpdate () :

Method called when new data received (ExternalSourceReportRepository.receiveUpdate(WeatherReport))

RuleSetRepository

Not implemented for this thesis.

Container class to maintain distinct rule sets.

Private Attributes:

ruleSets : Set(ReportRuleSet)

The set of rule sets (persistent).

Public Operations:

getRuleSet (ruleSetID :) : ReportRuleSet

Returns the ReportRuleSet (persistently) assigned to a particular report template.

TemplateRuleSet

Private Attributes:

rules : List

ruleSetID : String

Public Operations:

setParamaters (base :) :

templateRuleSet () : TemplateRuleSet

ElementProperties

Properties which apply to a particular element in a line of a weather report (applicable when the template is instantiated as a weather report).

For the implementation for this thesis, this object will be instantiated when the element is instantiated, so the reference to the element will be passed in the constructor.

As designed, the reference to the ReportElement to which these properties will apply when placd in an active template will be passed when the element is instantiated, but the attributes of the element properties will be maintained persistently and associated with the ElementIdentifier object, without a reference to any ReportElement.

Private Attributes:

isMandatory : Boolean = False

Sample element property.

Indicates the element is mandatory (i.e. the report may not be submitted without a valid value for this element).

isSuggested : Boolean = False

Sample element property.

Indicates the element is suggested (i.e. the report may be submitted without a valid value for this element, however the user will be notified that the value should be included before submission is finalized).

firstLineOnly : Boolean = False

Sample element property.

Indicates the element should be hidden from user input for all but the first line of the report.

lastLineOnly : Boolean = False

Sample element property.

Indicates the element should be hidden from user input for all but the last line of the report.

notInLastLine : Boolean = False

notInFirstLine : Boolean = False

inAllLines : Boolean = False

myElement : ReportElement

A back reference to the ReportElement that this object applies to. (NULL until associated with an instantiated ReportElement object).

IN_ALL_LINES : int = 0

Ease of use property setting for constrcutor.

IN_FIRST_LINE_ONLY : int = 1

Ease of use property setting for constrcutor.

IN_LAST_LINE_ONLY : int = 2

Ease of use property setting for constrcutor.

NOT_IN_FIRST_LINE : int = 3

Ease of use property setting for constrcutor.

NOT_IN_LAST_LINE : int = 4

Ease of use property setting for constrcutor.

IS_REPORT_LEVEL_ATTRBIUTE : int = 0

Ease of use property setting for constrcutor.

IS_NOT_REPORT_LEVEL_ATTRIBUTE : int = 1

Ease of use property setting for constrcutor.

IS_REQUIRED : int = 0

Ease of use property setting for constrcutor.

IS_NOT_REQUIRED : int = 1

Ease of use property setting for constrcutor.

Public Operations:

ElementProperties (int lineProperty : , int reportLevel : , int required :) :

Constructor. In the instantiation for this thesis, the ReportElement references are passed as arguments to the constructor. The reason for this is that there is no editor tool to create and maintain elements and properties generically. Were that the case, the ReportElement reference would be set upon association with an instantiated ReportElement.

TemplateLineProperties

Private Attributes:

isRepeatable : Boolean = True

ElementCategorySet

Not implemented for this thesis.

An ElementCategorySet is a set of weather category thresholds for a specific weather element, and is part of the overall weather category schema for a particular base type. For instance, for ceiling:

Cat A : 0-300 feet
Cat B: 300-1500 feet
Cat C: 1500-3000 feet
Cat D: 3000+ feet

A forecast element may be considered valid (i.e. an amendment is not required based on the element in question) if the observed value for the element is in the same range as the forecast value for the element.

Private Attributes:

elementName :

Name associated with the element for which the category thresholds apply. Must match identifier used in the ReportElement class and its sublclasses.

catSet : Set(Category)

Container which holds the individual category values.

negativeValsAllowed : Boolean

Set to true if negative values are allowed for the particular element. If they are allowed, but the actual element can never take on a negative value, the negative-valued categories will be ignored. Likewise, if an element can take on negative values, but this field is set to false, rules associating negative values assumed by the element at run-time will be ignored (verification won't work properly) and the user will be notified to adjust the category set appropriately.

Public Operations:

addCategory (lowVal : , hiVal :) :

Not implemented for this thesis.

Adds a category with the given range. Ranges must be continuous (automatically adjusted. I.E. If initial range of 300-1000 is input, the resulting category set will be:

A: 0-300
B: 300-1000
C: 1000+

Subsequent ranges will be handled similarly.)

Sufficient "intuitive" interface routines should be implemented to allow the user to specify categories as desired.

removeCategory (catID :) :

Removes the specified category threshold criteria. Adjusts category identifiers as appropriate. I.E. if current categories were:

A: 0-300
B: 300-1000
C: 1000-3000
D: 3000+

and category B were removed, the resulting set would be:

A: 0-1000
B: 1000-3000
C: 3000+

Note if category D were "removed" the automatic adjustment would add it back because the ranges must be continuous and cover the entire range.

Category

Not implemented for this thesis.

A specific threshold category for a particular element.

Private Attributes:

catID :

One-up (alphabetic) identifier for categories. Derived automatically when categories are added or deleted.

lowVal : float

The low value of the range for a particular category.

hiVal : float

The high value for the particular category.

BaseProperties

Partially implemented for this thesis.

Properties associated with a base (1-to-1 relationship) are any and all base-specific meteorological data that would be useful in generating forecaster decision-aid rules. Only a few possible exmples are shown here.

Functions to assign a BaseWeatherStation BaseProperties class to a particular station based on another station (copy) should be included in the ConfigurationEditor.

Private Attributes:

verifSfcStationSourceID : SourceID

The unique "SourceID" of the surface report associated with the BaseWeatherStation. For this implementation, the International Civil Aviation Organization (ICAO) code was used to identify stations.

verifUAStationSourceID : SourceID

The unique "SourceID" of the upper-air report associated with the BaseWeatherStation. For this implementation, the International Civil Aviation Organization (ICAO) code was used to identify stations.

alternateSourceStationSet : Set(SourceID)

Not implemented in this thesis

List of alternate "SourceIDs" of other location reports that might be used as data for agent rule-satisfication.

lowTempJan : TempF

Sample base-specific meteorological data.

highTempJan : TempF

Sample base-specific meteorological data.

avgWindDirJan : WindDirection

Sample base-specific meteorological data.

avgWindSpeedJan : WindSpeed

Sample base-specific meteorological data.

StationSet

Private Attributes:

stationSetID :

stations : Set

RuleEditor

Not implemented for this thesis.

Interface application to add, delete, and modify rules in particular rule sets.

Public Operations:

initEditor () :

Method to instantiate the RuleSetRepository and (possibly) load all ReportRuleSets from persistent storage.

getRuleSet (ruleSetID :) :

DistinctTemplateLine

Not implemented for this thesis.

A distinct line (container) which holds a set of elementIdentifiers--"signatures" for ReportElement objects (see ElementIdentifier description).

In a TAF or an Observation, there is only one "distinct" line (though the line configuration may be repeated).

Circumstances which would require more than one line would be, for instance, a report in which specific, verifiable forecast element information is given, followed by a narrative (descriptive, non-verifiable) summary paragraph. The verifiable information could be one line, and the narrative could be another. The line pairs could then be repeated again and again in an instantiation of a weather report, each pair valid at a different time.

For the purposes of this thesis, only one distinct template line will be implemented to accomodate TAFs and Obs.

Private Attributes:

elements : Set(ElementIdentifier)

The set of element identifiers ("signatures") for the particular template line.

lineID : int

Identifier for the line within the template. Use 1-up integer.

Public Operations:

getLineProperties () : LineProperties

Returns the LineProperties object associated with this line.

ReportProperties

Properties which apply to the entire weather report (applicable when it is instantiated as a weather report).

Private Attributes:

validDurationHours : int

Duration for which the report (when instantiated) is valid. I.E. a TAF would be 24-hours, while an Observation is unspecifed.

issueHours : int[]

Array containing hours this report is to be issued. If null, report assumed to be either on the next hour or at creation time, depending on context.

temporaryConditionValidDurationMins : int

Number of minutes a temporary condition is valid before the report must be ammended.

corEligibilityMaxMins : int

Number of minutes (if applicable) after which a correction is not allowed.

reportType : String

Type of report.

isForecast : Boolean

If true, indicates that the template is a forecast report (which can be verified). If not, it is considered to be an observation (which can be used to verify a forecast).

isAmmendment : Boolean

Indicates that this forecast is an ammedment.

isCorrection : Boolean

If true, indicates that this forecast is a correction.

isNormal : Boolean = True

If true, indicates that the report is one generated at the normal issue time (given by issueHours above).

If false, indicates a "special" report, generated for some other reason. Ammended reports, correct reports, and SPECI observations are exmples of reports that would warrant a false value for the isNormal attribute.

templateName : String

The name of the template upon which this report was based.

hoursOfObsToKeep : int

Number of hours of verifying observations to keep for this report.

Previous observations can be used to derive element change (i.e. pressure, temperature, etc) information.

numberOfSoundingsToKeep : int

Indicates the number of sounding reports (previous to this one) to associate with this report.

categorySet : WeatherCategorySet = NULL

Category set used to verify forecasts. Copied from persistent storage (maintained by ConfigurationEditor, not implemented for this thesis).

reportRuleSetDefaultID : String

ID of reportRuleSet associated with this report. As implemented in this thesis, the rule set cannot be changed at run-time. Later implementations, however, should consider allowing run-time setting of different rule sets for experimentation purposes.

template : Template

The template associated with this report.

weatherStation : BaseWeatherStation

The BaseWeatherStation object associated with this report.

ruleSet : ReportRuleSet

The ReportRuleSet object associated with this report.

String DELIMS : = "#"

Delimeter used for text report output formatting.

String NULL_SPACE_DELIMS : = "^"

Null space delimeter used for text report output formatting.

reportText : String

The formatted text of this report.

myInterface : SystemInterface

A local reference to the SystemInterface for text report updates.

textFormat : HashMap

A HashMap containing the format for report text output specific to this report type. See updateReportText method

Public Operations:

ReportProperties (ReportProperties :) :

Not implelemted for this thesis.

Creates a new ReportProperties object based on an existing one (copy).

(Sets all attributes equal to those of the object passed)

updateReportText () :

Format the report text using element values and element-specific formatting in the textFormat HashMap.

Iterate through all lines in the report.
For each line, iterate through all elements.
If the value of the element is not null:
  If the element name is a key in the HashMap, format text according the the map, and concatinate.
  Otherwise, concatinate the value of the element.
Concatinate a return character at the end of each line.
set the reportText to the concatinated string.

Element appears in string wherever the DELIMS appears.
Space is NOT concatinated at end of string if NULL_SPACE_DELIMS is at the end of textFormat entry.

HashMap formatting example:
HashMap entry: "some text # some other text"
Output: "some text <elementValue> some other text "
(Space at the end of the String)

HashMap formatting example:
HashMap entry: "some text # some other text^"
Output: "some text <elementValue> some other text"
(No space at the end of the String)

LineProperties

Properties which apply to a particular line in a weather report (applicable when the template is instantiated as a weather report).

Private Attributes:

isRepeatable : Boolean = True

Sample line property.

Indicates the line may be repeated any number of times when instantiated in a weather report.

maxLines : int

Sample line property.

Indicates the maximum number of lines of its type allowed in a particular instance of the weather report.

ExpertSystemShellAgent

An agent object to communicate between the SystemInterface and the employed expert system shell.

Note that in the implementation demonstration associated with this thesis, this object is not required. All functions regarding object registration are carried out by the SystemInterface. Registration of objects creates a binding between the object and JESS using JavaBeans properties (property change listeners) so that attribute changes are directly reflected in corresponding JESS facts).

If another implementation method was used, a separate agent to explicitly update the employed expert system shell may be required.

Public Operations:

update (Object o :) :

Method to update the shell with changes to the given object.

remove (Object o :) :

Method to remove an instance of an object binding between the shell and the system.

add (Object o :) :

Method to add an instance of an object binding to the shell.

generateWarning () :

Method to generate a warning based on rule firings.

retractWarning () :

Method to retract warnings based on rule firings.

initialize () :

Method to initialize system.

addRule (Rule r :) :

Method to add a rule to the system.

retractRule (Rule r :) :

Method to retract a rule from the system.

WeatherCategorySetManager

Not implemented for this thesis.

Container class to hold various WeatherCategorySets that may be associated with a BaseType at run-time. Sets will be persistent

Public Operations:

initManager () :

Method to load persistently stored WeatherCategorySets for editing.

SystemMaintenanceInterface

Not implemented for this thesis.

Central system interface to maintain base information, base properties, category schemas, validation and verification rules, and report templates.

Public Operations:

initInterface () :

Method to instantiate the ConfigurationEditor, the RuleEditor, and the TemplateEditor objects.

getTemplate (templateName :) : Template

getRuleSet (ruleSetID :) : ReportRuleSet

getStation (sourceID :) : BaseWeatherStation

getCategorySet (BaseWeatherStation :) : WeatherCategorySet

SoundingCondition

Not implemented for this thesis.

Sounding-specific information.

Derived from UpperAirCondition

Private Attributes:

soundingData : String

The sounding data code for this level/time (from which the other attributes may have been derived).

PirepCondition

Not implemented for this thesis.

PIREP-specific information.

Derived from UpperAirCondition

Private Attributes:

aircraftID : String

Identifier for the aircraft reporting the PIREP.

visibility : VisSM

Visibility reported.

UpperAirReport

Not implemented for this thesis.

Derived from WeatherReport

Private Attributes:

liftingCondensationLevel : AtmosphericHeight

Derived lifting condensation level for the valid time for the report.

WeatherReportTemplateSet

Set containing WeatherReportTemplates. For this thesis, implemented as a HashSet (subclass of Java.util.HashSet).

Public Operations:

containsType (BaseWeatherStation base : , String reportType :) : Boolean

Queries the set and returns true if it contains a report of the given type for the given base.

addTemplate (WeatherReportTemplate :) :

Adds a WeatherReportTemplate object to the set.

removeTemplate (WeatherReportTemplate :) : Boolean

Removes the given WeatherReportTemplate from the set, if it exists. Returns False if none were removed.

getTemplate (String id :) :

Not implemented for this thesis.

Iterates the set of templates, and returns the WeatherReportTemplate with the specified id if it exists.

LineList

List containing WeatherReportTemplateLines. For this thesis, implemented as a LinkedList (subclass of Java.util.LinkedList).

Public Operations:

getLastLine () : WeatherReportTemplateLine

Returns the last line in the list.

getLineAtIndex (int index :) : WeatherReportTemplateLine

Returns the WeatherReportTemplateLine at the specified index number.

deleteLineAtIndex (int index :) : Boolean

Removes the line at the given index. Calls each displaced line's decrementIndex method so that line numbers can be adjusted appropriately. Also calls the "updateElementOrder()" method for each line's ElementList to ensure all GUI components are in order (since some may have moved after the deletion)

insertLineAtIndext (int index : , WeatherReportTemplateLine line :) :

Insertsthe given line at the given index. Calls each displaced line's incrementIndex method so that line numbers can be adjusted appropriately. Also calls the "updateElementOrder()" method for each line's ElementList to ensure all GUI components are in order (since some may have moved after the insertion).

updateLineOrder () :

Method to ensure that GUI components associated with each line are displayed in the correct order as indicated by the line list.

updateElementList () :

Method to iterate through each line call the associated ElementList's updateElement() method to ensure that the GUI component associated with each element is displayed in the order indicate by the ElementList

ReportElementSubClass

This is an example of subclass to a ReportElement object. Many such elements have been implemented for this thesis and should be referenced as examples.

These elements must be self-sufficient in that:

1. They declare which attributes, if any, of the WeatherCondition and WeatherReport objects they modify.
2. They can parse/decode user input.
3. They can verify that user input is formatted properly (format, range, etc.)
4. In their verify methods, they must iterate through the myConditionAttributes and myReportAttributes AttributeValuePari HashSets of the ReportElement superclass and provide String-type values to the AttributeValuePairs associated with every attribute of the WeatherCondition or WeatherReport object they modify, respecitively.

There can be any number of subclasses to the ReportElement available to the system. Upon instantiation in a WeatherReportTemplateLine of a WeatherReportTemplate, however, these elements must register (ReportElement.register()) such that no two elements in the same report modify the same WeatherCondition or WeatherReport attribute.

Derived from ReportElement

Private Attributes:

modCondAttribs : String[]

Names of the attributes of the WeatherCondition that this element modifies.

modRepAttribs : String[]

Names of the attributes of the WeatherReport that this element modifies.

Public Operations:

ReportElementSubClass () :

Constructor.

Pass element name, field width and properties to the superclass ReportElement. ReportElement derives from the ElementProperties the type of element (lable, choice or text box), and instantiates local attributes as needed.

runRegistrationTasks () :

Method to format the element value prior to element registration. This is useful if, for instance, the element's value is formatted automatically (i.e. based on current time, or solely on some ReportProperties attribute).

verify () :

Method called when user inputs data into this element.

1. Retrieve elementValue from superclass.
2. Parse elementValue String and perform necessaryh conversion (to integer or float, etc.) to verify range values and format).

3. Call "generateWarning" method of ReportElement giving warning type (ReportElement.REQUIRED, RANGE, FORMAT, OTHER) if the user input is formatted improperly.
4. Call "retractWarning" method of RerportElement if user inpur is good.
5. Iterate through ReportElement superclass myReportAttributes and myConditionAttributes AttributeValuePairs HashSets and provide values for each attribute modified based on user input.

getModCondAttribs () :

Returns the modCondAttribs array -- implements ReportElement abstract class.

getModRepAttribs () :

Returns the modRepAttribs array -- implements ReportElement abstract class.

WeatherConditionSet

Public Operations:

addWeatherCondition (WeatherCondition wc :) :

removeWeatherCondition (WeatherCondition wc :) :

updateRefereces () :

Method to update the nextCondition and previousCondition references of all WeatherCondition objects.

WeatherReportSet

Set (implemented as a HashSet) to contain WeatherReport objects.

Public Operations:

addReport (WeatherReport wr :) :

Method to add a report to the set.

removeReport (WeatherReport wr :) :

containsType (BaseWeatherStation base : , String reportType :) :

Method to query whether a specific type of report exists in the set for a specific base.

ElementList

List to hold GUI ReportElements. Implemented as a LinkedList for this thesis.

Public Operations:

addElement (ReportElement re :) :

Adds an element to the end of the list.

removeAll () :

Removes all elements from the list.

ReportElementRegistrationManager

Object through which ReportElements contained in WeatherReportTemplateLines can be registered to modify attributes of WeatherReport and WeatherCondition objects.

Necessary so that a report template cannot be constructed such that two elements can modify the same attribute of either of those objects.

Public Operations:

ReportElementRegistrationManager (WeatherCondition wc :) :

Constructor.

1. Set local reference to WeatherCondition object.
2. reportAttributes = WeatherCondition.getMyWeatherReport().getAttributesSet() - (Get local copy of the attributesSet for the WeatherReport corresponding to the associated WeatherCondition.
3. conditionAttributes = WeatherCondition.getAttributesSet() - (Get local copy of the attributesSet for the associated WeatherCondition.

registerElement (ReportElement re : , String[] modifiableRepAttribs : , String[] modifiableCondAttribs :) : Boolean

Method to register a report element given the arrays containing the names of the WeatherReport and WeatherCondition objects' attributes the element will be modifying.

1. Verify the registration eligibility of the report element, given the attribute names supplied. Return false if another element has already registered for those attributes.
2. Add the report element to the conditionRegistrationMap and the reportRegistrationMap, indexed by all of the attribute names of the attributes the element will modify.
4. Return true.

deregisterElement (ReportElement re :) : Boolean

Method to deregister a ReportElement.

Iterate the element references in the reportRegistrationMap and the conditionRegistrationMap objects and remove any objects with the reference.

verifyRegistration (ReportElement re : , String[] modifiedRepAttrbis : , String[] modifiedCondAttrbis :) : Boolean

Method to verify that another element hasn't already registered to modify the attributes this element is registering for.

Iterate through the reportAttributeMap and conditionAttributeMap objects and verify that none of the attributes listed in the passed String arrays are contained in the maps. If they are, return false. If not, return true.

isWeatherConditionUpdatePermitted (String attributeName : , ReportElement re :) : Boolean

Method to determine the ReportElement's permission to update the attributeName.

If the ReportElement is contained in the conditionAttributesMap indexed by attributeName, return true. Otherwise return false.

isWeatherReportUpdatePermitted (String attrbuteName : , ReportElement re :) : Boolean

Method to determine the ReportElement's permission to update the attributeName.

If the ReportElement is contained in the reportAttributesMap indexed by attributeName, return true. Otherwise return false.

Bibliography

[1] Air Force Weather Agency. Observer. Vol.44, No.7. Offutt AFB, NE: HQ AFWA, September/October 1997.

[2] Association for Computing Machinery, Special Interest Group for Computer-Human Interaction Home Page. http://www.acm.org/sigchi

[3] Birnbaum, Larry, et al. "Compelling User Interfaces—How Much AI? (Position Statements)" in Proceedings of the International Conference of Intelligent User Interfaces (IUI97): 173-175. Orlando, FL: The Association for Computing Machinery (ACM), 1997.

[4] Chan, Patrick, et al. The Java Class Libraries Second Edition, Volume 1. Reading, Massachusetts: Addison-Wesley, 1998.

[5] Chan, Patrick, et al. The Java Class Libraries Second Edition, Volume 2. Reading, Massachusetts: Addison-Wesley, 1998.

[6] Chan, Patrick, et al. The Java Class Libraries Second Edition, Volume 1, Supplement for the Java 2 Platform Standard Edition, v1.2. Reading, Massachusetts: Addison-Wesley, 1999.

[7] Department of Commerce. Surface Weather Observations and Reports. FMH-1. Washington: December 1995. http://www.nws.noaa.gov/oso/oso1/oso12/fmh1.htm

[8] Department of the Air Force. Surface Weather Observations. AFMAN 15-111. Washington: HQ USAF, 1 November 1998.

[9] Department of the Air Force. Weather Support Evaluation. AFI 15-114. Washington: HQ USAF, 19 January 1994.

[10] Donahue, Capt Christopher A. TAFVER II Users Manual. (USAFETAC/TN-93/003). Scott AFB, IL: USAF Environmental Technical Applications Center, May 1993.

[11] Dryer, D. Christopher. "Wizards, Guides, and Beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents," in Proceedings of the International Conference of Intelligent User Interfaces (IUI97): 265-268. Orlando, FL: The Association for Computing Machinery (ACM), 1997.

[12] Duffield, George. TAFVER IV-A Training PowerPoint Presentation. Scott AFB, IL: Sumaria Systems, Inc, May 1998.

[13] Eisenberg, Michael and Fischer, Gerhard. "Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance," in Conference Proceedings on Human Factors in Computing

Systems (CHI 94): 431-437. Boston, MA, The Association for Computing Machinery (ACM), 1994.

[14] Electronic Systems Center. Acquisition Meteorology Home Page. http://www.hanscom.af.mil/esc-bp/axw/esc_axw.htm.

[15] Electronic Systems Center. Air Force Weather Systems. System Requirements Document (SRD) for the Reengineered Air Force Weather Weapon System (AFWWS). Release 2.2. Hanscom AFB, MA: Electronic Systems Center, Air Force Weather Systems (ESC/ACW), July 1999.

[16] Electronic Systems Center. Program Development Plan for the Operational Weather Squadron Production System II (OPS II) Program. Hanscom AFB, MA: Electronic Systems Center, Air Force Weather Systems (ESC/ACW), May 1999.

[17] Electronic Systems Center. Program Development Plan for the Weather Information Production System—Upgrade (WIPS-U) Program. Hanscom AFB, MA: Electronic Systems Center, Air Force Weather Systems (ESC/ACW), April 1999.

[18] Elliot, George. Weather Forecasting Rules, Techniques and Procedures. Boston, MA: American Press, 1988.

[19] Ernest J. Friedman-Hill, Jess, The Java Expert System Shell Version 4.4 (User's Manual). Livermore, CA: Distributed Computing Systems, Sandia National Laboratories, Jan 15, 1999. http://herzberg.ca.sandia.gov/jess

[20] Fischer, G., et al. "The Role of Critiquing in Cooperative Problem Solving," ACM Transactions on Information Systems, 9: 123-151. (April 1991).

[21] Forgy, Charles L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, 19: 17-37 (September 1982).

[22] Garrison, David. "The Air Force Global Weather Central Verification Program," in Colloquium on Weather Forecasting and Weather Forecasts: Models Systems and Users. Vol. 2, pp. 886-890. Boulder, CO: NCAR/CQ-5, Summer 1976.

[23] Giarratano, Joseph C. and Gary Riley. Expert systems: Principles and Programming. Boston, MA: PWS-Kent Publishing Company, 1989.

[24] Gonzalez, Avelino J. and Douglas D. Dankel. The Engineering of Knoledge-Based Systems. Englewood Cliffs, NJ: Prentice Hall, 1993.

[25] Hoque, Reaz. Programming JavaBeans 1.1. New York: McGraw-Hill, 1998.

[26] Jamison, Sherwin W. TAF Verification. (AFGWC/TN-86/002). Offutt AFB, NE: HQ Air Weather Service, Air Force Global Weather Central, Nov 1986.

[27] Joga, Meriellen. Data Warehouse Techniques to Support Global On-Demand Weather Forecast Metrics. MS Thesis, AFIT/GCS/ENG/00M-09. School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 2000.

[28] Lieberman, Henry, et al. "Panel II. IUI and Agents for the New Millennium, " in Proceedings of the International Conference of Intelligent User Interfaces (IUI99): 93-94. Redondo Beach, CA: The Association for Computing Machinery (ACM), 1999.

[29] Lieberman, Henry. "Autonomous Interface Agents," in Conference Proceedings on Human Factors in Computing Systems (CHI 97): 67-74. Atlanta, GA: The Association for Computing Machinery (ACM), 1997.

[30] Maybury, M "Intelligent User Interfaces: An Introduction," in Proceedings of the International Conference of Intelligent User Interfaces (IUI99). Redondo Beach, CA: The Association for Computing Machinery (ACM), 1999.

[31] Muller, Pierre-Alain. Instant UML. Birmingham, The United Kingdom: Wrox Press Ltd., 1997.

[32] Riley, Gary. C Language Integrated Production System Home Page http://www.ghg.net/clips/CLIPS.html.

[33] Telfer, Ray T. and Randall C Webb. Terminal Forecast Verification. (Technical Note #70-2). Offutt AFB, NE: Scientific Services Branch, Aerospace Sciences Division, 3d Weather Wing, Feb 1970.

[34] Trigg, Jimmie. A Study of Morning Radiation Fog Formation. MS Thesis, AFIT/GM/00M-14. School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 2000.

Curriculum Vitae

Captain Darryl N. Leon was born in Paterson, New Jersey, but raised from a very early age in Miami, Florida. After his 1984 graduation from Ransom-Everglades High School in Coconut Grove, Capt Leon enlisted in the Air Force and became a Chinese Cryptologic Linguist.

A Distinguished Graduate from the Chinese language department of the Defense Language Institute in Monterey, California, Capt Leon spent his enlisted career first at Kadena Air Base, Okinawa, Japan, as a crewmember aboard Air Force RC-135 aircraft, and later at Wheeler Air Force Base (now Army Air Station), Hawaii. It was in Hawaii that Capt Leon met his wife, Keri, who was also enlisted at the time.

In 1991, as a Staff Sergeant, Capt Leon was accepted into the Airman's Education Commissioning Program. In 1993, under this program, he was graduated Magna Cum Laude from Florida State University, with a Bachelor of Science in Meteorology and minors in Physics, Mathematics, and Computer Science.

A Distinguished Graduate of Officer Training School in Montgomery, Alabama, then Second Lieutenant Leon became Wing Weather Officer, and later interim Flight Commander, of the 314th Airlift Wing's Weather Flight at Little Rock Air Force Base, Arkansas.

In 1996 Capt Leon was assigned to the Air Force Operations Center at the Pentagon in Washington DC, where, in 1997, he was named Air Force Weather's Company Grade Officer of the Year.

Two years later, in August, 1998 (a month after the birth of his daughter, Dakota), Capt Leon entered the Graduate School of Engineering and Management at the Air Force Institute of Technology in Dayton, Ohio. Upon graduation, he will be assigned to the Air Force Weather Agency at Offutt Air Force Base, Nebraska.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
AN INTELLIGENT USER INTERFACE TO SUPPORT AIR FORCE WEATHER PRODUCT GENERATION AND AUTOMATED METRICS

5. FUNDING NUMBERS

6. AUTHOR(S)

Darryl N. Leon, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/00M-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ AFWA/XP
Attn: Mr. George Coleman
106 Peacekeeper Drive, Suite 2N3
Offutt AFB, NE 68113-4039

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Major Michael L. Talbert, ENG, DSN: 785-3636, ext. 4280

12a. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Air Force pilots require dependable weather reports so they may avoid unsafe flying conditions. In order to better gauge the accuracy of its weather products, Air Force Weather has established the requirement for an Air Force-wide automated weather metrics program. Under the guidelines for this program, forecasts will automatically be compared to observed weather to determine their accuracy. Statistics will be collected in the hopes of determining forecast error trends that can be corrected through education and training. In order for the statistical data produced by such a program to draw reliable conclusions about forecast accuracy, however, the correct format of the raw forecasts and observations must be ensured before the reports are disseminated.

Beyond a simple check for typographical errors, however, the system must also have weather domain knowledge to understand when the input data content does not fit the context of the report, even though it has been formatted properly. This thesis proposes the application of an intelligent user-interface "critic" advice system, to ensure not only correct product format and provide content quality control, but to collaborate with and advise the forecaster or observer during product generation, with the ultimate goal of producing more accurate weather products.

14. SUBJECT TERMS

Intelligent User Interface, Critic Systems, Weather Forecasting, Weather Forecast Verification, Automated Verification, Weather Forecast Metrics, Automated Metrics

15. NUMBER OF PAGES 212

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	**PR** - Project
G - Grant	**TA** - Task
PE - Program Element	**WU** - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE - See authorities.
NASA - See Handbook NHB 2200.2.
NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.
DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA - Leave blank.
NTIS - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.