Collaborative Arrival Planner:
Its Design and Analysis Using Object Modelling

by

David C. Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2000, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis describes the design, analysis, and implementation of a client/server ar-
chitecture for an infrastructural software system. The system designed is a prototype
of the Collaborative Arrival Planner (CAP), a new component being considered by
NASA to aid the exchange of information between the Center/TRACON Automation
System (CTAS) and external parties.

The CAP server receives continuous flight status information from a feed provided
by the Volpe National Transportation Center. For the clients, the CAP server provides
predicated filtering, a method by which the clients can specify relevance of certain
flights, and receive only those flight information from the server.

Alloy, an object modelling language similar to UML and Z, and its analysis tool
Alcoa were used to aid the design and analysis of the CAP prototype. Critical portions
of the CAP architecture was modeled in Alloy. These models were analyzed in Alcoa
to find flaws, and validate properties of the design. We use CAP as an example of
infrastructural software, and show that object modelling techniques can be efficiently
applied to help the design and analysis of real world software systems.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor
Acknowledgments

Many people contributed to this thesis project. I would like to thank my advisor Professor Jackson for teaching me the way of object modelling; Professor Gifford for his insights on predicated filtering; and Qian Wang for his help on the implementation and understanding the ASDI feed.

To Christine, thanks for always believing in me.

I dedicate this work to my family. Dad, thanks for all the advice you have given me. I hope you never stop giving them. Mom, thanks for showing me how to be strong. Jane, thanks for always watching out for me.
Contents

1 Introduction 10
  1.1 Background .................................................... 11
  1.2 Problem Statement ........................................... 12
  1.3 Motivation ....................................................... 14

2 Functionality 15
  2.1 Authentication ................................................. 16
  2.2 Predication ....................................................... 17
  2.3 Templates ....................................................... 17
  2.4 Persistence ....................................................... 19

3 Design and Analysis 20
  3.1 Registration .................................................... 20
  3.2 Flight Record .................................................. 24
  3.3 Predicate Matching ........................................... 25
  3.4 Template ........................................................ 27
  3.5 State Database ............................................... 31
  3.6 Flight ID Amendment ......................................... 32
    3.6.1 Templates and Flight ID Amendment .................. 32
    3.6.2 State Database and Flight ID Amendment .......... 34
  3.7 Summary ......................................................... 37

4 Implementation 39
<table>
<thead>
<tr>
<th></th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2</td>
<td>DumbClient.java</td>
<td>74</td>
</tr>
<tr>
<td>D.3</td>
<td>distributor package</td>
<td>76</td>
</tr>
<tr>
<td>D.4</td>
<td>parser package</td>
<td>78</td>
</tr>
<tr>
<td>D.5</td>
<td>predicater package</td>
<td>100</td>
</tr>
<tr>
<td>D.6</td>
<td>registrar package</td>
<td>109</td>
</tr>
<tr>
<td>D.7</td>
<td>util package</td>
<td>118</td>
</tr>
<tr>
<td>D.8</td>
<td>xml package</td>
<td>127</td>
</tr>
</tbody>
</table>
## List of Figures

2-1  Data Flow Paths in CAP ................................................. 16
3-1  Model describing Registration and Policy .......................... 21
3-2  Model for Flight Record Data Structure ............................ 23
3-3  Model Relating Predicate and Flight Record ....................... 25
3-4  Model for Predicate Matching with Naive Matching Method ....... 26
3-5  Model for Templates that Violates Flight Record Invariant ....... 28
3-6  Partial Model for Flight Database Operations ..................... 30
3-7  Partial Model for Flight Updates and Templates ................. 33
3-8  Model for Flight Update .................................................. 34
3-9  Partial Model for Flight Updates and State Database ............ 35
3-10 Complete Object Model for CAP ........................................ 37
4-1  CAP Components and Data Flow Paths .............................. 40
4-2  ASDI Message Hierarchy .................................................. 41
4-3  Flight Update and Flight Record MDD ................................ 43
4-4  User Database Format ..................................................... 46
B-1  CAP Components .......................................................... 65
B-2  Flight Record Fields ...................................................... 66
C-1  CAP Component Dependencies .......................................... 67
C-2  CAP Server Modules ....................................................... 68
C-3  Flight Record and Fields ................................................ 69
C-4  Parser Modules ............................................................ 69
| C-5 | Distributor Modules | 69 |
| C-6 | Predicater Modules  | 70 |
| C-7 | Registrar Modules   | 70 |
List of Tables

4.1 ASDI Message Types ................................................. 41
4.2 Flight Record Fields and Field Types ............................ 43
4.3 Field Types and Operators .......................................... 47
Chapter 1

Introduction

The Collaborative Arrival Planner (CAP) is an intermediate software module to aid the exchange of information between the Center/TRACON Automation System (CTAS) and external parties (e.g. airlines, FAA, researchers). The architecture of CAP was designed and analyzed using Alloy, an object modelling language that has precise semantics so its models are analyzable by software. A prototype of CAP was implemented using the Java programming language.

This chapter briefly describes the backgrounds on CTAS, CAP and Alloy, then sets the scope of this thesis.

Chapter 2 describes the functionalities provided by the CAP server, and its relation with ASDI.

Chapter 3 describes the design of the CAP using object modelling notations from the language Alloy, along with analysis of the design.

Chapter 4 describes the architecture of the prototype implementation of CAP that we developed, and its fulfillment of the design.

Chapter 5 concludes this paper by describing the experience of developing CAP using object modelling techniques, reviews the design and the decisions made, and mentions some extensions to CAP that are being considered.
1.1 Background

The first goal of air traffic control is to prevent collision by maintaining safe separations among aircrafts. Separation is provided by establishing approved longitudinal, lateral, or vertical distance between aircrafts. This is no easy task near large airports, where there are frequent landings and take-offs, and the concentration of aircrafts is dense.

The second goal of air traffic control is to make efficient use of the air space without sacrificing the first goal. In many countries including the United States, the rate at which aircraft can land at airports is the limiting factor in air traffic flow [DEGN91].

The Center/TRACON Automation System (CTAS) is a suite of air traffic control decision-support tools developed by NASA Ames Research Center to help air traffic controllers to manage air traffic flows efficiently.

The purpose of CTAS is to increase the landing rate at large airports. On a high level, CTAS receives aircraft arrival data that are within the Center or TRACON air space, along with other considerations such as weather conditions, airport configurations, and outputs a sequence of scheduling recommendations to air traffic controllers. After demonstrating a 10% improvement in the sustained landing rate at Dallas/Fort Worth (DFW), CTAS has been selected by the FAA for national deployment.

The Collaborative Arrival Planner (CAP) is a software system currently under development, to aid the exchange of information between CTAS and external parties such as airlines and FAA. The goal of CAP is to prevent arrival timing miscues by facilitating communication of arrival information between CTAS and airlines, consequently increasing airline efficiencies.

The preliminary requirements for CAP are described in more detail in [Fan]. In brief, CAP provides two major functions. First, it serves as a intermediary to facilitate communication between CTAS and airline dispatchers. CAP provides flight arrival information from CTAS to airline personnel, and accepts rescheduling requests back from airlines.

The secondary function of CAP is to relay aircraft status information among
CTAS installations. Currently CTAS installations are isolated from each other, but their functionalities can be improved by obtaining air traffic predictions from the surrounding Centers. By creating a network of CTAS using CAP, each CTAS can obtain more accurate predictions of flight status and thereby make better recommendations to the controllers.

The design of CTAS was the focus of a study in software design by the Software Design Group in the MIT Lab for Computer Science [JC00]. In this study, Daniel Jackson and others took apart CTAS, and make it dramatically simpler by using a number of software engineering techniques. These traditional techniques were so effective, that they did not employ advanced techniques such as object modelling. This thesis is a follow-on to the CTAS redesign project. Instead of re-engineering a system like CTAS, We apply object modelling techniques right from the start of the design of CAP. Our motivation is discussed later in this chapter.

1.2 Problem Statement

The use of CAP to network CTAS installations is not within the scope of this project, as we believe it requires a completely orthogonal design from its first functionality. CAP’s two functional requirements are better served by separate systems, because the combination of the two functionality in a single system leads to undesirable level of complexity.

The focus of this paper is on a preliminary design of CAP limited to providing filtered flight information from CTAS to airlines, and related issues such as authentication, registration, and distribution. While this is a limited design, it serves as a basis for additional functionality in the future.

The problem, now redefined, is to design and implement a prototype system for efficient distribution of real-time flight information to relevant clients, where relevance is specified by the client. Irrelevant flight information are filtered out by the server, and never sent to the client. So bandwidth is used efficiently. In addition, the system should be flexible in supporting definition of relevance by the client. For example, a
client should be able to specify a region of air space, then receive flight information on all flights entering and leaving that space.

We have many possible applications in mind for this system. Air traffic controllers can use it for monitoring the flights arriving at or leaving an airport. Researchers can record real flight traffic data for their studies, and avoid the task of filtering the data themselves. Displaying flight routes of several airlines is also a convenient application.

Instead of receiving data from CTAS installations, our prototype system will be using the Aircraft Situation Display To Industry (ASDI) data feed from the Volpe Center as the source for flight information. The ASDI feed provides real-time air traffic data from the National Airspace System (NAS), which includes flight plans, routes, positional and status information. This feed provides data similar to that needed by CAP, and is easily accessible by members of the aviation industry. Technical details of the ASDI feed can be found in [Vol99].

Take note that the data sent over the ASDI feed has been filtered. For example, military flights and other sensitive flights (such as flights of the Drug Enforcement Agency) are removed from the data stream; also, messages from certain facilities are sometimes removed. Consult the reference [Vol99], section 3.1, for more information on the filtering process.

We implement a prototype of this system using the Java programming language. We also use the object modelling language Alloy and its companion analysis tool Alcoa, to design and analyze the architecture of CAP.

Alloy is a lightweight, precise and tractable notation for object modelling [Jac99]. It is lightweight because it has a simple and small syntax that is expressive enough for most object modelling problems. It is precise because the semantics of the language is based on sets and relations. It is tractable so its models can be efficiently and fully analyzed semantically. Alloy is influenced by the modelling language UML and the specification language Z.

Alcoa is a tool for analyzing object models written in Alloy [Dan00]. It works by translating the constraints of a model to boolean formulas, and then applying a SAT (satisfiability) solver to check the satisfaction of the original constraints. Using the
solution of the solver, Alcoa can output instances of the model that satisfy a set of constraints, or counter-examples that violate some constraints.

1.3 Motivation

There are two motivations behind this thesis. The first one is practical. The collaborative arrival planner is a real system being considered by NASA as a part of a major effort to upgrade and modernize the air traffic control system in the U.S. With this thesis, we help design the architecture of CAP and provide a prototype with majority of the functionality.

The second one is to evaluate the methodology of software design assisted by an object modelling language that is checkable. Software systems are being used in more and more aspects of our lives, such as banking, utility, and traffic control. These infrastructural software systems are large and complex. Their designs are difficult and time consuming using traditional techniques. Their implementation stages are often plagued by discovery details not accounted for in the initial design, thus delaying delivery and increasing cost.

Alloy and Alcoa are designed together to help tackling these problems from the design stage. Alloy provides a precise language for expressing designs and specifying their behavior. Alcoa is a model checker that can find flaws in Alloy models. Together, they have been used to analyze several existing designs of software systems in different domains. But they have not been used in formulating the design of a new system. With this thesis, we wish to demonstrate, by using CAP as an example, that an object modelling language such as Alloy can be easily applied to real world system design. Object modelling techniques, combined with static analysis at the semantic level, improves traditional software methods.
Chapter 2

Functionality

This chapter describes the functionalities provided by the CAP server, and its relation with the Aircraft Situation Display To Industry (ASDI).

CAP uses a client-server model that is common to many infrastructural software systems today, as shown in Figure 2-1. The input to CAP is the ASDI flight information feed. From ASDI, CAP receives actual and projected status information about all flights in the United States. The ASDI feed is incremental, its update messages only contain updated parts of flight status, unchanged parts of the flight status are left out (except for the flight identifier, which is needed to distinguish the messages).

CAP server parses that information and then forwards that information to all interested clients. CAP differs from traditional client-server relationship by providing streaming data that is highly customized to fit each client’s need. Clients express their interests in certain flights by registering predicates and templates with the CAP server. Although Figure 2-1 shows a single client, the CAP server is designed to handle multiple clients on a single server, within the limits of the processing power of the host hardware.

CAP client functionality may vary depending on its intended application. While we have some client applications in mind, we do not intend to discuss them here. For the rest of this paper, we will focus on the CAP server. To the client, the CAP server provides four services: authentication, predication, templates, and persistence. Each function is described separately in the following sections.
2.1 Authentication

The client identifies a user with a user name and password pair. Upon connecting with the CAP server, the client uses the user name and password pair to login. After the server verifies the user name and password pair, the client and the user using the client are one and the same to the server. We use the words “client” and “user” interchangeably in the rest of this paper.

Each user also has a set of privileges that determine the type of flights the user can track using the client application. These privileges set by a system administrator upon the creation of the user account in a user database. When a user login, the server read the user database and retrieves the associate privileges, along with the user’s password.

The privileges are used to restrict the user to query flights belonging to a specific airline, or flights arriving at a specific airport. A client is free to define any set of flight that are relevant to its needs, but it never receives information regarding those flights forbidden by its privileges. Therefore, the privileges take precedence over requests.
2.2 Predication

Predication refers to the mechanism by which the clients specify the flights relevant to their interests. Different clients may have very different interests. One client may only want specific flights identified by their flight ID, another may want to know about all flights entering or leaving certain airport. Predication is a flexible mechanism to support many modes of flight selection by the client.

Logically, a predicate is a function, mapping from flight status records to booleans. If a predicate evaluates to \textit{true} for a flight record, we say it \textit{matches} that flight record. A client can register for flights by sending the server a predicate that matches those flights. Examples of predicates are \textit{\"(Airline = US Airways)\"}, \textit{\"(Altitude > 500)\"}, \textit{\"(destination = Boston)\"}, etc.

When performed on the server side, predication helps to reduce the bandwidth needed between the CAP server and the client. The server receives update messages from ASDI regarding nearly every flight in the U.S.. This data stream requires a lot of bandwidth. A CAP client usually will connect using some low-bandwidth connection, but it is only interested in a small subset of the flights CAP knows about. Predication allows us to trade-off between server load and bandwidth, by allowing computation to occur on the server side. Predicates also simplify the logic on the client side. As we will show later in section 3.1, the same mechanism can be used to implement and enforce user privileges.

The predicate mechanism must be flexible enough for clients to specify a wide variety of relevance criteria. We imagine a client may want to specify an arbitrary region of airspace, such that all flights inside that region are relevant to the client.

2.3 Templates

In addition to predication, templates are a second mechanism to further reduce bandwidth needed between the CAP server and client. As mentioned before, the flight status update messages from ASDI are incremental: only the updated parts of a flight
status is sent. Templates takes this idea further, it allows the client to specify the parts of the flight record that the client needs. The extraneous parts of the flight record are filtered out by the server.

In order to support filtering within a flight record, we decompose each flight record into a set of fields. Each field has a fixed name, and an associated value. For example, “flight ID” is a field name, and “USA391” is a possible value. Note that field values are optional. Due to the incremental nature of the ASDI feed, a flight record may be incomplete at times. Incomplete flight records are represented using missing field values.

Given this structure of flight records, a template is simply a set of flight status record field names. The fields included in the template are the fields to update the client with. When ASDI sends an update message to the CAP server, several fields within a flight record are changed by the update message. A template is applied to the post-update flight record to strip out the fields not needed by the client. The resulting partial flight record is the update message to be sent to the client.

There are two types of templates. They differ in how they are applied to a flight record. Consider all the fields in a recently updated flight record. They can be categorized into four groups, depending on if the field has changed or not, and if the template includes the field. If a field is not in the template, then it is not included in the message to the client regardless of it being changed or not. If a field has been changed and is in the template, then it is always included in the update message to the client.

If a field has not been changed, but is in the template, then its inclusion in the update message depends on the type of the template. If the template is a update template, then this type of field is excluded. That is, a client that registers an update template will only receive those fields that has changed recently and are included in the template. If the template is a complete template, then all fields in the template are to be sent to the client.

In general, update templates generate equal or smaller sized update messages than complete templates. However, update templates force the client application to store
some state in order to remember the values in fields that have not changed. Complete
templates are thus simpler to process by the CAP server. At the same time, they
simplify the logic on those clients that do not wish to store state by incurring some
cost on bandwidth. Complete templates are useful for client platforms that have very
small storage space compared to their communication bandwidth, such as mobile
phones.

2.4 Persistence

As a consequence of ASDI sending only incremental update messages to its clients, an
ASDI client must have some form of persistence to store its flight status. Suppose a
client disconnects from the ASDI feed temporarily. When the client reconnects, it has
missed some of the updates to flight information. Since ASDI supplies a streaming
data feed, not a request/reply protocol, the client can only wait for future updates in
order to ensure accuracy of its data.

Unfortunately there is no guarantee that certain fields will be updated again.
With a field such as coordinate fix, it is very likely that updates are sent frequently.
But for other fields such as flight route, no update will be sent unless it changes, and
flight routes don’t change frequently.

CAP sends incremental messages to its clients as well, so we have taken care
to avoid such problems on the client side. CAP is designed so that a client can
disconnect, and reconnect at some point later, and receive any flight status info it
needs, provided that those flights are still active. Thus a CAP client does not need
persistent states. All the persistence is provided by the CAP server, which has a
persistent store of all the flight plans that are active in the U.S. When a CAP client
connects and registers its predicate and template, it will first receive a series of update
messages to download the latest flight status of all the fields specified by its template.

This is just a simple mechanism allowing the client to pull data from the server.
With this function, CAP clients can be built without persistent storage because all
the persistence needed is provided by the server.
Chapter 3

Design and Analysis

This chapter describes the process by which we designed and analyzed the architecture of CAP using the object modelling language Alloy. Alloy is a new object modelling language developed by the Software Design Group at the MIT Lab for Computer Science. A detailed description of Alloy can be found in [Jac99].

This chapter does not assume any knowledge of Alloy by the reader. Rather, it introduces parts of the language as they are used in the models. Alloy has a graphical syntax that is a subset of the UML graphical notation to complement its textual language. We use both notations here for our models to facilitate reading.

3.1 Registration

We start by modelling the registration of flights by clients. RegistrationModel is shown in Figure 3-1.

An Alloy model focuses on sets and relations. The first paragraph, marked by the keyword domain, lists all the basic sets in this model. The second paragraph, marked by the keyword state, lists all the relations between the sets, and subsets of the basic sets. Relations can have multiplicity markings on both ends of its mapping. A multiplicity marking indicates how many elements may participate on the left and right-side of a given relationship. The markings in Alloy and their interpretations are: *, zero or more; +, one or more; ?, zero or one; !, exactly one.
Figure 3-1: Model describing Registration and Policy
Graphically, the sets and relations are represented by boxes and arrows, respectively. Multiplicity markings are placed on the ends of relation arrows. The marking * can be omitted both in textual and graphical notations.

In RegistrationModel, each client can register a set of flights relevant to its interest. They do so by sending a request containing a predicate. That predicate is then used to match a set of flights. Each client also has a reject predicate that is not a part of its request. The keyword static says a client cannot change the reject predicate associated with it. The corresponding graphical syntax places a hatch next to the arrow of the reject relation. Each client may have zero or one reject predicate, as denoted by the ? marking. The idea here is that reject predicates are imposed by the system on the clients.

The remaining paragraphs in RegistrationModel impose additional constraints on the relations of the model. The syntax for constraints in Alloy is based on set notation. It should look familiar to anyone with background in discrete mathematics. The registers relation, defined in the def registers paragraph, states the set of flights a client can register is the set matched by its request predicate, minus the set matched by its reject predicate, if the client has an associated reject predicate.

The last paragraph states an assertion about this model that is checkable by the analysis tool Alcoa. It simply states that no client can register for flights matched by its reject predicate. It serves as a sanity check that we defined the registration policy correctly.

The RegistrationModel is simplified to provide an introduction to how modelling is done in Alloy. RegistrationModel treats flight records, predicates as discrete entities, which is convenient because we can hide a lot of details that are not relevant in this model. In the following sections, we will develop more detailed models for these components, as well as other parts of CAP.
model FlightRecordModel
{
    domain {FlightRecord, fixed FieldName, FieldValue}

    state {
        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    // f.id is just a shorthand for f.value[FlightID]
def id {
        all f | all fid: FlightID | f.id = f.value[fid]
    }

    inv NameValuePaired {
        all f | all fn | fn in f.field -> some f.value[fn]
        all f | all fn | some f.value[fn] -> fn in f.field
    }

    // ASDI always provides a FlightID
    inv FlightIDRequired {
        all f | FlightID in f.field
    }

    // FlightRecords always have some Fields and Values
    // valid: since FlightID is required
    assert NoEmptyRecord {
        all f | some f.field && all fn: f.field | one f.value[fn]
    }
}

Figure 3-2: Model for Flight Record Data Structure
3.2 Flight Record

The FlightRecordModel models the representation of flight records in CAP, it is shown in Figure 3-2. A flight record consists of a set of fields, each field has a name and a value. Together, we call this pair a field of a flight record. For example, “speed” is a field name, and “500 knots” is a field value. For clarity, we do not include field as a domain in our model. Instead, we directly relate a flight record with field names and field values.

Note that FieldName is a fixed set. A set is fixed if it neither gains nor loses members at all times. Graphically, fixed sets are marked with two vertical lines down the left and right-hand side of its box. FieldName is fixed because a flight record stores a pre-determined set of properties of flights. FlightID is a distinguished field name that is useful later. It is declared as a singleton subset of FieldName by using the ! marking next to its name. Graphically, Alloy represents subsets as boxes with a triangular-head arrow pointing to its parent set.

The value relation is an indexed relation. In Alloy, an indexed relation declared as $r[X]: S \rightarrow T$ is a collection of relations, such that for a given element $x$ of the domain $X$, $r[x]$ is a relation from $S$ to $T$.

Therefore, the value declaration reads as for each field name $a$, value[$a$] maps a flight record to zero or one field value. It is a convenient notation to relate field names, field values and flight records together. Field values are optional because ASDI, the input to CAP, provides incremental updates of flight status, it is not possible to guarantee a value is known for every field of a flight record.

Finally, the id relation provides a shorthand for referring to the flight id of a flight, without using the verbose value relation.

So far we have not sufficiently restricted the flight record representation because there is no correspondence between the field names and values of a flight record. The NameValuePaired invariant imposes a one-to-one correspondence between the field names and the values mapped to by one flight record. In addition, the FlightIDRequired invariant requires every flight record to have a flight ID.
3.3 Predicate Matching

Figure 3-4 shows the model for predicate matching and flight records. A predicate consists of a set of clauses, each clause has one or more terms. A term consists of a field name, and a field value used for comparison.

The key relation being defined in this model is matches. As shown in its definition, a predicate matches a flight record if all the terms in one of its clauses matches that flight record. In other words, clauses are connected by “or”s, and terms are connected by “and”s.

Building this model uncovers two design issue. First, as demonstrated by the NoClause assertion, the matches definition allows a client to send a predicate that contains no clauses. Such a predicate never matches any flight record by the current definition of matches. This is not a desirable possibility.

However, this is not easily fixed by constraining predicates to have more than one clause. That approach does not allow the possibility of a “wild-card” predicate: one that matches any flight record. A better solution is to interpret a predicate with no clauses as the “wild-card” predicate, by using the following definition:
model PredicateModel
{
    domain {FlightRecord, fixed FieldName, FieldValue,
            Predicate, Clause, Term}

    state {
        matches: Predicate -> FlightRecord
        clause: Predicate -> Clause
        term: Clause -> Term+
        tfield: Term -> FieldName!
        tvalue: Term -> FieldValue!

        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    def matches {
        all p | p.matches = {f | some c: p.clause |
                             all t: c.term |
                             all fn: t.tfield | f.value[fn] = t.tvalue}
    }

    // a predicate with no clause
    // never matches any flight record
    // valid: the matching rule is too simple
    assert NoClause {
        all p | no p.clause -> no p.matches
    }

    inv FlightRecordInv {
        all f | all fn | fn in f.field -> one f.value[fn]
        all f | all fn | one f.value[fn] -> fn in f.field
        all f | FlightID in f.field
    }

    def id {
        all f | all fid: FlightID | f.id = f.value[fid]
    }
}

Figure 3-4: Model for Predicate Matching with Naive Matching Method
def matches {
    all p | no p.clauses -> p.matches = FlightRecord
    all p | some p.clauses -> p.matches =
        {f | some c: p.clauses | all t: c.term |
          all fn: t.tfield | f.value[fn] = t.value}
}

The other design issue uncovered is the matching of a term with a flight record that is missing the field name specified in the term. Consider the matching of the predicate "(Destination = BOS) and (Altitude > 300)" with a flight record whose altitude value is missing. If this is a match, then the client may receive a superset of the flights it intended to. If this is not a match, then the client receives a subset of its flights.

It should be clear that in the above definition of matches, flight records with missing fields do not match a term, so all matching flights satisfy the predicate completely. This is a design choice we made. The other choice is also acceptable. However, the actual choice made is not the important point of this discussion. Rather, it should be noted that object modelling uncovered these detailed design decisions with very little work in building the models themselves.

### 3.4 Template

Figure 3-5 shows the model for templates. Templates are very simple components. They simply consists of a set of field names selected by the client. As such, we omit the graphical representation of the template model.

Templates are used to generate a flight record by removing some unwanted fields from another flight record. The generation of a flight record is not easily modeled using a relation, since the set FlightRecord is being changed. Alloy can model changes in state of a model using operations. An operation is modeled by specifying constraints relating the state before the operation, and the state after the operation. Post-operation states are specified using a prime character after state variable, similar to standard math notation. In addition, operations can take elements of the domains as arguments of the operation.
Figure 3-5: Model for Templates that Violates Flight Record Invariant
The TemplateModel models a single operation: applyTemplate. Argument $f$ is a single flight record in the pre-operation set FlightRecord, while $output$ is an element of the post-operation set. $t$ is the template being applied.

The first constraint states that $output$ is a new flight record generated by this operation. This is necessary because the argument list only specifies $output$ as a member of the FlightRecord’ set, it does not constrain the membership of $output$ in the FlightRecord set. The second constraint states the fields included in the $output$ flight record must part of flight record $f$ and selected by the template $t$. The next two constraints specify the field values for $output$ such that it maintains the FlightRecordInv invariant. The fifth constraint adds $output$ to the post-operation set FlightRecord’, and remaining constraints make sure no other flight records are modified by this operation.

Judging from the detailed specification of the operation, it would seem that applyTemplate preserves the representation invariant of flight records. Nevertheless, using Alcoa to analyze applyTemplate against FlightRecordInv finds a example where the invariant is violated. This model fails to specify that templates should always include the flight ID. If a template that does not select flight ID is applied, the $output$ flight record certainly would not have flight ID. To uphold the FlightRecordInv invariant, we add the following definition to the model.

```
def selects {
   all t | FlightID in t.selects
}
```

This definition reminds us that in our implementation, we should disallow a client to include in its request a template that does not select the flight ID field. In addition, even if the flight ID is always selected, a template can still generate a flight record with only a flight ID, and no other fields. Since such a flight record contains no useful information, the CAP server should discard these empty messages.
model StateDatabaseModel
{
    domain {FlightRecord, fixed FieldName, FieldValue}

    state {
        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
        StateDB: FlightRecord // we keep a DB of flightRecords
    }

    inv FlightIDUnique { // no duplicate flight ID within the database
        all f1, f2 : StateDB | f1 != f2 <-> f1.id != f2.id
    }

    op addFlight (f : FlightRecord!) {
        f !in StateDB
        all fr: StateDB | fr.id != f.id
        FlightRecord’ = FlightRecord
        StateDB’ = StateDB + f
        all fr | fr.field’ = fr.field
        all fr | all fn | fr.value’[fn] = fr.value[fn]
    }

    op updateFlight (newF: FlightRecord!, oldF : StateDB!) {
        newF.id = oldF.id
        StateDB’ = StateDB
        FlightRecord’ = FlightRecord
        oldF.field’ = oldF.field + newF.field
        all fn : newF.field | oldF.value’[fn] = newF.value[fn]
        all fn : FieldName - newF.field | oldF.value’[fn] = oldF.value[fn]
        all fr: FlightRecord - oldF | fr.field’ = fr.field
        all fr: FlightRecord - oldF | all fn | fr.value’[fn] = fr.value[fn]
    }

    op removeFlight (newF: FlightRecord!, oldF : StateDB!) {
        newF.id = oldF.id
        StateDB’ = StateDB - oldF
        FlightRecord’ = FlightRecord + oldF
        all fr | fr.field’ = fr.field
        all fr | all fn | fr.value’[fn] = fr.value[fn]
    }
}

Figure 3-6: Partial Model for Flight Database Operations
3.5 State Database

Now that we have introduced Alloy's ability to model state transitions, we can model a critical component of the CAP server, the flight state database. In the previous models, the domain $\text{FlightRecord}$ represents the set of all possible flight records. We can do so because Alloy models are abstract. Our database, however, can only store a finite set of flight records. We model the database as a subset of the domain $\text{FlightRecord}$. As with the template's model, we omit the graphical representation for this model.

The flight state database is a set of the flight records that represents the current known status of all the flights. The flight records in the database contain unique flight IDs, otherwise it would be unclear as to which record is the correct information. As ASDI sends messages to CAP, these messages bring updated flight status. We model these messages as flight records that are not a part of the database.

There are three different database operations that are of interest here: adding a flight record to the database, updating a flight record already present in the database, and removing a flight record from the database, as shown in Figure 3-6. The $\text{FlightRecordInv}$ invariant and the $id$ definition are removed from Figure 3-6 due to space limitations. They are the same as those in Figure 3-5.

These operation definitions are mostly straightforward. Note that $\text{removeFlight}$ takes two arguments, $newF$ and $oldF$. It does so because the database does not know when to remove a flight record unless another flight record updates it with field values that indicate the flight has completed, canceled, or otherwise terminated. Since a removal can only occur after an update, it takes on similar arguments.

As they are defined, these operations ensure the database maintain a set of flight records with unique IDs ($\text{FlightIDUnique}$), and flight records have properly paired field names and values ($\text{FlightRecordInv}$). In addition, this set of operations never change the flight ID of any flight record in the database ($\text{updateFlight}$ only operates when the ID of the new record is the same as the ID of the old one). It was discovered later that ASDI actually sends a flight amendment message that can change the ID for
some existing flight. The effects of that discovery is discussed in the next section.

3.6 Flight ID Amendment

The flight amendment (AF) message consists of a list of changes to a flight record. But it differs from other ASDI messages because it can include a change to the flight ID. A flight ID amendment message basically consists of the old flight ID and the new flight ID.

The design described so far in this chapter does not consider such changes to a flight record. In fact, one can write an assertion in Alloy to check that a flight records in the state database does not change with any operation. We leave that as an exercise for the reader. The presence of this message requires us to reconsider our entire design. Fortunately, the object models we have built allow us to identify all the changes we need to make in our design, and analyze the effects of those changes on the invariants of our design quickly.

We examine each of the models presented so far in this chapter. \textit{RegistrationModel}, \textit{FlightRecordModel}, \textit{PredicateModel} do not depend on modifications of flight records. They are unaffected by the presence of the AF message. It should be clear that since the AF message basically specifies a state transition, it affects only those models that specify operations: \textit{TemplateModel} and \textit{StateDatabaseModel}.

3.6.1 Templates and Flight ID Amendment

The \textit{TemplateModel} is affected because in the previous design, a template is applied to generate a flight record that is sent to those clients registered this flight record. But a flight record is insufficient to capture the effects of an AF message, since it can only contain one flight ID. A data type is needed, which we call \textit{FlightUpdate} (see Figure 3-7).

Compared to a flight record, a flight update simply has one additional field value that is the old ID of the flight record being updated (\textit{oldid}). Every flight update contains a flight record that has all the new field values. In the case where the ID is
model TemplateModel2
{
  domain {FlightRecord, FlightUpdate, fixed FieldName,
         FieldValue, Template}

  state {
    selects: Template -> FieldName+
    field: FlightRecord -> FieldName
    FlightID: fixed FieldName!
    value[FieldName]: FlightRecord -> FieldValue?
    id: FlightRecord -> FieldValue?
    StateDB: FlightRecord

    oldid: FlightUpdate -> FieldValue!
    newStatus: FlightUpdate -> FlightRecord!
    updates: FlightUpdate -> StateDB?
  }

  def updates {
    all fup | fup.updates = {fr | fr.id = fup.oldid}
  }

  def newStatus {
    all fup | fup.newStatus !in StateDB
  }

  op applyTemplate (fup: FlightUpdate!, f: FlightRecord!,
                    output: FlightUpdate'!, t: Template!)
  {
    f in fup.updates
    output !in FlightUpdate
    output.oldid' = fup.oldid
    output.newStatus.field' = t.selects & f.field
    all fn: output.newStatus.field'
    output.newStatus.value'[fn] = f.value[fn]
    all fn: FieldName - output.newStatus.field' |
    no output.newStatus.value'[fn]
    FlightUpdate' = FlightUpdate + output
    FlightRecord' = FlightRecord + output.newStatus
    all fr: FlightRecord | fr.field' = fr.field
    all fr: FlightRecord | all fn | fr.value'[fn] = fr.value[fn]
  }
}

Figure 3-7: Partial Model for Flight Updates and Templates
being changed, the new ID would be inside the newStatus flight record. Each flight update message updates one flight record in the state database, or zero if this flight is being added.

With these changes, a client still registers for flight records. But it must receive flight updates from the CAP server instead of flight records. The applyTemplate operation must generate a flight update. To do so, it must use the original flight update send to CAP by ASDI. The definition of applyTemplate is modified accordingly. Notice that the generated output flight update has the same oldid as the original flight update.

Once we change the definition of applyTemplate, Alcoa allows us to easily verify the new definition does not violate the invariants FlightIDUnique and FlightRecordInv. We now modify the design of the state database model to incorporate the idea of flight updates. This modification uncovers a subtlety that demonstrates the power of Alloy very nicely.

### 3.6.2 State Database and Flight ID Amendment

The changes made to StateDatabaseModel are shown in Figure 3-9. Again, to save space, the definitions of newStatus, updates, and invariants of the flight record are omitted.

The three new operation definitions of the database are each checked against invariants FlightRecordInv and FlightIDUnique. Alcoa finds a violation of FlightIDUnique
model StateDatabaseModel2
{
    domain {FlightRecord, fixed FieldName, FieldValue, FlightUpdate}

    state {
        StateDB: FlightRecord
        oldid: FlightUpdate -> FieldValue!
        updates: FlightUpdate -> StateDB?
        newStatus: FlightUpdate -> FlightRecord!
        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    op addToStateDB (fu : FlightUpdate!) {
        all fr: StateDB | fr !in fu.updates
        FlightRecord’ = FlightRecord + fu.updates
        StateDB’ = StateDB + fu.updates
        all fr | fr.field’ = fr.field
        all fr | all fn | fr.value'[fn] = fr.value[fn]
    }

    op updateFlight (fu: FlightUpdate!, oldF : StateDB!) {
        oldF in fu.updates
        StateDB’ = StateDB
        FlightRecord’ = FlightRecord
        oldF.field’ = oldF.field + fu.newStatus.field
        all fn : fu.newStatus.field |
            oldF.value'[fn] = fu.newStatus.value[fn]
        all fn : FieldName - fu.newStatus.field |
            oldF.value'[fn] = oldF.value[fn]
        all fr: FlightRecord - oldF | fr.field’ = fr.field
        all fr: FlightRecord - oldF | all fn | fr.value'[fn] = fr.value[fn]
    }

    op removeFlight (fu: FlightUpdate!, f : StateDB!) {
        f in fu.updates
        StateDB’ = StateDB - f
        FlightRecord’ = FlightRecord + f
        all fr | fr.field’ = fr.field
        all fr | all fn | fr.value'[fn] = fr.value[fn]
    }
}

Figure 3-9: Partial Model for Flight Updates and State Database
by the *updateFlight* message. The instance it generates involved a change of flight ID to a flight record in the database, where the new ID is already used by another flight record also present in the database.

This case is permitted by the current *updateFlight* definition, but at first we did not find it interesting. After all, it is quite unthinkable for ASDI to feed a flight ID amendment that results in duplicate IDs. We also did not find any ASDI documentation that mentions such a case. Isn’t it reasonable to assume that flights have unique IDs in the ASDI stream?

Just to be sure we added an assertion in our CAP implementation to notify us when such a case occurs. Much to our surprise, the case actually occurred, and its circumstances are quite subtle. The tracking information provided by ASDI comes from a number of facilities called Air Route Traffic Control Centers (ARTCCs), commonly known as “Centers.” There are about 20 of these in the continental U.S., and several located offshore. Each Center tracks the flights within its predefined airspace, and they transfer control when flights cross Center boundaries. It is possible for two different Centers to track the same flight at times, when a flight flies along a boundary line, for example. This does not cause any problems since these Center use the same flight ID.

The ASDI feed merges the tracking information from Centers in the U.S., and those in Canada. The Canadian system, being separate from the U.S. Centers, can choose different flight IDs for flights. When a Canadian Center and a U.S. Center track the same flight, but using different flight IDs, that flight may appear as two flights flying next to each other in the ASDI stream. When this is detected, ASDI sends a *AF* message to change one of the flight ID to match the other one, thus creating the counterexample found by Alcoa.

The first lesson learned here is of course to check any assumption used in system design, no matter how reasonable it may seem. The second lesson is that a good tool, such as Alcoa, can help to identify all those assumptions in the design and specification stage, before the implementation and testing, to reduce the ill effects of an design flaw.
The case aforementioned is handled by removing the duplicate entry in the state database, since the two record entries actually correspond to the same flight. We modify the `updateMessage` operation by changing the second constraint to

\[ \text{StateDB}' = \text{StateDB} - \{f | f.id = \text{fup.newStatus.id}\} \]

### 3.7 Summary

In this chapter, we showed how to use Alloy and its companion analysis tool Alcoa to assist the design several key components in CAP. We have defined the structure of a predicate, a flight record, and how to match a set of flight records using a predicate. We also defined the operations involving templates and the flight state database, and analyzed the effect of these operations on the system invariants of CAP using Alcoa.

Here we summarize the ways Alloy, or object modelling in general, can help with system design. An object model is a form of specification. Building an object model has many of the same benefits of writing down the specification. First, it serves to solidify high-level concepts by putting them in more concrete representations. In the `PredicateModel`, when we broke down the idea of predicates into clauses and terms,
we discovered the corner case of an empty predicate. Of course these details can be discovered during implementation, but often with some cost in labor or time.

Second, building an object model helps to identify design decisions early on. Building the `PredicateModel` forces us to write a specification of the matching rule, which revealed a choice in handling flight records with missing fields. Avoiding these surprises in the implementation stage helps the development process to proceed smoothly.

Finally, automatic analysis at the semantic level helps to identify flaws in the design. Experience shows that the bugs that stem from design flaws are much more costly to correct than bugs in implementation, because a change in design require much more change in code. Alcoa helped to find a flaw in the first design of CAP because we had an invalid assumption regarding flight ID amendment messages. Due to the infrequency of the problem scenario in the input, it would be very difficult to find this bug, if we did not have Alcoa’s counter example ahead of time.

Overall, object modelling combined with semantic analysis enables an architect design systems using the top-down approach, but with the rigor and precision that is often associated with the bottom-up approach. This combination helps to discover corner cases, uncover design decisions, and detect flaws much earlier in development process than traditional methodologies.
Chapter 4

Implementation

This chapter examines the architecture and implementation of the CAP server prototype.

CAP is divided into the server component and the client component. The client architecture depends on its function and application, therefore is not within the scope of this paper. There are five main modules in the server: the parser, the registrar, the filter engine, the flight status database, and the distributor (see Figure 4-1).

The parser parses the stream of messages from the ASDI into individual message objects. The registrar is responsible for client authentication and registering requests from clients with the filter engine. The filter engine performs predication and applies templates to generate appropriate update messages for registered clients. The flight status database aids the filter engine by keeping the latest status information for all active flights. The distributor takes update messages from the filter engine, and sends these update messages to each client.

Each component runs in its own thread to perform their separate tasks. The communication between components are handled using FIFO queues that are synchronized to be thread-safe. That is, message objects parsed out by the parser are placed on the queue for the filtering engine to process them. This architecture is necessary since the filtering process have variable delays depending on the number of clients, their requested predicates and templates. During this processing time, CAP appears unresponsive to ASDI, and would be disconnected by ASDI. Threads
provide the parallelism needed to process predicates and parse incoming messages simultaneously. Queues provide buffering of delays between the components.

4.1 Parser

The parser parses the ASDI message stream into concrete message objects. The ASDI feed is a stream of real-time air traffic data collected from National Airspace Systems (NAS) and Enhanced Traffic Management System (ETMS). The data in the stream are separated into individual messages. Messages are categorized by types. There are ten types of messages: seven for NAS messages, three for ETMS messages. Figure 4-2 shows the complete ASDI message hierarchy. Table 4.1 shows the purpose of each message, and their information content.

Several details of the ASDI feed relevant to our design. First, ASDI is an incremental feed. This means that it does not send a complete flight status in each message. Rather, ASDI server sends the parts of the flight status that has changed since the last message, and assumes the ASDI client has stored the last flight status update. For example, the amendment flight message (AF) only contains a flight ID, a flight status field reference, and the updated field value [Vol99]. The incremental nature of ASDI is one of the reasons the CAP server must have a flight status database.
Figure 4-2: ASDI Message Hierarchy

<table>
<thead>
<tr>
<th>Type</th>
<th>Purpose</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF</td>
<td>Amendment</td>
<td>Origin, Destination, Amendment Data</td>
</tr>
<tr>
<td>AZ</td>
<td>Arrival</td>
<td>Origin, Destination, Arrival Time</td>
</tr>
<tr>
<td>DZ</td>
<td>Departure</td>
<td>Aircraft Data, Origin, Destination, Departure Time, Estimated Arrival Time</td>
</tr>
<tr>
<td>FZ</td>
<td>Flight Plan</td>
<td>Aircraft Data, Speed, Requested Altitude, Route</td>
</tr>
<tr>
<td>RZ</td>
<td>Cancellation</td>
<td>Origin, Destination</td>
</tr>
<tr>
<td>TZ</td>
<td>Tracking</td>
<td>Speed, Altitude, Position</td>
</tr>
<tr>
<td>UZ</td>
<td>Update</td>
<td>Aircraft Data, Speed, Boundary Crossing Point, Boundary Crossing Time, Altitude, Route</td>
</tr>
<tr>
<td>HB</td>
<td>Heart Beat</td>
<td></td>
</tr>
<tr>
<td>RT</td>
<td>Flight Plan</td>
<td>Flight Status, Route, Physical Class, User Class, Origin, Destination</td>
</tr>
<tr>
<td>TO</td>
<td>Oceanic Report</td>
<td>Reported Position, Altitude, Origin, Destination</td>
</tr>
</tbody>
</table>

Table 4.1: ASDI Message Types
Second, ASDI sends an update whenever newer data is available for a flight. It allows the possibility that the status for one flight is recorded more recent than another flight. To give an example, suppose there were two flights, flight A and B. ASDI can send two update messages for A and B. One corresponding to A’s status at 6:00pm, the other to B’s at 5:40pm. Since the ASDI client’s knowledge of flight B’s status can be stale compared to that of flight A, one should be very careful when interpreting the relationship between two flights using the ASDI data. In particular, one should not interpret the collection of flight status information from ASDI as a snapshot of the air space at any particular time.

The same issue presents itself in CAP. To the CAP clients, the server sends individual flight updates similar to that of ASDI. Consider the design where the server maintain consistency of flight states among all flights, in order to present a snapshot view of the air space to the clients. Since ASDI does not provide such information, the CAP server must extrapolate current flight status from previous status. For example, a flight’s current coordinate fix can be easily interpolated using its last reported coordinate fix, speed, and time passed since the last update.

This design is not desirable for several reasons. First, it is not CAP’s function to make air traffic forecasts. CAP’s main role is data distribution, it should not mutate the air traffic data in the process. Second, estimation and forecasts are just as easily performed on the client side. In fact, we are considering this feature in the client application we are building. It is only advantageous for the server to perform this task if all the clients want estimated data, then results of the computation can be shared. It should be clear that some clients require the original data. Third, while it is easy to interpolate aircraft position, it is not clear how to estimate changes in other fields. If we only interpolate aircraft position and ignore the other fields, then we may introduce inconsistency of data among the fields of a flight record.

For the above reasons, we chose to only ensure that individual flight record in CAP contains the last known status of that flight, but the entire flight status database does not reflect the state of the air space at any moment in time.
Figure 4-3: Flight Update and Flight Record MDD

<table>
<thead>
<tr>
<th>Field No.</th>
<th>Field Name</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flight ID</td>
<td>ID</td>
</tr>
<tr>
<td>1</td>
<td>Computer ID</td>
<td>CID</td>
</tr>
<tr>
<td>2</td>
<td>Aircraft Data</td>
<td>Aircraft</td>
</tr>
<tr>
<td>3</td>
<td>Coordinate Fix</td>
<td>Coordinate</td>
</tr>
<tr>
<td>4</td>
<td>Coordinate Time</td>
<td>Time</td>
</tr>
<tr>
<td>5</td>
<td>Speed</td>
<td>Speed</td>
</tr>
<tr>
<td>6</td>
<td>Requested Altitude</td>
<td>Altitude</td>
</tr>
<tr>
<td>7</td>
<td>Assigned Altitude</td>
<td>Altitude</td>
</tr>
<tr>
<td>8</td>
<td>Route</td>
<td>Route</td>
</tr>
<tr>
<td>9</td>
<td>Remark</td>
<td>Remark</td>
</tr>
<tr>
<td>10</td>
<td>Tracking Position</td>
<td>Position</td>
</tr>
<tr>
<td>11</td>
<td>Origin</td>
<td>Fix</td>
</tr>
<tr>
<td>12</td>
<td>Destination</td>
<td>Fix</td>
</tr>
<tr>
<td>13</td>
<td>Arrival Time</td>
<td>Time</td>
</tr>
<tr>
<td>14</td>
<td>Departure Time</td>
<td>Time</td>
</tr>
<tr>
<td>15</td>
<td>Status</td>
<td>Status</td>
</tr>
<tr>
<td>16</td>
<td>Physical Class</td>
<td>PhysicalClass</td>
</tr>
<tr>
<td>17</td>
<td>User Class</td>
<td>UserClass</td>
</tr>
<tr>
<td>18</td>
<td>Coordinate</td>
<td>Coordinate</td>
</tr>
<tr>
<td>19</td>
<td>Reported Altitude</td>
<td>Altitude</td>
</tr>
<tr>
<td>20</td>
<td>En-route Time</td>
<td>TimeInterval</td>
</tr>
<tr>
<td>21</td>
<td>Boundary Crossing Time</td>
<td>Time</td>
</tr>
</tbody>
</table>

Table 4.2: Flight Record Fields and Field Types
4.1.1 Flight Update

The output of the parser is a sequence of flight update objects. Flight update is simple class, consists of a flight ID and a flight record (see Figure 4-3). The flight ID is used to identify the flight record in the flight status database. The flight record within flight update contains updated field values (field values are optional, so unchanged fields are left empty). See 3.6 for detailed analysis involving the flight update.

Note that there is a field class, instead of a field name class and a field value class. This is because the field names are represented using integers, and field values are represented using classes. We choose this representation because there are no operations on field names other than equality, but there are a lot more operations on field values. Thus, the field class in 4-3 is the superclass of all field values. ID is a subclass of field used to represent flight ID.

Figure 4-3 does not show all the subclasses of field. Refer to table 4.2 for a complete list of field names, numbers, and types. Field numbers are internal to the CAP server, and are not exposed to the client. Some of the field types inherit from each other. For example, fix and coordination are both subclasses of position. This is necessary because some ASDI messages may report position as either a fix (“SFO”) or a coordinate (“3457N/08111W”) [Vol99].

Given this flight record representation, the parser simply takes an ASDI message, parse the message content into field values, and set the value of the appropriate field number. In some cases the parser will set fields that are not explicitly available in the ASDI message. For example, the flight cancellation message (RZ) only has flight ID, origin, and destination. If the parser outputs a flight record with these fields, it is not clear that the flight is canceled. For that, the parser sets the flight status to “canceled”. The arrival message (AZ) requires similar handling.

4.2 Registrar

The registrar handles authentication and registration from the client. The registrar reads from an user database, which contains user names, passwords, and privileges of
all users. The registrar does not write to the user database. It is assumed that a system administrator will modify this database whenever needed. Recall that privileges are implemented using reject predicates. Each user has a reject predicates, which are set by a system administrator in the user database. Recall that a predicate is used to match a set of flights. A reject predicate is used to forbid an user from ever receiving status of a set of flights. Naturally, an user cannot change his/her associated reject predicate without notifying the system administration.

The client initiates a session with the CAP server by connecting with the registrar. Upon establishing the connection, the client sends its request. A request consists user name, password, a predicate that specifies the relevant flights, and a template that specifies the relevant fields.

The registrar’s first task is to parse the client’s request. When parsing is completed, the registrar read the user database to authenticate the user. If the authentication fails, either due to invalid user name or wrong password, the registrar notifies the client and closes the connection.

If the authentication is correct, then the registrar creates a client info object and registers that with the filter engine. This client info object contains the client’s network socket, the request and reject predicates, as well as the template. Once the client is registered, the registrar returns to accept new connections from other clients.

4.2.1 Predicate Operators

In order for predicates to be useful, they must be expressive. In our Alloy model, each term in a predicate only compared the equality of two field values. This is sufficient for the purpose of modelling the operation of a predicate, but not expressive in practice. In our implementation, a term consists of a field name, a field value, and an boolean operator. The addition of an operator allows us to implement a wide variety of boolean operations in addition to equality.

Table 4.3 lists all different combinations of forming a term. For each value type of a flight record, there are several operators available. The fields that have the same type, such as origin, and destination have the same set of operators. Notice that the
### CAP authentication file

Each line is:

- `<username>=<passwd>
  <reject predicate>`

```plaintext
davidz=cappass1
  { (ID isAirline AA) }
qzwang=cappass2
```

Figure 4-4: User Database Format

Argument type can be of a different type from the field value type. For example, the `isAirline` operator compares an ID value and an Airline value. Thus, we can formulate very expressive terms easily, such as "(Route contains BOS)".

This aspect of predicates is not modeled since it is a feature of the implementation. So long as each operator is implemented correctly, this feature will not affect the overall system behavior.

#### 4.2.2 User Database

A sample user database is shown in Figure 4-4 to demonstrate its format. User names and passwords are plain strings, so they are stored as is. We define a format for predicates, in order to store, and retrieve them from the user database. In string form, clauses are enclosed between braces, and terms are enclosed between parenthesis. In Figure 4-4, there are two users. User davidz is forbidden from receiving flights of American Airlines, and user qzwang has no reject predicate.

#### 4.3 Filter Engine

From here on we will refer to predicates specified in user request as accept predicates, to distinguish from system imposed reject predicates.

The filter engine is responsible for generating the correct update messages for each client registered with the server. It performs the bulk of tasks inside the CAP server. Its inputs are the queued flight updates from the parser, the registered clients
<table>
<thead>
<tr>
<th>Field Type</th>
<th>Operators</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>=</td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>isAirline</td>
<td>Airline</td>
</tr>
<tr>
<td>CID</td>
<td>=</td>
<td>CID</td>
</tr>
<tr>
<td>Aircraft</td>
<td>=</td>
<td>Aircraft</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>Integer</td>
</tr>
<tr>
<td>Coordinate</td>
<td>=</td>
<td>Coordinate</td>
</tr>
<tr>
<td></td>
<td>inside</td>
<td>Rectangle</td>
</tr>
<tr>
<td>Time</td>
<td>=</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>before</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>after</td>
<td>Time</td>
</tr>
<tr>
<td>Speed</td>
<td>=</td>
<td>Speed</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>Speed</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>Speed</td>
</tr>
<tr>
<td>Altitude</td>
<td>=</td>
<td>Altitude</td>
</tr>
<tr>
<td></td>
<td>within</td>
<td>AltitudeRange</td>
</tr>
<tr>
<td></td>
<td>overlaps</td>
<td>AltitudeRange</td>
</tr>
<tr>
<td>Route</td>
<td>=</td>
<td>Route</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td>Fix</td>
</tr>
<tr>
<td></td>
<td>orig.at</td>
<td>Fix</td>
</tr>
<tr>
<td></td>
<td>dest.at</td>
<td>Fix</td>
</tr>
<tr>
<td>Remark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>=</td>
<td>Position</td>
</tr>
<tr>
<td></td>
<td>inside</td>
<td>Rectangle</td>
</tr>
<tr>
<td>Fix</td>
<td>=</td>
<td>Fix</td>
</tr>
<tr>
<td></td>
<td>inside</td>
<td>Rectangle</td>
</tr>
<tr>
<td>Status</td>
<td>=</td>
<td>Status</td>
</tr>
<tr>
<td>PhysicalClass</td>
<td>=</td>
<td>Physical Class</td>
</tr>
<tr>
<td>UserClass</td>
<td>=</td>
<td>User Class</td>
</tr>
<tr>
<td>TimeInterval</td>
<td>=</td>
<td>TimeInterval</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>TimeInterval</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>TimeInterval</td>
</tr>
</tbody>
</table>

Table 4.3: Field Types and Operators
from the registrar, and the stored flight records of active flights in the flight status database. For each flight update in the parser queue, the filtering process takes three steps: update database, match with predicates, apply templates.

4.3.1 Update Database

After getting a flight update from the parser queue, the filter engine searches for a flight record in the flight status database, using the flight ID as the key. If it does not find a record, then a new record is created using the flight ID and the field values present in the flight update object, and added to the database. If the filter engine did locate a record in the database, it updates that flight record with the new field values, and stores that in the database.

The engine also removes flight record from the database in some cases. Whenever a flight becomes inactive, either by arrival at destination or changes in flight plan, its flight record is removed from the database. So all flight records in the database are active flights.

4.3.2 Predicate Matching

At this point the database is up-to-date, the filter engine is ready for the predication step. For each client that is registered and alive (liveliness discussed in section 4.5), the engine tries to match the new flight record with the client's predicates.

The reject predicates takes precedence over the accept predicates. A client's accept predicate will only be matched against a flight record if the reject predicate failed to match. Therefore, the set of flights a client can receive updates about are those matched by the accept predicates minus those matched by the reject predicates. The predicate matching algorithm short-circuits when one of the clause matches the flight, to avoid unnecessary computation.
4.3.3 Apply Template

For those clients that passed the predication step (flight record matches accept predicate but not reject predicate), the filter engine applies their templates to generate client specific flight update messages. Recall that there are two types of templates, update template and complete template. If the client registered a complete template, then unnecessary fields of the flight status are removed to form the update message. If the client registered an update template, then all the unchanged fields are removed first, then the template is applied in the same manner as a complete template. The generated update messages are passed on to the distributor using a queue similar to that of the parser.

4.3.4 New Client Connections

The normal flow of operations for the filter engine is disrupted when a new client registers after being authenticated by the registrar. Recall one of the feature of the CAP server is to provide persistence for the client. When a client connect, it can receive the latest information regarding all the flights it can register.

In order to fulfill that requirement, the filter engine must operate very differently. Normally the filter engine takes one flight record, and try to match that against all connected clients. Now the filter engine must match the new client’s predicate against all flight records in the state database. We call this the download process for new client connections.

Performing the download presents a couple of issues. First, the filter engine must delay processing any flight update from the parser because it is single-threaded. This usually causes a short period of congestion in the parser queue. Second, a new client may connect during this download period. This new client is placed in the message queue between the registrar and the filter engine. When the filter engine finishes the current download process, it can continue to process flight updates, or perform another download for the new client.

It is problematic if the filter engine always performs download for new client,
since it will make no progress on the incoming flight update messages. Rapid client
connection can eventually cause the parser message queue to overflow and cause failure
in the server.

The filter engine must balance between performing downloads for new clients,
and processing flight update messages to ensure consistent progress being made on
both messages queues. There are a number of different strategies for accomplishing
this. It can monitor the size of each queue, and choose to operate on the larger one.
Currently, our prototype implements the simple strategy of never performing two
consecutive downloads.

The filter engine also unregisters any client that is not alive, by terminating their
connections and removing their predicates. However, the filter engine has no way
of detecting the liveness of a client, since it does not actually perform the network
send operation. We get around this by having the distributor mark the clients that
it detect as unresponsive.

4.4 Flight State Database

The flight state database stores the last known state of all active flights. It supports a
several generic database operations: adding a flight record, removing a flight record,
updating a flight record, and extracting keys for all records.

All database operations uses the flight ID as the entry key. Since flight ID may
change while the flight is active, updating a flight record must change the entry key
as well.

The operation to extract all keys in the database is useful for when a new client
registers with the filter engine. The engine must match all records in the database
with the client’s predicate to generate a set of update messages for that client.

One issue of the state database is the rule for removing flight records. Currently,
our prototype removes a flight if it is reported as arrived, completed, or canceled.
But there are flights whose destinations are outside the North American continent,
and ASDI does not contain arrival messages for these flights. The current removal
rule leave these flights in the database.

It is conceivable for the flight database to perform some form of garbage collection, where it periodically search for and remove outdated entries in the database. That feature is currently unimplemented in this prototype.

4.5 Distributor

The distributor component handles the downward flow of information from server to the clients. The primary jobs of the distributor are sending the update messages generated by the filter engine, and unregistering off-line and otherwise disconnected CAP clients.

The task of sending the update messages is straight forward. The distributor off-loads the work of sending messages from the filter engine because filtering and sending should be parallelized due to the amount of delay involved in a network operation.

The task of unregistering off-line or disconnected clients is a bit more complex. The distributor can detect an disconnected client when sending the update message returns an error. In that case, it is easy to notify the filter engine that the client is no longer alive.

However, update messages are generated only if the client’s predicate matches some flight record. Consider the case where the client’s predicate is too restrictive to match any flight. If this client is disconnected, the distributor will not unregister it as long as the filter engine does not generate an update message for this client.

This is undesirable since the disconnected client’s predicate is still being processed by the filter engine, which consumes resources on the server. We want to guarantee that disconnected clients will be unregistered in fixed amount of time. In order to do this, the distributor sends out heart beat messages periodically to probe the clients. A heart beat message is just a dummy message that can be safely ignored by the client. If an error is detected in sending the heart beat message, then the distributor notifies the filter engine that the client is no longer alive.
Chapter 5

Conclusion

5.1 Object Modelling

In this thesis, we applied object modelling techniques to aid the design and analysis of CAP, and learned some lessons about designing with an object modelling language such as Alloy.

First, constructing an object modelling can be very helpful for brainstorming or working out initial concepts. Since Alloy is based on sets and relations, it forces the user to actively think about how the objects in the problem domain relate to each other. Alloy has many expressive language features to capture these relationships succinctly. By focusing the design around the problem state instead of the program state, a resulting system is usually simpler and more flexible.

Second, it is not necessary to build an entire model of the system to benefit from the analysis. Indeed, all the models of CAP were developed to capture the behavior of a part of the system. Still we gained many discoveries and insights. Alloy can be used to model as little or as much of the system as desired. For example, we ignored the behavior of operators in our models, since their behavior is very clear to us and did not contribute to the overall analysis.

Finally, the benefits of having an automatic analysis tool such as Alcoa should not be overlooked. Alcoa generated problem instances and counter-examples add a magnitude of precision to the design process than plain object modelling. Its ability
to locate flaws in the model quickly and efficiently more than justifies the initial effort in constructing the object model.

5.2 Design Review

Our prototype implementation satisfies all the functional requirements described in Chapter 2. It has provided the architecture for information distribution with authentication. Predication is introduced as a flexible mechanism to define relevance of flights to clients. We showed that predication can be used to impose policy on the flights accessible to clients. In addition, we introduced the use of templates to specify the relevant fields within each flight record.

In building this prototype, a number of issues were exposed. First, the predicate matching algorithm must consider the case were a field named in the predicate is missing within the flight record. Second, the flight amendment message must be handled differently, since may change the flight ID, which is a field critical to the entire CAP system. Specifically, all the state database operations must be implemented carefully to ensure uniqueness of flight IDs.

There are a number of issues remaining in this design. Some of these have been discussed in the previous chapters. They will be repeated here for completeness.

One issue occurs when a new client connects with the server. Since the CAP server provides persistence, it needs to send the client all the flight status of every flight that matches its predicates. In this case, the filter engine needs to iterate through all the records in the flight status database, not just the one updated by ASDI.

The problem with this design is that when a client connects to a server, if the client registers for a lot of flights (predicates that are too relaxed, or too many predicates), then the client can cause a spike in the server load, due to the computation needed to download the current flight status. Effectively, each new client connection has costly overhead at the beginning of the connection. In such a system, many connections of short lifespans is problematic for the server. The severity of this problem depends on the size of flight records and the average number of active flights in U.S. air space.
Another issue involves ending a connection from the client. The current design only terminates disconnected clients, so the client always initiates unregistration. It may be desirable for the server to detect when a set of predicates can no longer match any flight, and terminate the client's connection when all flights currently matching the predicates become inactive.

We have mentioned in section 4.1 that flight ID may change for active flights. Suppose a client registers for flights based on their flight ID, and one of the flights changes its flight ID. It is unclear if the server should forward update messages of that flight after the flight ID change. Currently the server would forward the flight ID update to the client, after that no more messages are sent to the client regarding that flight since it no longer matches the client's predicate. It may be desirable to change the predicate for the client automatically so client continues to receive updates.

5.3 Extensions

Several extensions to the current design are being considered. While predication is a very flexible mechanism, evaluating a predicate is computationally costly. One observation is that the predicates registered by many clients may share common clauses, and we can improve the server performance by avoiding to evaluate common clauses multiple times. This led to the idea of using a tree structure for the predicates, where the leaf nodes in this tree are clients, the internal nodes are clauses. The filter engine can perform predication by propagating the updated flight record from the root node towards the leaves. This idea is being investigated.

The format of the messages from the distributor to the clients are currently defined by Java's serialization specifications. Using Java serialization simplifies many issues, but assumes the client will be written in Java. A better solution for cross-platform compatibility is to use XML for encoding these messages. There is plan underway to compress these messages to further reduce the bandwidth requirements.

There is still the issue of how to handle clients when the server load is fully utilized. One approach is to have a redirect message to send a client to a different server, one
with capacity to handle more requests. This introduces the idea of a network of servers, and the load balancing issues involved are nontrivial.
Appendix A

Alloy Models

A.1 Registration

model RegistrationModel
{
    domain {Client, FlightRecord, Request, Predicate}

    state {
        // the main relationship
        registers: Client -> FlightRecord

        sends: Client! -> Request!
        includes: Request -> Predicate!
        matches: Predicate -> FlightRecord
        rejects: Client -> static Predicate?
    }

    // Definition of register: to register for some flights,
    // client sends a request with a predicate that match those flights
    def registers {
        all c | no c.rejects -> c.registers = c.sends.includes.matches
        all c | some c.rejects -> c.registers =
            c.sends.includes.matches - c.rejects.matches
    }

    // valid: the register rule is defined correctly
    assert NoLeakPolicy {
        all c | no c.rejects.matches & c.registers
    }
}
A.2 Flight Record

model FlightRecordModel
{
    domain {FlightRecord, fixed FieldName, FieldValue}

    state {
        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    // f.id is just a shorthand for f.value[FlightID]
    def id {
        all f | all fid : FlightID | f.id = f.value[fid]
    }

    inv NameValuePaired {
        all f | all fn | fn in f.field -> some f.value[fn]
        all f | all fn | some f.value[fn] -> fn in f.field
    }

    // ASDI always provides a FlightID
    inv FlightIDRequired {
        all f | FlightID in f.field
    }

    // FlightRecords always have some Fields and Values
    // valid: since FlightID is required
    assert NoEmptyRecord {
        all f | some f.field & all fn: f.field | one f.value[fn]
    }
}
A.3 Predicate

model PredicateModel
{
    domain {FlightRecord, fixed FieldName, FieldValue,
            Predicate, Clause, Term}

    state {
        matches: Predicate -> FlightRecord
        clause: Predicate -> Clause
        term: Clause -> Term+
        tfield: Term -> FieldName!
        tvalue: Term -> FieldValue!

        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    def matches {
        all p | no p.clause -> p.matches = FlightRecord
        all p | some p.clause -> p.matches = {f | some c: p.clause |
            all t: c.term |
            all fn: t.tfield | f.value[fn] = t.tvalue}
    }

    // invalid: a predicate with no clause is a "wild-card"
    assert NoClause {
        all p | no p.clause -> no p.matches
    }

    inv FlightRecordInv {
        all f | all fn | fn in f.field -> one f.value[fn]
        all f | all fn | one f.value[fn] -> fn in f.field
    }

    def id {
        all f | all fid : FlightID | f.id = f.value[fid]
    }

    inv FlightIDRequired {
        all f | FlightID in f.field
    }
}
A.4 Flight Update

model FlightUpdateModel
{

domain {FlightRecord, FlightUpdate, fixed FieldName, FieldValue}

state {
field: FlightRecord -> FieldName
FlightID: fixed FieldName!
value[FieldName]: FlightRecord -> FieldValue?
id: FlightRecord -> FieldValue?
StateDB: FlightRecord

oldid: FlightUpdate -> FieldValue!
updates: FlightUpdate -> StateDB?
newStatus: FlightUpdate -> FlightRecord!
}

def updates {
	all fu | fu.updates = {fr | fr.id = fu.oldid}
}

def newStatus {
	all fu | fu.newStatus !in StateDB
}

// f.id is just a shorthand for f.value[FlightID]
def id {
	all f | all fid : FlightID | f.id = f.value[fid]
}

inv NameValuePaired {
	all f | all fn | fn in f.field -> one f.value[fn]
	all f | all fn | one f.value[fn] -> fn in f.field
}

// ASDI always provides a FlightID
inv FlightIDRequired {
	all f | FlightID in f.field
}

// FlightRecords always have some Fields and Values
// valid: since FlightID is required
assert NoEmptyRecord {
	all f | some f.field && all fn: f.field | one f.value[fn]
}
A.5 Template

model TemplateModel
{
  domain {FlightRecord, FlightUpdate, fixed FieldName, FieldValue, Template}

  state {
    selects: Template -> FieldName+
    field: FlightRecord -> FieldName
    FlightID: fixed FieldName!
    value[FieldName]: FlightRecord -> FieldValue?
    id: FlightRecord -> FieldValue?
    StateDB: FlightRecord

    oldid: FlightUpdate -> FieldValue!
    updates: FlightUpdate -> StateDB?
    newStatus: FlightUpdate -> FlightRecord!
  }

  def selects { all t | FlightID in t.selects }

  op applyTemplate (fu: FlightUpdate!, f: FlightRecord!,
    output: FlightUpdate!, t: Template!) {
    f in fu.updates
    output !in FlightUpdate
    output.oldid' = fu.oldid
    output.newStatus.field' = t.selects & f.field
    all fn: FieldName = output.newStatus.field' | no output.newStatus.value'[fn]
    all fn: output.newStatus.field' | output.newStatus.value'[fn] = f.value[fn]
    FlightUpdate' = FlightUpdate + output
    FlightRecord' = FlightRecord + output.newStatus
    all fr: FlightRecord | fr.field' = fr.field
    all fr: FlightRecord | all fn | fr.value'[fn] = fr.value[fn]
  }

  def updates {
    all fu | fu.updates = {fr | fr.id = fu.oldid}
  }

  def newStatus {
    all fu | fu.newStatus !in StateDB
  }

  // no duplicate flight ID within the database
  inv FlightIDUnique {
    all f1, f2 : StateDB | f1 != f2 <-> f1.id != f2.id
  }

  inv FlightRecordInv {
    all f | all fn | fn in f.field -> one f.value[fn]
    all f | all fn | one f.value[fn] -> fn in f.field
    all f | FlightID in f.field
  }
}
def id {
    all f | all fid : FlightID | f.id = f.value[fid]
}

A.6 State Database

model StateDatabaseModel
{
    domain {FlightRecord, fixed FieldName, FieldValue, FlightUpdate}

    state {
        StateDB: FlightRecord
        oldid: FlightUpdate -> FieldValue!
        updates: FlightUpdate -> StateDB?
        newStatus: FlightUpdate -> FlightRecord!

        field: FlightRecord -> FieldName
        FlightID: fixed FieldName!
        value[FieldName]: FlightRecord -> FieldValue?
        id: FlightRecord -> FieldValue?
    }

    // no duplicate flight ID within the database
    inv FlightIDUnique {
        all f1, f2 : StateDB | f1 != f2 <-> f1.id != f2.id
    }

    op addToStateDB (fu : FlightUpdate!) {
        all fr: StateDB | fr !in fu.updates
        FlightRecord' = FlightRecord + fu.updates
        StateDB' = StateDB + fu.updates
        all fr | fr.field' = fr.field
        all fr | all fn | fr.value'[fn] = fr.value[fn]
    }

    op updateFlight (fu: FlightUpdate!, oldF : StateDB!) {
        oldF in fu.updates
        StateDB' = StateDB - {f | f.id = fu.newStatus.id}
        FlightRecord' = FlightRecord
        oldF.field' = oldF.field + fu.newStatus.field
        all fn : fu.newStatus.field | oldF.value'[fn] = fu.newStatus.value[fn]
        all fn : FieldName - fu.newStatus.field | oldF.value'[fn] = oldF.value[fn]
        all fr: FlightRecord - oldF | fr.field' = fr.field
        all fr | all fn | fr.value'[fn] = fr.value[fn]
    }

    op removeFlight (fu: FlightUpdate!, f : StateDB!) {
        f in fu.updates
        StateDB' = StateDB - f
        FlightRecord' = FlightRecord + f
        all fr | fr.field' = fr.field
        all fr | all fn | fr.value'[fn] = fr.value[fn]
    }

    def updates {
        all fu | fu.updates = {fr | fr.id = fu.oldid}
    }
}
def newStatus {
   all fu | fu.newStatus !in StateDB
}

inv FlightRecordInv {
   all f | all fn | fn in f.field -> one f.value[fn]
   all f | all fn | one f.value[fn] -> fn in f.field
   all f | FlightID in f.field
}

def id {
   all f | all fid : FlightID | f.id = f.value[fid]
}
}
Appendix B

Code Object Models

The following object models were generated using Womble a tool for extraction object models from Java bytecode [JW99]. There are two types of lines being shown: solid edges show plain relations, and dotted edges show inheritance relations. In our code, we use the name “Predicater” to refer to the filter engine. That name is used in these object models.
Figure B-1: CAP Components
Figure D-2: Flight Record Fields

- Aircraft
- Origin
- Destination
- Fix
- Time Interval
- Delay
- Coordinate
- Flight ID
- Flight Record
- Flight Status
- Speed
- Status
- Time
- User Class
Appendix C

Module Dependency Diagrams

The following MDDs were generated using Womble. There are two types of dependencies being shown: solid edges show strong dependencies, and dotted edges show weak dependencies. In our code, we use the name "Predicater" to refer to the filter engine. That name is used in these MDDs.

![Module Dependency Diagram](image)

Figure C-1: CAP Component Dependencies
Figure C-2: CAP Server Modules
Figure C-3: Flight Record and Fields

Figure C-4: Parser Modules

Figure C-5: Distributor Modules
Figure C-6: Predicater Modules

Figure C-7: Registrar Modules
Appendix D

Code

D.1 Server.java

```
/**
 * Server.java
 *
 * contains the main method to start the CAP server
 */

import java.io.*;
import java.net.*;
import java.util.Properties;
import parser.*;
import predicater.*;
import registrar.*;
import distributor.*;
import util.*;

public class Server {
    static final String usage = "usage: java Server [ServerPort] [PasswordFile] [ASDIHostname] [ASDIPort]\n" +
                             "java Server [config.file]\n" +
                             "java Server";

    static final String defaultPropertiesFile = "server.properties";

    static final String defaultProperties[][] =
        /* Property Name, Default Value */
        {{"ServerPort", "8888"},
         {"PasswordFile", "cap.passwd"},
```
public static void main(String args[]) {
    String asdiHost=null, asdiPort=null, serverPort=null, passwdFile=null;

    switch (args.length) {
    case 0:
        // debug mode
        serverPort = defaultProperties[0][1];
        passwdFile = defaultProperties[1][1];
        asdiHost = null;
        asdiPort = null;
        break;
    case 1:
        // load parameters from config file
        Properties configs = new Properties();
        try {
            InputStream fileIn = new FileInputStream(args[0]);
            configs.load(fileIn);
        } catch (IOException ioe) {
            System.err.println("IOException caught loading file "+ defaultProperties[0]);
            System.err.println("using defaults");
        } finally {
            serverPort = configs.getProperty(defaultProperties[0][0], defaultProperties[0][1]);
            passwdFile = configs.getProperty(defaultProperties[1][0], defaultProperties[1][1]);
            asdiHost = configs.getProperty(defaultProperties[2][0], defaultProperties[2][1]);
            asdiPort = configs.getProperty(defaultProperties[3][0], defaultProperties[3][1]);
        }
        break;
    case 4:
        serverPort = args[0];
        passwdFile = args[1];
        asdiHost = args[2];
        asdiPort = args[3];
        break;
    default:
        System.err.println(usage);
        System.exit(1);
        break;
    }
    
    Socket skt = null;
    ServerSocket sskt = null;

    // by default, read messages from stdin
    InputStream theASDIFeed = System.in;
try {
    // parse arguments
    sskt = new ServerSocket(Integer.parseInt(serverPort));

    // theASDIFeed comes from a TCP stream
    if (asdiHost != null) {
        skt = new Socket(asdiHost, Integer.parseInt(asdiPort));
        theASDIFeed = skt.getInputStream();
    }
    } catch (IOException ioe) {
        System.err.println(ioe);
        System.exit(1);
    }

    System.err.println("ServerPort= " + sskt.getLocalPort());
    if (System.in == theASDIFeed)
        System.err.println("reading from System.in");

    Parser par;
    Predicater pred;
    Registrar reg;
    Distributor dist;

    try {
        par = new Parser(theASDIFeed);
        dist = new Distributor();
        pred = new Predicater(par, dist);
        reg = new Registrar(pred, sskt, passwdFile);

        new Thread(par).start();
        new Thread(pred).start();
        new Thread(dist).start();
        new Thread(reg).start();
    } catch (IOException ioe) {
        System.err.println(ioe);
        System.exit(2);
    } catch (BadFormatException bfe) {
        System.err.println(bfe);
        System.exit(3);
    }
}
}
D.2 DumbClient.java

```java
import java.io.*;
import java.net.*;
import parser.*;
import predicater.*;
import xml.*;

public class DumbClient {
    static final String usage = "usage: java DumbClient <CAP hostname> <CAP port>";

    public static void main (String args[]) {
        Socket skt = null;
        ObjectInputStream theCAPFeed = null;
        try {
            // parse arguments
            String host = args[0];
            int asdiport = Integer.parseInt(args[1]);

            skt = new Socket (host, asdiport);
            ObjectOutputStream outstream = new ObjectOutputStream (skt.getOutputStream());
            theCAPFeed = new ObjectInputStream (skt.getInputStream());

            // send username/password
            BufferedReader stdin =
                new BufferedReader(new InputStreamReader(System.in));
            System.out.println("username:");
            String username = stdin.readLine();
            System.out.println("password:");
            String passwd = stdin.readLine();
            System.out.println("predicate:");
            try {
                Predicate predicate =
                    registrar.PredicateParser.it.parse(stdin.readLine());

                outstream.writeObject(username);
                outstream.writeObject (passwd);
                outstream.writeObject (predicate);
            } catch (util.BadFormatExecption bfe) {
                System.err.println(bfe);
                System.exit(1);
            }
        }
    }
}
```
Template f = new Template();
f.setType(Template.T_COMPLETE);
outstream.writeObject(f);

System.err.println(theCAPFeed.readObject());
while (true) {
    String obj = (String) theCAPFeed.readObject();
    System.out.println(obj);
}

} catch (ArrayIndexOutOfBoundsException oobe) {
    // wrong number of commandline arguments
    System.err.println(usage);
    System.exit(1);
}

} catch (Exception uhe) {
    // error resolving the hostname
    System.err.println(uhe);
    System.exit(1);
}
}
D.3 distributor package

distributor/Distributor.java

package distributor;

import java.io.*;
import java.net. *
import registrar.*;
import util. *
import xml. *

public class Distributor implements Runnable
{
    public void run() {
        System.out.println("Distributor");
        while (true) {
            Task t = (Task) taskQueue.dequeue();
            if (t == null) {
                System.err.println("null task in task queue");
                continue;
            }
            // client maybe dead from messages sent before,
            // so don't bother with them.
            if (t.client.isAlive())
                try {
                    // System.out.println("writing");
                    // serialization right now is just convert to string
                    t.client.getOutputStream().writeObject(t.message.toXML());
                } catch (IOException ioe) {
                    // disconnect client and notify predicater
                    System.out.println("error writing");
                    t.client.kill();
                }
        }
        // System.out.println("Distributor thread ended");
    }

    public Distributor () {
        // maybe use a hashtable??
        taskQueue = new Queue(32);
    }

    public void send (Client c, FlightUpdate m) {
        taskQueue.enqueue(new Task (c, m));
    }

    public class Task {
        private Client client;
        private FlightUpdate message;

        public Task (Client c, FlightUpdate m) {
            client = c;
            message = m;
        }
    }
}

}
class Task {
    Client client;
    FlightUpdate message;

    Task (Client c, FlightUpdate m) {
        this.client = c;
        this.message = m;
    }
}
D.4 parser package

package parser;

import xml.*;
import util.*;
import java.io.*;

abstract class Message {
    // Defines an abstract Message object. This is the parent class
    // of all the different message types

    // denotes the beginning and end character for the parts of the
    // message header. Header ends with first space character
    private static final int[][] msgHeaderSpacing = {{0, 3}, {4, 11}, {12, 15}, {16, 17}};

    // The position of the sequence number in the header
    private static final int seqNumPos = 0;
    // The position of the time stamp in the header
    private static final int timeStampPos = 1;
    // The position of the facility id in the header
    private static final int facilityIdPos = 2;
    // The position of the message type in the header
    private static final int messageTypePos = 3;

    // Number of characters in header
    protected static final int headerSize = 18;

    private String seqNum;  // hexadecimal sequence number of msg
    private String timeStamp;  // Date-time stamp of msg as ddhhmmss
    private String facilityId;  // Originating facility id
    private String messageType;  // Type of ASDI message

    public Message(String msgBuf) {
        // Requires: A one line string containing the ASDI message header
        // Effects: Parses the header of the String and returns a new
        // Message object
        String[] headerData = new String[msgHeaderSpacing.length];
        for (int i = 0; i < msgHeaderSpacing.length; i++) {
            headerData[i] = msgBuf.substring(msgHeaderSpacing[i][0], msgHeaderSpacing[i][1] + 1).trim();
        }
        seqNum = headerData[seqNumPos];
        timeStamp = headerData[timeStampPos];
        facilityId = headerData[facilityIdPos];
        messageType = headerData[messageTypePos];
    }

    public String getType() {

return messageType;
}

// return the flight update object
// with parsed fields
public abstract FlightUpdate getFlightUpdate();

public static String extractMsgType(String msg) {
  // Requires: A one line ASDI msg string
  // Effects: Returns the two character String message type
  return msg.substring(msgHeaderSpacing[messageTypePos][0],
                      msgHeaderSpacing[messageTypePos][1] + 1);
}

public static Message parseMessage(String msg)
throws BadFormatException
{
  // Requires: A one line ASDI msg string
  // Effects: Returns a new Message object based on msg
  // Throws NoMsgTypeException if the message type is
  // unidentified
  try {
    String type = extractMsgType(msg);
    return parseMessage(msg, type);
  } catch (RuntimeException e) {
    // Any exception arising during string parsing
    // e.printStackTrace();
    throw new BadFormatException
      ("Received RuntimeException "+ e.toString()+
       "\n\t when parsing: " + msg);
  }
}

public static Message parseMessage(String msg, String type)
throws BadFormatException
{
  // Requires: A one line ASDI msg string and a two character message type
  // Effects: Returns a new Message object based on msg
  // Throws NoMsgTypeException if the message type is
  // unidentified. Throws BadFormatException if the msg
  // can't be parsed
  // Now return the correct type of message object

  if (type.equals("AF")) {
    return new AFMessage(msg);
  }
  else if (type.equals("AZ")) {
    return new AZMessage(msg);
  }
  else if (type.equals("DZ")) {
    return new DZMessage(msg);
  }
else if (type.equals("FZ")) {
    return new FZMessage(msg);
}
else if (type.equals("RZ")) {
    return new RZMessage(msg);
}
else if (type.equals("TZ")) {
    return new TZMessage(msg);
}
else if (type.equals("UZ")) {
    return new UZMessage(msg);
}
else if (type.equals("RT")) {
    return new RTMessage(msg);
}
else if (type.equals("TO")) {
    return new TOMessage(msg);
}
else if (type.equals("HB")) {
    return new HBMessage(msg);
}
throw new BadFormatException
    ("Unknown type " + type + " in:" + msg);
}

// given an array of field numbers, and their values in String format
// parse those fields and set them according to their field numbers
// in FlightUpdate
protected static void setFlightUpdateFields(FlightRecord frecord,
        int fieldNos[],
        String values[])
    throws BadFormatException
{
    for (int i = 0; i < fieldNos.length; i++) {
        if (values[i] != null) {
            Field f = parseField (fieldNos[i], values[i]);
            if (f != null)
                frecord.setField(fieldNos[i], f);
            // no exception yet since some fields we don't parse yet
            // else
            // throw new BadFormatException
            // "Error parsing field " + fieldNos[i] + " using:" + values[i]);
        }
    }
}

public static Field parseField (int fieldNo, String fieldStr)
    throws BadFormatException
{
    Field f = null;
    switch (fieldNo) {
        case FlightRecord.ID:
f = IDParser.it.parse(fieldStr);
break;
case FlightRecord.CID:
    f = CIDParser.it.parse(fieldStr);
    break;
case FlightRecord.ARRIVAL_TIME:
    {
        Time t = (Time) TimeParser.it.parse(fieldStr);
        t.setType(Time.T_ARRIVAL);
        f = t;
    }
    break;
case FlightRecord.COORD_TIME:
    {
        Time t = (Time) TimeParser.it.parse(fieldStr);
        t.setType(Time.T_COORDINATION);
        f = t;
    }
    break;
case FlightRecord.DEPART_TIME:
    {
        Time t = (Time) TimeParser.it.parse(fieldStr);
        t.setType(Time.T_DEPARTURE);
        f = t;
    }
    break;
case FlightRecord.PHYS_CLASS:
    f = PhysicalClassParser.it.parse(fieldStr);
    break;
case FlightRecord.SPEED:
    f = SpeedParser.it.parse(fieldStr);
    break;
case FlightRecord.USER_CLASS:
    f = UserClassParser.it.parse(fieldStr);
    break;
case FlightRecord.STATUS:
    f = StatusParser.it.parse(fieldStr);
    break;
case FlightRecord.DEST:
    {
        f = new Fix(fieldStr);
    }
    break;
case FlightRecord.ORIG:
    {
        f = new Fix(fieldStr);
    }
    break;
case FlightRecord.TRACK_POS:
    {
        int separator_pos = fieldStr.indexOf('/');
        String lat = fieldStr.substring(0, separator_pos);
        String lon = fieldStr.substring(separator_pos + 1);
        f = new Coordinate(lat, lon);
    }
```java
break;
case FlightRecord.REMARK:
    // no remark, not supported
    break;
case FlightRecord.ROUTE:
    break;
case FlightRecord.AIRCRAFT:
    // need aircraft type here
    break;
case FlightRecord.ALT_ASSIGNED:
    break;
case FlightRecord.ALT_REQUESTED:
    break;
case FlightRecord.COORD_FIX:
    break;
default:
    throw new BadFormatException("unknown field number");
}
return f;
}
}

/parser/Parser.java

/* XML Parser component
 */

package parser;

import java.io.*;
import util.*;
import xml.*;

public class Parser implements Runnable
{
    BufferedReader theFeedReader;

    Queue msgQueue; // a queue of parsed messages

    // need FIFO datastructure
    public Parser (InputStream theASDIFeed) {
        theFeedReader = new BufferedReader
            ( new InputStreamReader( theASDIFeed) );
        // use a message queue of at least 32 elements big
        msgQueue = new Queue(32);
    }
```
/* keep parsing the feed and put */ public void run() { try { System.err.println("Parser"); // throw away the first line read from ASDI // since that line is usually an incomplete message String buf = theFeedReader.readLine(); Message msg = null; while ((buf = theFeedReader.readLine() != null) { try { msg = Message.parseMessage(buf); // HB messages are left out, they have no meaning if (!(msg instanceof HBMessage)) msgQueue.enqueue(msg.getMessage flightUpdate()); } catch (BadFormatException exp) { System.err.println(exp); } System.err.println("ASDI stream ended"); } catch (IOException ioe) { System.err.println("IOException reading ASDI: "+ioe); System.exit(3); } System.err.println("Parser thread ended"); } } /* could return null if there are no messages in queue */ public FlightUpdate nextFlightUpdate() { return (FlightUpdate) msgQueue.dequeue(); } public boolean ready() { return !msgQueue.empty(); } }
import util.*;

class AFMessage extends Message {
   // Defines an AF Message object. This message provides
   // amendments to the flight plan.
   FlightUpdate flightUpdate;

   public AFMessage(String msg)
      throws BadFormatException
   {
      // Requires: A valid AF message
      // Effects: Returns a new AFMessage object with fields set to values in msg
      super(msg); // Parses the header portion of message

      StringTokenizer tokens = new StringTokenizer
         (msg.substring(headerSize));

      // first token is ACID
      String IDStr = tokens.nextToken();

      // now take the ID part of ACID
      int slash = IDStr.lastIndexOf('/');
      if (slash != -1)
      IDStr = IDStr.substring(0, slash);

      ID id = (ID) parseField(FlightRecord.ID, IDStr);

      flightUpdate = new FlightUpdate(id);

      // the same id is used unless the rest of the message corrects it
      flightUpdate.setField(FlightRecord.ID, id);

      Field f = parseField (FlightRecord.ORIG, tokens.nextToken());
      flightUpdate.setField(FlightRecord.ORIG, f);

      f = parseField (FlightRecord.DEST, tokens.nextToken());
      flightUpdate.setField(FlightRecord.DEST, f);

      while (tokens.hasMoreTokens()) {
         String fieldRef = tokens.nextToken();
         String fieldData = tokens.nextToken();
         try {
            int field = Integer.parseInt(fieldRef);
            field = translateNASFieldNo (field);
            if (field == -1)
            throw new BadFormatException
               ("error translating field reference: " + fieldRef);
            if (field == FlightRecord.ID) {
               // amendment to ID
               slash = fieldData.lastIndexOf('/');
               if (slash != -1) {
                  f = parseField (FlightRecord.CID, fieldData.substring(slash+1));
               }
            }
         } catch (NumberFormatException e) {
            throw new BadFormatException
               ("error parsing field value: " + fieldRef);
         }
      }
   }
}
flightUpdate.setField(FlightRecord.CID, f);
fieldData = fieldData.substring(0, slash);
}

f = parseField(field, fieldData);
flightUpdate.setField(field, f);
} catch (NumberFormatException nfe) {
    throw new BadFormatException("bad field reference " + fieldRef + " in AF");
}
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

private static final int FieldNumTable[] = {
    -1, // 0
    -1, // 1
    FlightRecord.ID, // 2
    FlightRecord.AIRCRAFT, // 3
    -1, // 4
    FlightRecord.SPEED, // 5
    FlightRecord.COORD_FIX, // 6
    FlightRecord.COORD_TIME, // 7
    FlightRecord.ALT_ASSIGNED, // 8
    FlightRecord.ALT_REQUESTED, // 9
    FlightRecord.ROUTE, // 10
    FlightRecord.REMARK, // 11
    -1, // 12
    -1, // 13
    -1, // 14
    -1, // 15
    -1, // 16
    -1, // 17
    -1, // 18
    -1, // 19
    -1, // 20
    -1, // 21
    -1, // 22
    FlightRecord.TRACK_POS, // 23
    -1, // 24
    -1, // 25
    FlightRecord.ORIG, // 26
    FlightRecord.DEST, // 27
    FlightRecord.ARRIVAL_TIME // 28
};

private int translateNASFieldNo(int NASField) {
    return FieldNumTable[NASField];
}
package parser;

import java.util.*;
import xml.*;
import util.*;

class AZMessage extends Message {
    // Defines an AZ Message object. This message announces
    // flight arrivals

    FlightUpdate flightUpdate;

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.ORIG,
        FlightRecord.DEST,
        FlightRecord.ARRIVAL_TIME,
        FlightRecord.CID
    };

    public AZMessage(String msg)
        throws BadFormatException
    {
        // Requires: A valid AZ message
        // Effects: Returns a new AZMessage object with fields
        // set to values in msg

        super(msg); // Parses the header portion of message

        StringTokenizer tokens = new StringTokenizer
            (msg.substring(headerSize));

        String[] msgParts = new String[FIELDS.length];
        int i = 0;
        while (tokens.hasMoreTokens()) {
            msgParts[i] = tokens.nextToken();
            i++;
        }

        // break ACID into ID/CID
        int slash = msgParts[0].lastIndexOf('/');
        if (slash != -1) {
            msgParts[msgParts.length-1] = msgParts[0].substring(slash+1);
            msgParts[0] = msgParts[0].substring(0, slash);
        }

        flightUpdate = new FlightUpdate(new ID(msgParts[0]));
        setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);
        flightUpdate.setField(FlightRecord.STATUS, Status.COMPLETED);

        // should we set the status field here? if yes, which status is it?
```java
/ *
  Time arr = (Time) flightUpdate.getField(FlightRecord.ARRIVAL_TIME);

  if (arr.isMode(Time.M.ACTUAL))
      flightUpdate.setField(FlightRecord.STATUS, Status.COMPLETED);
 */

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

```

### parser/DZMessage.java

```java
package parser;
import java.util.*;
import xml.*;
import util.*;

class DZMessage extends Message {
    // Defines an DZ Message object. This message announces
    // flight departures.

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.AIRCRAFT,
        FlightRecord.ORIG,
        FlightRecord.DEPART_TIME,
        FlightRecord.DEST,
        FlightRecord.ARRIVAL_TIME,
        FlightRecord.CID
    };

    FlightUpdate flightUpdate;

    public DZMessage(String msg)
        throws BadFormatException
    {
        // Requires: A valid DZ message
        // Effects: Returns a new DZMessage object with fields
        // set to values in msg

        super(msg); // Parses the header portion of message

        StringTokenizer tokens = new StringTokenizer
            (msg.substring(headerSize));
```
String[] msgParts = new String[FIELDS.length];
int i = 0;
while (tokens.hasMoreTokens()) {
    msgParts[i] = tokens.nextToken();
    i++;
}
// break ACID into ID/CID
int slash = msgParts[0].lastIndexOf('/');
if (slash != -1) {
    msgParts[msgParts.length-1] = msgParts[0].substring(slash+1);
    msgParts[0] = msgParts[0].substring(0, slash);
}

flightUpdate = new FlightUpdate(new ID(msgParts[0]));
setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

package parser;

import java.util.*;
import xml.*;
import util.*;

class FZMessage extends Message {
    // Defines an FZ Message object. This message announces
    // flight plan information.

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.AIRCRAFT,
        FlightRecord.SPEED,
        FlightRecord.COORD_FIX,
        FlightRecord.COORD_TIME,
        FlightRecord.ROUTE,
        FlightRecord.ALT_ASSIGNED,
        FlightRecord.CID
    };

    FlightUpdate flightUpdate;
    public FZMessage(String msg)
        throws BadFormatException
    {
        // Requires: A valid FZ message
// Effects: Returns a new FZMessage object with fields set to values in msg

super(msg); // Parses the header portion of message

StringTokenizer tokens = new StringTokenizer(msg.substring(headerSize));

String[] msgParts = new String[FIELDS.length];
int i = 0;
while (tokens.hasMoreTokens()) {
    msgParts[i] = tokens.nextToken();
    i++;
}

// break ACID into ID/CID
int slash = msgParts[0].lastIndexOf('/');
if (slash != -1) {
    msgParts[msgParts.length-1] = msgParts[0].substring(slash+1);
    msgParts[0] = msgParts[0].substring(0, slash);
}

flightUpdate = new FlightUpdate(new ID(msgParts[0]));
setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}


package parser;
import java.util.*;
import xml.*;
import util.*;

class RZMessage extends Message {
    // Defines an RZ Message object. This message announces
    // flight cancellations.

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.ORIG,
        FlightRecord.DEST,
        FlightRecord.CID,
    };

    FlightUpdate flightUpdate;
public RZMessage(String msg) throws BadFormatException
{
    // Requires: A valid RZ message
    // Effects: Returns a new RZMessage object with fields set to values in msg
    super(msg); // Parses the header portion of message

    StringTokenizer tokens = new StringTokenizer(msg.substring(headerSize));

    String[] msgParts = new String[FIELDS.length];
    int i = 0;
    while (tokens.hasMoreTokens()) {
        msgParts[i] = tokens.nextToken();
        i++;
    }
    // break ACID into ID/CID
    int slash = msgParts[0].lastIndexOf('/');
    if (slash != -1) {
        msgParts[msgParts.length - 1] = msgParts[0].substring(slash + 1);
        msgParts[0] = msgParts[0].substring(0, slash);
    }

    flightUpdate = new FlightUpdate(new ID(msgParts[0]));
    setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);
    // not done yet, need to set the status field to cancelled
    flightUpdate.setField(FlightRecord.STATUS, Status.CANCELLED);
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

package parser;

import java.util.*;
import xml.*;
import util.*;

class TZMessage extends Message {
    // Defines an TZ Message object. This message announces
    // position updates.

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.SPEED,
        FlightRecord. ALT_ASSIGNED,
    }
FlightRecord.TRACK.POS,
FlightRecord.CID
);

FlightUpdate flightUpdate;

public TZMessage(String msg)
    throws BadFormatException
{
    // Requires: A valid TZ message
    // Effects: Returns a new TZMessage object with fields set to values in msg
    super(msg); // Parses the header portion of message

    StringTokenizer tokens = new StringTokenizer(msg.substring(headerSize));

    String[] msgParts = new String[FIELDS.length];
    int i = 0;
    while (tokens.hasMoreTokens()) {
        msgParts[i] = tokens.nextToken();
        i++;
    }
    // break ACID into ID/CID
    int slash = msgParts[0].lastIndexOf('/');
    if (slash != -1) {
        msgParts[msgParts.length-1] = msgParts[0].substring(slash+1);
        msgParts[0] = msgParts[0].substring(0, slash);
    }

    flightUpdate = new FlightUpdate(new ID(msgParts[0]));
    setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}
}

package parser;

import java.util.*;
import xml.*;
import util.*;

class UZMessage extends Message {
    // Defines a UZ Message object. This message announces
    // boundary crossings.
public static final int FIELDS[] = {
    FlightRecord.ID,
    FlightRecord.AIRCRAFT,
    FlightRecord.SPEED,
    FlightRecord.COORD_FIX,
    FlightRecord.COORD_TIME,
    FlightRecord.ALT_ASSIGNED,
    FlightRecord.ROUTE,
    FlightRecord.CID
};

FlightUpdate flightUpdate;

public UZMessage(String msg)
    throws BadFormatException
{
    // Requires: A valid UZ message
    // Effects: Returns a new UZMessage object with fields set to values in msg
    super(msg); // Parses the header portion of message

    StringTokenizer tokens = new StringTokenizer(msg.substring(headerSize));
    // StringTokenizer tokens = new StringTokenizer(msg);

    String[] msgParts = new String[FIELDS.length];
    int i = 0;
    while (tokens.hasMoreTokens()) {
        msgParts[i] = tokens.nextToken();
        i++;
    }
    // break ACID into ID/CID
    int slash = msgParts[0].lastIndexOf('/');
    if (slash != -1) {
        msgParts[msgParts.length - 1] = msgParts[0].substring(slash + 1);
        msgParts[0] = msgParts[0].substring(0, slash);
    }

    flightUpdate = new FlightUpdate(new ID(msgParts[0]));
    setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}
}
import java.util.*;
import xml.*;
import util.*;

class HBMessage extends Message {
    // Defines a HB Message object. This message announces
    // that the ASDI server is running.
    
    public HBMessage(String msg) {
        super(msg);
        // Requires: A valid HB message
        // Effects: Returns a new HBMessage object
        // with fields set to values in msg
    }

    public FlightUpdate getFlightUpdate() {
        return null;
    }

    public String parseToXML() {
        return new String("This is an HB message");
    }
}

package parser;

import xml.*;
import util.*;

class RTMessage extends Message {
    // Defines an RT Message object. This message announces
    // ETMS data and predictions. It is not wholly implemented.
    
    /** The message information. Each sub array gives a different piece
    * of the message. The format is: {start index, end index, byte format}
    * where byte format is 0,1,2, or 3. A 0 means there is no special
    * encoding while the others mean 1-byte, 2-byte and 3-byte encoding.
    * Note that the message information leaves off the rt tag which is
    * parsed in the header. Also, blank spaces are not designated in this
    * array. *
    */
    private static final int[][] msgFormat = {
        {4, 10, 0},
        {11, 13, 0},
        {16, 21, 0},
        {22, 24, 3},
        {25, 27, 3},
        {28, 30, 3},
        {31, 33, 3},
    };
}
{34, 36, 3},
{37, 39, 3},
{40, 40, 0},
{41, 41, 0},
{42, 42, 0},
{43, 44, 2},
{45, 45, 1},
{46, 46, 1},
{47, 47, 1},
{48, 48, 1},
{49, 50, 2},
{51, 53, 3},
{54, 56, 3},
{57, 59, 3},
{60, 62, 3},
{63, 66, 0},
{67, 70, 0},
{71, 71, 0},
{72, 72, 1}};

/** This is the mapping table to convert 8 bits into 6 */
private static final short eight_to_six[] = {
    // 00
    58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 0, 58, 58,
    58, 58, 58,
    // 16
    58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58,
    58, 58, 58,
    // 32
    1, 43, 44, 40, 58, 46, 47, 48, 49, 50, 42, 41, 51,
    52, 39, 38,
    // 48
    2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 53, 54, 55,
    56, 57, 58,
    // 64
    59, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26,
    // 80
    27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 60, 58,
    61, 58, 58,
    // 96
    58, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26,
    // 102
    27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 62, 45,
    63, 58, 58,
    // 128
    58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58,
    58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58,
    58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58,
/** The different fields of the message. */
private String acid, cid, arrFix, depDay, edt, cdt, eta, cta,
arrFixTime, flightUpdate, acPhysClass, acUserClass, wayCount,
sectorsCount, fixCount, airCount, centerCount, routeCount,
flightIndex, ogtd, ogta, depAir, arrAir, depCenter, msgType,
waypoints, sectors, fixes, airways, centers, route;

FlightUpdate flightUpdate;

public RTMessage(String msg)
throws BadFormatException
{
    // Requires: A valid RT message
    // Effects: Returns a new RTMessage object with fields set to values in msg
    super(msg); // Parses the header portion of message

    // remove msg header
    msg = msg.substring(headerSize-2);

    String[] msgParts = new String[msgFormat.length];
    for (int i = 0; i < msgFormat.length; i++) {
        String part = msg.substring(msgFormat[i][0] - 1,
        msgFormat[i][1]);
        //System.out.print(part + " = ");
        msgParts[i] = part; // decodeMsg (part, msgFormat[i][2]);
        //System.out.println(msgParts[i]);
    }

    ID id = (ID) parseField(FlightRecord.ID, msgParts[0]);
    flightUpdate = new FlightUpdate(id);

    flightUpdate.setField(FlightRecord.ID, id);

    Field f = parseField(FlightRecord.CID, msgParts[1]);
    flightUpdate.setField(FlightRecord.CID, f);

    //    arrFix = msgParts[2];
    //    depDay = msgParts[3];
    //    edt = msgParts[4];
    //    cdt = msgParts[5];
    //    eta = msgParts[6];
    //    cta = msgParts[7];
arrFixTime = msgParts[8];  
flightUpdate = msgParts[9];  
f = parseField(FlightRecord.STATUS, msgParts[9]);  
flightUpdate.setField(FlightRecord.STATUS, f);  
acPhysClass = msgParts[10];  
acUserClass = msgParts[11];  
wayCount = msgParts[12];  
sectorsCount = msgParts[13];  
fixCount = msgParts[14];  
airCount = msgParts[15];  
centerCount = msgParts[16];  
routeCount = msgParts[17];  
int offset = msg.length() - decodeTwoBytes(msgParts[17]);  
f = parseField(FlightRecord.ROUTE, msg.substring(offset));  
flightUpdate.setField(FlightRecord.ROUTE, f);  
flightIndex = msgParts[18] + msgParts[19];  
ogtdd = msgParts[20];  
ogtla = msgParts[21];  
depCenter = msgParts[24];  
msgType = msgParts[25];  
/*  
   // Now parse the variable length fields  
   waypoints = ;  
   sectors = msgParts[ ];  
   fixes = msgParts[ ];  
   airways = msgParts[ ];  
   centers = msgParts[ ];  
   route = msgParts[ ];  
*/  
/** Requires: A field of the RT message and an integer 0,1,2 or 3  
* Effects: Decodes the msgPart using the correct byte encoding method  
* according to the codingType */  
public static String decodeMsg(String msgPart, int codingType) {  
    switch (codingType) {  
    case 0: return msgPart;  
    case 1: return String.valueOf(decodeOneByte(msgPart));  
    case 2: return String.valueOf(decodeTwoBytes(msgPart));  
    case 3: return String.valueOf(decodeThreeBytes(msgPart));  
    }  
    return msgPart;  
}  
/** Requires: A piece of the message coded in one byte format
*  **Effects:** Takes a one byte piece of the message and decodes it into an
*  integer */
public static int decodeOneByte(String msgPart) {
    byte[] bytes = msgPart.getBytes();
    return (int) (eight_to_six[bytes[0]] - 1);
}

/**  Requires: A piece of the message coded in two byte format
*  **Effects:** Takes a two byte piece of the message and decodes it into an
*  integer */
public static int decodeTwoBytes(String msgPart) {
    byte[] bytes = msgPart.getBytes();
    return (int) ((eight_to_six[bytes[0]] - 1) * 62 +
                  (eight_to_six[bytes[1]] - 1));
}

/**  Requires: A piece of the message coded in three byte format
*  **Effects:** Takes a three byte piece of the message and decodes it into an
*  integer */
public static int decodeThreeBytes(String msgPart) {
    byte[] bytes = msgPart.getBytes();
    return (int) (((eight_to_six[bytes[0]] - 1) * 62 +
                    (eight_to_six[bytes[1]] - 1)) * 62 +
                   (eight_to_six[bytes[2]] - 1));
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

package parser;

import java.util.*;
import xml.*;
import util.*;

class TOMessage extends Message {
    // Defines a TO Message object. This message announces
    // oceanic position reports.

    public static final int FIELDS[] = {
        FlightRecord.ID,
        FlightRecord.SPEED,
        FlightRecord.ORIG,
        FlightRecord.DEST
    };
    
    parser/TOMessage.java

    package parser;

    import java.util.*;
    import xml.*;
    import util.*;

    class TOMessage extends Message {
        // Defines a TO Message object. This message announces
        // oceanic position reports.

        public static final int FIELDS[] = {
            FlightRecord.ID,
            FlightRecord.SPEED,
            FlightRecord.ORIG,
            FlightRecord.DEST
        };
    }
FlightUpdate flightUpdate;
public TOMessage(String msg) throws BadFormatException {
    // Requires: A valid TO message
    // Effects: Returns a new TOMessage object with fields set to values in msg
    // first token is the msg header
    super(msg);

    StringTokenizer tokens = new StringTokenizer(msg.substring(headerSize));
    int tokenCount = tokens.countTokens();

    String msgParts[] = new String[7];
    msgParts[0] = tokens.nextToken();
    msgParts[1] = tokens.nextToken();

    int i = 2, j = 2;
    do {
        String throwaway = tokens.nextToken() + ' ' +
            tokens.nextToken() + ' ' +
            tokens.nextToken();
        i = i + 3;
        j++;
    } while (i != (tokenCount - 1));

    String dept_arr = tokens.nextToken();

    if (dept_arr.equals("--")) {
    } else if (dept_arr.charAt(0) == '-') {
        msgParts[2] = null;
        msgParts[3] = dept_arr.substring(1);
        if (msgParts[3].equals("-"))
            msgParts[3] = null;
    } else {
        msgParts[2] = dept_arr.substring(0, 4);
        msgParts[3] = dept_arr.substring(4);
        if (msgParts[3].equals("-"))
            msgParts[3] = null;
    }
    // break ACID into ID/CID
    int slash = msgParts[0].lastIndexOf('/');
    if (slash != -1) {
        msgParts[msgParts.length - 1] = msgParts[0].substring(slash + 1);
        msgParts[0] = msgParts[0].substring(0, slash);
    }
flightUpdate = new FlightUpdate(new ID(msgParts[0]));
setFlightUpdateFields(flightUpdate.getFlightRecord(), FIELDS, msgParts);
}

public FlightUpdate getFlightUpdate() {
    return flightUpdate;
}

public String parseToXML() {
    return new String("TO messaged not yet parsed");
}
D.5  predicater package

predicater/Predicater.java

/*
* Predicater
* figures out which clients to send new FlightStatus to
* by evaluating the predicates for each client
*/

package predicater;

import java.io.*;
import java.net.*;
import java.util.*;
import parser.*;
import distributor.*;
import xml.*;
import util.*;
import registrar.*;
import java.util.Enumeration;

public class Predicater implements Runnable
{
    Parser parser;
    Distributor distributor;
    StateDB s;

    // need linkedlist, but that's jdk1.2
    Vector clients;
    Queue newClients;

    public Predicater (Parser p, Distributor d)
    {
        parser = p;
        distributor = d;
        s = new StateDB();
        clients = new Vector();
        newClients = new Queue(16);
    }

    public void addClient (Client c)
    {
        /* send every flight status that is in the
         * the state that matches the client's predicate
         * to the client
         */

        newClients.enqueue(c);
    }
public void run() {
    FlightUpdate m = null;
    System.out.println("Predicater");

    while (true) {
        if (!newClients.empty())
            serviceNewClient((Client) newClients.dequeue());

        m = parser.nextFlightUpdate();

        if (m == null) {
            System.err.println("null FlightUpdate returned by Parser");
            continue;
        }

        FlightRecord f = updateState(m);

        // evaluate all the predicates
        // generate a list of clients for the distributor
        for (int i = 0; i < clients.size(); i++) {
            Client c = (Client) clients.elementAt(i);
            if (c.isAlive()) {
                try {
                    if (c.wants(f))
                        distributor.send(c, c.getTemplate().apply(f, m));
                } catch (RuntimeException rte) {
                    // client sent a bad predicate or filter
                    c.kill();
                }
            } else {
                // remove c from list
                clients.removeElementAt(i);
            }
        }
    }
}

private void serviceNewClient (Client c) {
    Enumeration flights = s.flights();
    FlightRecord f;
    while (flights.hasMoreElements()) {
        f = (FlightRecord) flights.nextElement();
        if (c.isAlive()) {
            if (c.wants(f))
                distributor.send(c, c.getTemplate().apply(f, null));
        }
    }
    if (c.isAlive())
        clients.addElement(c);
private FlightRecord updateState (FlightUpdate m) {
// returns the updated, and complete FlightRecord

    FlightRecord f = s.getFlight (m.getID());
    FlightRecord newf = m.getFlightRecord();

    if (f == null) {
        // our policy is to add anything for now
        if (!newf.completed())
            s.addFlight (newf);
        return newf;
    }

    // an ID change
    if (!f.getID().equals(newf.getID()))
        if (s.getFlight(newf.getID()) != null) {
            System.err.println(“conflict”);
            System.err.println(“-----new flight------”);
            System.err.println(newf);
            System.err.println(“-----old flight------”);
            System.err.println(s.getFlight(newf.getID()));
            System.exit(3);
        }

        s.removeFlight(f.getID());
        f.merge(newf);
        if (!f.completed())
            s.addFlight (f);
        return f;
    }

    f.merge(newf);

    if (f.completed())
        s.removeFlight(f.getID());

    return f;
}

package predicater;

import java.io.*;
import xml.*;
import java.util.*;
public class Predicate implements Serializable
{
    Vector clauses;

    public Predicate () {
        clauses = new Vector();
    }
    public int size() {
        return clauses.size();
    }

    public void addClause (Clause c) {
        clauses.addElement (c);
    }

    // a predicate matches a flight
    // iff one of the terms matches this flight
    public boolean match (FlightRecord flight) {
        if (clauses.size() == 0)
            return true;

        for (int i = 0; i < clauses.size(); i++) {
            Clause c = (Clause) clauses.elementAt(i);
            if (c.match (flight))
                return true;
        }

        return false;
    }
}

predicater/Clause.java

package predicater;

import xml.*;
import java.util.*;

public class Clause implements java.io.Serializable
{
    Vector terms;

    // a term must have at least 1 clause
    public Clause (Term t) {
        terms = new Vector();
        terms.addElement (t);
    }

    public void addTerm (Term t) {

terms.addElement (t);

// a clause matches a flight
// if all of its clauses match this flight
public boolean match (FlightRecord flight) {
    for (int i=0; i < terms.size(); i++) {
        Term t = (Term) terms.elementAt(i);
        if (!t.match (flight))
            return false;
    }
    return true;
}

package predicater;
import xml.*;
import java.util.*;
public class Term implements java.io.Serializable {
    int field;
    Operator op;
    Field value;

    public Term (int f, Operator o, Field v) {
        field = f;
        op = o;
        value = v;
    }

    public boolean match (FlightRecord flight) {
        return op.eval (flight, field, value);
    }
}

package predicater;
import xml.*;
public abstract class Operator implements java.io.Serializable
public abstract boolean eval(FlightRecord f, int field, Field value2);

public class OpEqual extends Operator
{
    public static final OpEqual it = new OpEqual();

    public boolean eval(FlightRecord flight, int field, Field value2)
    {
        Field value1 = flight.getField(field);
        if (value1 == null)
            return false;
        return value1.equals(value2);
    }
}

public class OpIsAirline extends Operator
{
    public boolean eval(FlightRecord flight, int field, Field value2)
    {
        if (field != FlightRecord.ID)
            return false;
        return ((ID) flight.getField(field)).isAirline(((Airline) value2));
    }
}
predicater/OpNot.java

```java
package predicater;

import xml.*;
public class OpNot extends Operator
{
    Operator op;

    public OpNot (Operator op)
    {
        this.op = op;
    }

    public boolean eval(FlightRecord f, int field, Field value2) {
        return !op.eval(f, field, value2);
    }
}
```

predicater/StateDB.java

```java
package predicater;

import parser.*;
import java.util.Hashtable;
import java.util.Enumeration;
import xml.*;

class StateDB
{
    Hashtable ht;

    public StateDB()
    {
        ht = new Hashtable();
    }

    public void addFlight (FlightRecord f) {
        // add a flight to the state
        // removes any flight with the same ID
        ht.put ((ID) f.getField(FlightRecord.ID), f);
        System.err.println("state size: " + ht.size());
    }

    public FlightRecord getFlight (ID fid) {
        return (FlightRecord) ht.get (fid);
    }

    public Enumeration flights () {
        return ht.elements();
    }
```
public FlightRecord removeFlight (ID fid) {
    System.err.println("-state size: " + ht.size());
    return (FlightRecord) ht.remove (fid);
}

public synchronized void addField (int field) {
    pass[field] = true;
}

public void setType (boolean type) {
    complete = type;
}

public FlightUpdate apply (FlightRecord flight, FlightUpdate oldUpdate) {
FlightUpdate rtn;

if (oldUpdate != null)
    rtn = new FlightUpdate(oldUpdate.getID());
else
    rtn = new FlightUpdate(flight.getID());

if (!complete && (oldUpdate != null)) {
    flight = oldUpdate.getFlightRecord();
}

int [] avail = flight.getAvailableFields();
for (int i=0; i < avail.length; i++)
    if (pass[avail[i]])
        rtn.setField(avail[i], flight.getField(avail[i]));

return rtn;
}
D.6 registrar package

registrar/Registrar.java

package registrar;

import java.io.*;
import java.net.*;
import predicater.*;
import distributor.*;
import util.*;

public class Registrar implements Runnable
{
    Predicater predicater;
    ServerSocket sskt;
    UsersDB userdb;

    public Registrar (Predicater p, ServerSocket s, String passwdFile)
             throws IOException, BadFormatException
    {
        predicater = p;
        sskt = s;
        userdb = new UsersDB(passwdFile);
    }

    public void run() {
        System.out.println("Registrar");
        while (true) {
            try {
                // listen for new connections
                Socket sock = sskt.accept();
                System.out.println("Client connected from " + sock.getInetAddress() + " "+ sock.getPort());
                Client c;
                ObjectInputStream clientIn =
                        new ObjectInputStream(sock.getInputStream());
                ObjectOutputStream clientOut =
                        new ObjectOutputStream(sock.getOutputStream());
                try {
                    // authenticate the client
                    String username = (String) clientIn.readObject();
                    String passwd = (String) clientIn.readObject();
                    if (userdb.authenticate(username, passwd)) {
                        Predicate reject = userdb.getRejectPredicateFor(username);
                        Predicate accept = (Predicate) clientIn.readObject();
                        Template template = (Template) clientIn.readObject();
                        c = new Client(sock, reject, accept, template, clientOut);
                    }
                } catch (IOException e) {
                    // handle exception
                }
            } finally {
                clientIn.close();
                clientOut.close();
                sock.close();
            }
        }
    }
}
// add to predicater
predicater.addClient(c);
clientOut.writeObject("registration complete");
}
else {
    // send message to client
    clientOut.writeObject("invalid authentication");
    clientOut.close();
    sock.close();
    System.err.println("failed authentication using username: "+ username);
}
}
} catch (ClassNotFoundException cnfe) {
    System.err.println("Punting bad request from "+ sock.getInetAddress() + "+" + sock.getPort());
    sock.close();
    continue;
}
} catch (IOException io) {
    System.err.println("IOException Caught " + io.toString());
}
} // System.out.println("Registrar thread ended");
}

registrar/Client.java

package registrar;

import java.io.*;
import java.net.*;
import predicater.*;
import xml.*;

public class Client {
    Socket skt;
    boolean alive = true;
    ObjectOutputStream objout = null;
    Predicate accept;
    Predicate reject;
    Template f;

    public Client (Socket s, Predicate reject)
        throws IOException, BadRequestException
        { public Client (Socket s, Predicate reject)
```java
{ this.reject=reject;
skt = s;
objout = new ObjectOutputStream (skt.getOutputStream());
ObjectInputStream instream =
    new ObjectInputStream(skt.getInputStream());

    // get the client's request
    try {
        accept = (Predicate) instream.readObject();
        f = (Template) instream.readObject();
    }
    catch (ClassNotFoundException cnfe) {
        throw new BadRequestException();
    }
    catch (ClassCastException cce) {
        throw new BadRequestException();
    }
}

public Client (Socket s, Predicate reject, Predicate accept,
    Template f, ObjectOutputStream clientout)
{
    skt = s;
    this.accept=accept;
    this.reject=reject;
    this.f=f;
    this.objout = clientout;
}

public boolean isAlive() {
    return alive;
}

public void kill() {
    try {
        skt.close();
    } catch (IOException ioe) {
    } finally {
        alive = false;
    }
}

public boolean wants (FlightRecord flight) {
    if (reject != null)
        if (reject.match(flight))
            return false;
    return accept.match(flight);
}

public Template getTemplate () {
    return f;
}
```
public ObjectOutputStream getOutputStream () {
    return objout;
}

package registrar;
import predicater.*;
import java.util.*;
import java.io.*;
import util.*;

class UsersDB {
    private class UserProfile {
        String username;
        String passwd;
        Predicate reject;

        UserProfile (String n, String p, Predicate r) {
            username = n;
            passwd = p;
            reject = r;
        }
    }

    Hashtable userdb;

    UsersDB (String passwdFile) throws IOException, BadFormatException {
        Properties users;
        users = new Properties();
        InputStream fileln = new FileInputStream(passwdFile);
        users.load(fileln);

        Enumeration allusers = users.keysO;
        userdb = new HashtableO;

        while (allusers.hasMoreElements()) {
            String username = (String) allusers.nextElement();
            String passwd = getPasswdPart(users.getProperty(username));
            String predstr = getPredicatePart(users.getProperty(username));
            if (predstr == null)
                throw new BadFormatException("no reject predicate for " + username);
            Predicate reject = PredicateParser.it.parse(predstr);
            if (reject.size() == 0)
reject = null;
userdb.put(username, new UserProfile (username, passwd, reject));
}
}

public boolean authenticate (String username, String passwd) {
UserProfile user = (UserProfile) userdb.get(username);
if (user == null)
    return false;

return passwd.equals(user.passwd);
}

public Predicate getRejectPredicateFor (String username) {
UserProfile user = (UserProfile)userdb.get(username);
if (user == null)
    return null;

return user.reject;
}

private String getPasswdPart (String userProfile) {
    int separator = userProfile.indexOf('
');
    if (separator == -1)
        return userProfile;
    else
        return userProfile.substring(0, separator);
}

private String getPredicatePart (String userProfile) {
    int separator = userProfile.indexOf('\n');
    if (separator == -1)
        return null;
    else
        return userProfile.substring(separator+1);
}

public class PredicateParser
{
    public static final PredicateParser it = new PredicateParser();

    public Predicate parse (String predicateStr)
    throws BadFormatException
    {
        Vector clauses = new Vector();
        int start = 0, end = 0;
        predicateStr = predicateStr.trim();
        while (!predicateStr.equals("")) {
            start = predicateStr.indexOf('{');
            end = predicateStr.indexOf('}');
            if (end <= start)
                throw new BadFormatException("error parsing predicate " + predicateStr);
            clauses.addElement(parseClause (predicateStr.substring(start+1, end)));
            predicateStr = predicateStr.substring(end+1).trim();
        }
        Predicate rtn = new Predicate();
        for (int i = 0 ; i < clauses.size(); i++)
            rtn.addClause((Clause) clauses.elementAt(i));
        return rtn;
    }

    /* clause is enclosed in braces */

    public Clause parseClause (String clauseStr)
    throws BadFormatException
    {
        int start = clauseStr.indexOf(' (');
        int end = clauseStr.indexOf(') ');
        if (end <= start)
            throw new BadFormatException("error parsing term " + clauseStr);
        Clause rtn = new Clause(parseTerm(clauseStr.substring(start+1, end)));
        clauseStr = clauseStr.substring(end+1).trim();
        while (!clauseStr.equals("")) {
            start = clauseStr.indexOf(' (');
            end = clauseStr.indexOf(') ');
            if (end <= start)
                throw new BadFormatException("error parsing term " + clauseStr);
            rtn.addTerm(parseTerm (clauseStr.substring(start+1, end)));
            clauseStr = clauseStr.substring(end+1).trim();
        }
    }
}
clauseStr = clauseStr.substring(end+1).trim();
}
return rtn;


/* a term is enclosed in parens () */
public Term parseTerm (String termStr) throws BadFormatException {
StringTokenizer tokenizer = new StringTokenizer(termStr, " ");
if (tokenizer.countTokens() != 3)
    throw new BadFormatException("error parsing clause " + termStr);
String fieldName = tokenizer.nextToken();
String opStr = tokenizer.nextToken();
String valStr = tokenizer.nextToken();
int f = parseFieldName2Num (fieldName);
Operator op = parseOperator (f, opStr);
Field value = parseValue (f, op, valStr);
return new Term(f, op, value);
}

private int parseFieldName2Num (String fieldName) throws BadFormatException {
    for (int i =0; i < FlightRecord.FIELD_NAMES.length; i++)
        if (FlightRecord.FIELD_NAMES[i].equals (fieldName))
            return i;
    throw new BadFormatException("bad field name:" +fieldName);
}

private Operator parseOperator(int f, String opStr) throws BadFormatException {
    for (int i=0; i < parseTable[f].length; i++)
        if (parseTable[f][i].opStr.equals(opStr))
            return parseTable[f][i].op;
    throw new BadFormatException("invalid operator name " + opStr);
}

private Field parseValue (int f, Operator op, String valStr) throws BadFormatException {
    for (int i=0; i < parseTable[f].length; i++)
        if (parseTable[f][i].op.equals(op))
            return parseTable[f][i].fp.parse(valStr);
    throw new BadFormatException("invalid value in clause " + valStr);
}
private static class clauseRec {
    int f;
    String opStr;
    Operator op;
    FieldParser fp;

    clauseRec (int f, String opStr, Operator op, FieldParser fp) {
        this.f = f;
        this.opStr = opStr;
        this.op = op;
        this.fp = fp;
    }
}

// this table is first ordered by the FlightRecord field numbers,
// for example, FlightRecord.ID operations must be first
// because its value is 0
private static final clauseRec parseTable[][] = {
    { // FlightRecord.ID operations
        new clauseRec (FlightRecord.ID, "=", OpEqual.it, IDParser.it),
        new clauseRec (FlightRecord.ID, "isAirline", OpIsAirline.it, AirlineParser.it),
    },
    { // FlightRecord.CID operations
        new clauseRec (FlightRecord.CID, "=", OpEqual.it, CIDParser.it),
        new clauseRec (FlightRecord.SPEED, "=", OpEqual.it, SpeedParser.it),
        new clauseRec (FlightRecord.SPEED, "<", OpLess.it, SpeedParser.it),
        new clauseRec (FlightRecord.SPEED, ">", OpGreater.it, SpeedParser.it),
    },
    */
    */
};

registrar/BadRequestException.java

package registrar;

class BadRequestException extends Exception {
    public BadRequestException () { }

    public BadRequestException (String s) {
        super(s);
    }
}
D.7 util package

util/AirlineParser.java

```java
package util;
import xml.*;

public class AirlineParser implements FieldParser {
    public static final AirlineParser it = new AirlineParser();

    public Field parse (String idStr) {
        return new Airline(idStr);
    }
}
```

util/BadFormatException.java

```java
package util;

public class BadFormatException extends Exception {
    public BadFormatException () { }

    public BadFormatException (String s) {
        super(s);
    }
}
```

util/CIDParser.java

```java
package util;
import xml.*;

public class CIDParser implements FieldParser {
    public static final CIDParser it = new CIDParser();

    public Field parse (String idStr) {
        return new CID(idStr);
    }
}
```
util/EmptyQueueException.java

package util;

public class EmptyQueueException extends Exception
{
    public EmptyQueueException() { }

    public EmptyQueueException(String s) {
        super(s);
    }
}

util/FieldParser.java

package util;
import xml.*;

public interface FieldParser
{
    Field parse(String fieldStr) throws BadFormatException;
}

util/IDParser.java

package util;
import xml.*;

public class IDParser implements FieldParser
{
    public static final IDParser it = new IDParser;

    public Field parse(String idStr) {
        return new ID(idStr);
    }
}

util/PhysicalClassParser.java

package util;
import xml.*;


public class PhysicalClassParser implements FieldParser
{
    public static final PhysicalClassParser it = new PhysicalClassParser();

    public Field parse (String fieldStr) throws BadFormatException
    {
        if (fieldStr.equals(" "))
            return PhysicalClass.UNKNOWN;
        else if (fieldStr.equals("P"))
            return PhysicalClass.PISTON;
        else if (fieldStr.equals("T"))
            return PhysicalClass.TURBO;
        else if (fieldStr.equals("J"))
            return PhysicalClass.JET;
        throw new BadFormatException("unknown physical class");
    }
}

util/Queue.java

package util;

/* Dynamic Queue using an array implementation
* grows by doubling the capacity of the queue when full
* shrinks by halving the size when 3/4 of the queue is empty
* this strategy avoids “thrashing”
* Maybe a linked list implementation is better */

public class Queue
{
    Object [] list;
    int head, count, minCapacity;

    public Queue () {
        this(4);
    }

    public Queue (int minCapacity) {
        list = new Object[minCapacity];
        head = 0;
        count = 0;
    }
this.minCapacity = minCapacity;
}

public boolean empty() {
    return count == 0;
}

/* take a peek at the head of the queue
 * the object returned is the same object
 * returned by the next call to dequeue()
 * except when Queue is empty, then peek()
 * returns null, but dequeue waits for an
 * object to be appended, and only returns
 * null if waiting is interrupted
 */
public Object peek() {
    return list[(head+count) % list.length];
}

public int size() {
    return count;
}

public int capacity() {
    return list.length;
}

/* add to the tail of the queue
 * resize if necessary
 */
public synchronized void enqueue(Object item)
{
    if (count == list.length)
        resize (count * 2);

    int tail = (head+count) % list.length;
    list[tail] = item;
    count++;
    notifyAll();
}

/* remove from head of the queue
 * resize if necessary
 */
public synchronized Object dequeue()
{
    try {
        while (count == 0)
            wait();
    } catch (InterruptedException e) {
        return null;
    }

    // resize when only 1/4 of the capacity is used
    // and capacity is at least twice the minCapacity

if ((count < (list.length / 4)) && ((list.length / 2) >= minCapacity))
    resize (list.length / 2);

    Object rtn = list[head];
    list[head]=null;
    head = (head + 1) % list.length ;
    count--;  
    return rtn;
}

/* change the size of the list array
 * by creating a new array of a new size
 * requires: newsize >= count
 * modifies: list, head
 */
private void resize(int newsize) {
    System.out.println("queue size is " + newsize);
    Object [] newlist = new Object[newsize];
    /*
     *  int i,j;
     *  for (i = 0, j = head; i < count; i++) {
     *      newlist[i] = list[j];
     *      j = (j + 1) % list.length;
     *  }
     *
     *  int wraparound = (head + count) - list.length;
     *  if (wraparound > 0) {
     *      System.arraycopy (list, 0, newlist, count−wraparound, wraparound);
     *      System.arraycopy (list, head, newlist, 0, count−wraparound);
     *  }else
     *      System.arraycopy(list, head, newlist, 0, count);
     *
    
    list = newlist;
    head = 0;
    }
}

util/SpeedParser.java
package util;
import xml.*;

public class SpeedParser implements FieldParser
{
    public static final SpeedParser it = new SpeedParser();

    public Field parse (String fieldStr)
throws BadFormatException
{
    if (fieldStr.equals("SC"))
        // speed is classified
        return new Speed(Speed.T_CLASSIFIED);
    else
        if (fieldStr.startsWith("M")) {
            int speed = Integer.parseInt(fieldStr.substring(1));
            return new Speed(speed, Speed.T_MACH);
        } else {
            int type;
            if (fieldStr.length() == 4)
                type = Speed.T_TRUE_AIR;
            else
                type = Speed.T_GROUND;
            try {
                int speed = Integer.parseInt(fieldStr);
                return new Speed(speed, type);
            } catch (NumberFormatException nfe) {
                throw new BadFormatException(nfe.toString());
            }
        }
}

util/StatusParser.java

package util;
import xml.*;

public class StatusParser implements FieldParser
{
    public static final StatusParser it = new StatusParser();

    public Field parse (String fieldStr)
        throws BadFormatException
    {
        if (fieldStr.equals(" "))
            return Status.UNKNOWN;
        else if (fieldStr.equals("N"))
            return Status.NONE;
        else if (fieldStr.equals("S"))
            return Status.SCHEDULED;
        else if (fieldStr.equals("L"))
            return Status.CONTROLLED;
        else if (fieldStr.equals("F"))
            return Status.FILED;
        else if (fieldStr.equals("A"))
            return Status.ACTIVE;
else if (fieldStr.equals("R"))
    return Status.ASCENDING;
else if (fieldStr.equals("C"))
    return Status.CRUISING;
else if (fieldStr.equals("D"))
    return Status.DESCENDING;
else if (fieldStr.equals("T"))
    return Status.COMPLETED;
else if (fieldStr.equals("X"))
    return Status.CANCELLED;
else if (fieldStr.equals("M"))
    return Status.DECONTROLLED;
else if (fieldStr.equals("E"))
    return Status.ERROR;
    throw new BadFormatException("unknown user class");

util/TestQueue.java

package util;
public class TestQueue
{
    public static void main (String args[])
    {
        Queue q = new Queue();

        for (int i = 0; i < 20; i++)
            q.enqueue(new Integer(i));

        while (!q.empty())
            System.out.println (q.dequeue());
        // System.out.println (q.dequeue());

        for (int i = 0; i < q.capacity(); i++)
            q.enqueue(new Integer(i));

        int j = q.capacity() * 3;
        for (int i = 0; i < j; i++) {
            System.out.println (q.dequeue());
            q.enqueue(new Integer(i));
        }

        while (!q.empty())
            System.out.println (q.dequeue());

        System.out.println (" size = " + q.size()
                + " capacity = " + q.capacity());
package util;

import xml.*;

public class TimeParser implements FieldParser {
    // returns a time field with type = TCOORDINATION
    public static final TimeParser it = new TimeParser();

    public Field parse(String timeStr) throws BadFormatException {
        int mode = Time.M_ESTIMATED;
        int start = 1;

        if (Character.isDigit(timeStr.charAt(0))) start = 0;
        else if (timeStr.startsWith("E")) mode = Time.M_ESTIMATED;
        else if (timeStr.startsWith("A")) mode = Time.M_ACTUAL;
        else if (timeStr.startsWith("P")) mode = Time.M_PROPOSED;
        else if (timeStr.startsWith("D")) mode = Time.M_ACTUAL;
        else throw new BadFormatException("unknown mode in time");

        try {
            int hour = Integer.parseInt(timeStr.substring(start, start+2));
            int minute = Integer.parseInt(timeStr.substring(start+2));
            return new Time(hour, minute, mode, Time.TCOORDINATION);
        }
        catch (NumberFormatException nfe) {
            throw new BadFormatException(nfe.toString());
        }
    }
}
util/UserClassParser.java

package util;
import xml.*;

public class UserClassParser implements FieldParser
{
    public static final UserClassParser it = new UserClassParser();

    public Field parse (String fieldStr)
    throws BadFormatException
    {
        if (fieldStr.equals(" "))
            return UserClass.UNKNOWN;
        else if (fieldStr.equals("T"))
            return UserClass.AIR_TAXI;
        else if (fieldStr.equals("F"))
            return UserClass.CARGO;
        else if (fieldStr.equals("C"))
            return UserClass.COMMERCIAL;
        else if (fieldStr.equals("G"))
            return UserClass.GENERAL_AVIATION;
        else if (fieldStr.equals("M"))
            return UserClass.MILITARY;

        throw new BadFormatException("unknown user class");
    }
}
D.8 xml package

xml/Aircraft.java

package xml;

/**
 * represent the Aircraft Data field in NAS
 * also used in ETMS
 * see Appendix B of ASDI Doc, field 03
 */

public class Aircraft extends Field {

    boolean heavy;
    String aircraftType;
    String equipment;
    int count;

    public Aircraft(boolean heavy, String type, String equip, String count) {
        this.heavy = heavy;
        aircraftType = type;
        equipment = equip;
        this.count = Integer.parseInt(count);
    }

    public Aircraft(boolean heavy) {
        this.heavy = heavy;
    }

    public void setType(String type) {
        aircraftType = type;
    }

    public void setEquipment(String equip) {
        equipment = equip;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public void setCount(String countStr) {
        try {
            this.count = Integer.parseInt(countStr);
        } catch (Exception e) {
            this.count = 1;
        }
    }

    public boolean isHeavy () {
        return heavy;
    }
}
public boolean isType (String type) {
    return type.equals(aircraftType);
}

public int getCount() {
    return count;
}

public String toString() {
    return aircraftType + " / " + count + " / " + equipment;
}

public String toXML() {
    StringBuffer out = new StringBuffer("<aircraft heavy=");
    if (heavy) out.append("true") ";
    else out.append("false") ");
    out.append("<aircrafttype>").append(aircraftType)
    .append("</aircrafttype> ");
    out.append("<count>").append(String.valueOf(count))
    .append("</count> ");
    out.append("<equipment>").append(equipment)
    .append("</equipment></aircraft>");
    return out.toString();
}

xml/Airline.java

package xml;

/* represents an airline code, 
 * for example, USA, DAL, AA 
 * 
 * not used by ASDI, but needed in predicates 
 */

public class Airline extends Field
{
    String al;

    public Airline (String a)
    {
        al = a.trim();
    }

    public String toString() {
        return al;
public class Altitude extends Field {
   // types
   public static final int
      T_ASSIGNED = 0,
      T_REQUESTED = 1,
      T_REPORTED = 2;
   
   public static final String typeStrings[] =
      { "assigned",
         "requested",
         "reported" };
   
   // modes
   public static final int
      M_PLAIN = 0,
      M_BLOCK = 1,
      M_INTERIM = 2,
      M_MODEC = 3;
   
   public static final String modeStrings[] =
      { "plain",
         "block",
         "interim",
         "modec"};
   
   int type;
   int mode;
   int altitude;
   int altitude2;

   public Altitude(String tp, String md) {
      if (tp.equals(typeStrings[0]))
         type = T_ASSIGNED;
      else if (tp.equals(typeStrings[1]))
         type = T_REQUESTED;
      else
         type = T_REPORTED;
   
   if (md.equals(modeStrings[0]))
mode = M.PLAIN;
else if (md.equals(modeStrings[1]))
    mode = M.BLOCK;
else if (md.equals(modeStrings[2]))
    mode = M.INTERIM;
else
    mode = M.MODEC;
}

public void setType (int type) {
    this.type = type;
}

public void setMode (int mode) {
    this.mode = mode;
}

public void setValue(int altitude) {
    this.altitude = altitude;
}

public void setValue(String altStr) {
    int split = altStr.indexOf('-');
    // this is to handle OTP/* altitudes
    int split2 = altStr.indexOf('/');
    if (split != -1) {
        altitude = Integer.parseInt(altStr.substring(0, split));
        altitude2 = Integer.parseInt(altStr.substring(split + 1));
    } else if (split2 != -1) {
        altitude = Integer.parseInt(altStr.substring(split2 + 1));
    } else {
        this.altitude = Integer.parseInt(altStr);
    }
}

public boolean isType(int type) {
    return this.type==type;
}

public boolean isMode(int mode) {
    return this.mode==mode;
}

public int getValue () {
    return altitude;
}

public String toString () {
    return String.valueOf(altitude);
}

// need to be fixed to handle block altitudes
public boolean equals(Object p1) {
    if (p1 instanceof Altitude)
return ((Altitude) p1).altitude == altitude;
return false;
}

public String toXML() {
    StringBuffer out = new StringBuffer("<altitude type="" +
    typeStrings[type].append(" mode="" +
    modeStrings[mode].append("">") +
    String.valueOf(altitude));
    if (mode == M.BLOCK) out.append("\-" +
    String.valueOf(altitude2));
    out.append("</altitude>" +
    return out.toString();
}
if (isNew) return "<newcid>" + cid + "</newcid>";
else return "<cid>" + cid + "</cid>";
}

xml/Coordinate.java

package xml;

public class Coordinate extends Field {

    String latStr, lonStr;

    public Coordinate(String lat, String lon) {
        this.latStr = lat;
        this.lonStr = lon;
    }

    public Coordinate() {};

    public void setLat(String lat) {
        this.latStr = lat;
    }

    public void setLon(String lon) {
        this.lonStr = lon;
    }

    public String toString() {
        return latStr + "/" + lonStr;
    }

    public boolean equals (Object p1) {
        return false;
    }

    public boolean inside (rectangle, CoordFix) {
    }

    public boolean inside (Shape, CoordFix) {
    }

    public String toXML() {
        StringBuffer out = new StringBuffer("<coordinate><lat>");
        out.append(latStr).append("</lat><lon>");
    }
}
package xml;

public class Dest extends Fix {
    public Dest(String dest) {
        super(dest);
    }

    public String toXML() {
        return "<dest>" + fix + "</dest>";
    }
}

package xml;

import java.io.*;

// superclass for all field values in FlightRecord
public abstract class Field implements java.io.Serializable {
    //public abstract boolean equals (Field p1);
    public abstract String toXML();
}

package xml;

public class Fix extends Field {
    String fix;

    public Fix(String fix) {
        ...
this.fix = fix;
}

public void setValue(String fix) {
    this.fix = fix;
}

public boolean equals(Object p1) {
    if (p1 instanceof Fix)
        return ((Fix) p1).fix.equals (fix);
    return false;
}

public String toString () {
    return fix;
}

public String toXML() {
    return "<fix>" + fix + "</fix>";
}

xml/FlightRecord.java

package xml;

import java.io.*;
import java.util.*;

public class FlightRecord implements Serializable
{

    public static final int
    ID = 0,
    CID = 1,
    AIRCRAFT = 2,
    COORD_FIX = 3,
    COORD_TIME = 4,
    SPEED = 5,
    ALT_ASSIGNED = 6,
    ALT_REQUESTED = 7,
    ROUTE = 8,
    REMARK = 9,
    TRACK_POS = 10,
    ORIG = 11,
    DEST = 12,
    ARRIVAL_TIME = 13,
    DEPART_TIME = 14,
    STATUS = 15,
    PHYS_CLASS = 16,
public static final String FIELD_NAMES[] = {
    "ID",
    "CID",
    "AIRCRAFT",
    "COORD_FIX",
    "COORD_TIME",
    "SPEED",
    "ALT_ASSIGNED",
    "ALT_REQUESTED",
    "ROUTE",
    "REMARK",
    "TRACK_POS",
    "ORIG",
    "DEST",
    "ARRIVAL_TIME",
    "DEPART_TIME",
    "STATUS",
    "PHYS_CLASS",
    "USER_CLASS",
    "COORDINATE",
    "ALT_REPORTED",
    "ENROUTE_TIME",
    "BOUND CROSS_TIME"
};

private Field[] fields;

public FlightRecord () {
    fields = new Field[TOTAL_FIELDS];
}

public ID getID () {
    return (ID) fields[ID];
}

public Field getField (int fieldNo ) {
    return fields[fieldNo];
}

public synchronized int[] getAvailableFields () {
    int count = 0;
    for (int i = 0; i < fields.length; i++)
        if (fields[i] != null)
count++;

int rtn[] = new int [count];
int j = 0;

for (int i = 0; i < fields.length; i++)
    if (fields[i] != null) {
        rtn[j] = i;
        j++;
    }
return rtn;
}

public synchronized void setField (int fieldNo, Field newValue) {
    fields[fieldNo] = newValue;
}

public String toString() {
    String rtn = new String();
    int [] avail = getAvailableFields();
    for (int i = 0; i < avail.length; i++)
        rtn = rtn + FIELD_NAMES[avail[i]]
             + " = " + getField(avail[i]) + "\n";
    return rtn;
}

// if the flight landed, cancelled
public boolean completed() {
    if (fields[STATUS] != null)
        if (fields[STATUS].equals(Status.COMPLETED) ||
            fields[STATUS].equals(Status.CANCELLED))
            return true;
    /*
     if (fields[ARRIVAL_TIME] != null) {
         Time arrival = (Time) fields[ARRIVAL_TIME];
         if (arrival.isMode(Time.M_ACTUAL))
             return true;
     else {
         // compare with current GMT time
         // if arrival time is in the past, then completed
         Calendar time = new GregorianCalendar(TimeZone.getTimeZone("GMT"));
         time.set(Calendar.HOUR, arrival.getHours());
         time.set(Calendar.MINUTE, arrival.getMinutes());
         Date now = new Date();
         if (time.getTime().before(now))
             return true;
     else
         return false;
    }
    */
    return false;
}
merge with another flight record
// fields in newf override fields in this flight record
public synchronized void merge (FlightRecord newf) {
    for (int i = 0; i < newf.fields.length; i++)
        if (newf.fields[i] != null)
            fields[i] = newf.fields[i];
}

public String toXML(ID oldID) {
    StringBuffer out = new StringBuffer("<flight>\n    if (fields[STATUS] != null)
        out.append(" status="").append(((Status) fields[STATUS]).toString()) .append("\n");
    if (fields[PHYSCLASS] != null)
        out.append(" physclass="").append(((PhysicalClass) fields[PHYSCLASS]).toString()) .append("\n");
    if (fields[USERCLASS] != null)
        out.append(" userclass="").append(((UserClass) fields[USERCLASS]).toString()) .append("\n");
    out.append(">");
    if (oldID != null) out.append(oldID.toXML());
    int[] fs = getAvailableFields();
    for (int i = 0; i < fs.length; i++) {
        int temp = fs[i];
        if (temp != STATUS && temp != PHYSCLASS && temp != USERCLASS) {
            out.append(fields[temp].toXML());
        }
    }
    out.append("</flight>");
    return out.toString();
}

public String toXML() {
    return toXML(null);
}

xml/FlightUpdate.java

package xml;

import java.io.*;

public class FlightUpdate implements Serializable {
    ID oldID;
}
public FlightUpdate (ID id, FlightRecord fs) {
    if (id != null) oldID = id;
    else oldID = (ID) fs.getField(FlightRecord.ID);
    newStatus = fs;
}

public FlightUpdate (ID id) {
    this (id, new FlightRecord());
}

public ID getID() {
    return oldID;
}

public FlightRecord getFlightRecord() {
    return newStatus;
}

public void setField (int fieldNo, Field newValue) {
    newStatus.setField(fieldNo, newValue);
}

public Field getField (int fieldNo) {
    return newStatus.getField(fieldNo);
}

public String toString() {
    return "oldID = " + oldID + "\n" + newStatus;
}

public String toXML() {
    if (oldID.equals(newStatus.getField(FlightRecord.ID)))
        return newStatus.toXML();
    else return newStatus.toXML(oldID);
}

xml/ID.java

package xml;

/*
 * represent the Aircraft Identification field in NAS
 * also used in ETMS
 * see Appendix B of ASDI Doc, field 02
 */
public class ID extends Field {

    String id;
    boolean isNew = false;

    public ID(String id) {
        // remove blanks patted by ASDI
        this.id = id.trim();
    }

    public ID(String id, boolean isNew) {
        this.id = id.trim();
        this.isNew = isNew;
    }

    public boolean equals(Object p1) {
        if (p1 instanceof ID)
            return id.equals(((ID)p1).id);
        return false;
    }

    public boolean isAirline (Airline al) {
        return id.startsWith(al.toString();
    }

    public int hashCode() {
        return id.hashCode();
    }

    public String toString() {
        return id;
    }

    public String toXML() {
        if (isNew) return ＜newid＞" + id + "</newid> ";
        else return ＜id＞" + id + "</id>";
    }
}

xml/Origin.java

package xml;

public class Origin extends Fix {

    public Origin(String orig) {
        super(orig);
    }
}
public String toXML() {
    return "<origin>" + fix + "</origin>";
}

xml/PhysicalClass.java

package xml;

/**<p>
 * represent the AC_PHYSICAL_CLASS field in ETMS
 * see Appendix C of ASDI Doc
 */
public class PhysicalClass extends Field {

    // this works because this class is
    // immutable and fixed
    public static final PhysicalClass
        UNKNOWN = new PhysicalClass("unknown"),
        PISTON = new PhysicalClass("piston"),
        TURBO = new PhysicalClass("turbo"),
        JET = new PhysicalClass("jet");

    String physicalClass;

    public PhysicalClass(String physicalclass) {
        this.physicalClass = physicalclass;
    }

    public boolean equals(Object p1) {
        if (p1 instanceof PhysicalClass)
            return physicalClass.equals(((PhysicalClass) p1).physicalClass);
        return false;
    }

    public String toString() {
        return physicalClass;
    }

    public String toXML() { return ""; }
xml/Route.java

package xml;

import java.util.*;

public class Route extends Field {

    Vector route;

    public Route() {
        this.route = new Vector();
    }

    public void addPosition(Position pos) {
        route.addElement(pos);
    }

    public String toString() {
        StringBuffer out = new StringBuffer();
        Enumeration e = route.elements();
        while (e.hasMoreElements()) {
            out.append(((Position) e.nextElement()).toString()).append(".");
        }
        if (out.length() > 0) {
            out.setLength(out.length() - 1);
        }
        return out.toString();
    }

    public boolean equals(Object p1) {
        return false;
    }

    public String toXML() {
        StringBuffer out = new StringBuffer("<route>");
        Enumeration e = route.elements();
        while (e.hasMoreElements()) {
            out.append(((Position) e.nextElement()).toXML());
        }
        out.append("</route>");
        return out.toString();
    }
}

xml/Speed.java

package xml;
public class Speed extends Field {

    public static final int
    T_GROUND = 0,
    T_TRUE_AIR = 1,
    T_MACH = 2,
    T_CALCULATED = 3,
    T_CLASSIFIED = 4;

    public static final String[] typeStrings =
    { "ground", "trueair", "mach",
      "calculated", "classified" };

    int speed;
    int type;

    // no mach speed for ground
    public Speed(int speed, int type) {
        this.speed = speed;
        this.type = type;
    }

    public Speed(int type) {
        this.type = type;
    }

    public Speed(String typeStr) {
        type = T_GROUND;
        if (typeStr.equals("trueair")) type = T_TRUE_AIR;
        else if (typeStr.equals("mach")) type = T_MACH;
        else if (typeStr.equals("calculated")) type = T_CALCULATED;
        else if (typeStr.equals("classified")) type = T_CLASSIFIED;
    }

    public void setValue(int speed) {
        this.speed = speed;
    }

    public void setValue(String speedStr) {
        this.speed = Integer.parseInt(speedStr);
    }

    public int getValue() {
        return speed;
    }

    public boolean isType(int type) {
        return this.type == type;
    }

    public boolean isClassified() {
        return type == T_CLASSIFIED;
    }
}
public String toString() {
    return String.valueOf(speed);
}

public boolean equals(Object p1) {
    if (p1 instanceof Speed) {
        Speed s = (Speed) p1;
        return (speed == s.speed) && (type == s.type);
    }
    return false;
}

public String toXML() {
    StringBuffer out = new StringBuffer("<speed type=");
    out.append(typeStrings[type]).append("">");
    out.append(String.valueOf(speed)).append("</speed>");
    return out.toString();
}

xml/Status.java

package xml;

/*
 * represent the FLIGHT STATUS field in ETMS
 * see Appendix C of ASDI Doc
 */

public class Status extends Field {

    // this works because this class is immutable and fixed
    public static final Status
        NONE = new Status("none"),
        SCHEDULED = new Status("scheduled"),
        CONTROLLED = new Status("controlled"),
        FILED = new Status("filed"),
        ACTIVE = new Status("active"),
        ASCENDING = new Status("ascending"),
        CRUISING = new Status("cruising"),
        DESCENDING = new Status("descending"),
        COMPLETED = new Status("completed"),
        CANCELLED = new Status("cancelled"),
        DECONTROLLED = new Status("decontrolled"),
        ERROR = new Status("error"),
        UNKNOWN = new Status("unknown");

    String status;
public Status(String status) {
    this.status = status;
}

public String toString() {
    return status;
}

public boolean equals(Object p1) {
    if (p1 == this)
        return true;
    if (p1 instanceof Status)
        return status.equals(((Status)p1).status);
    return false;
}

public String toXML() { return ""; }

xml/Time.java

package xml;
import java.text.DecimalFormat;

public class Time extends Field {

    // modes
    public static final int M_ESTIMATED = 0,
    M_PROPOSED = 1,
    M_ACTUAL = 2,
    M_CONTROLLED = 3,
    M_GATE = 4;

    public static String[] modeStrings = {
        "estimated",
        "proposed",
        "actual",
        "controlled",
        "gate"};

    // types
    public static final int T_COORDINATION = 0,
    T_ARRIVAL = 1,
    T_DEPARTURE = 2,
    T_BOUNDCROSS = 3,
T_ENROUTE = 4;

public static String[] typeStrings = {
    "coordination",
    "arrival",
    "departure",
    "boundcross",
    "enroute"};

int hour;
int minute;

int mode;
int type;

public Time (int hour, int minute, int mode, int type) {
    this.hour = hour;
    this.minute = minute;
    this.mode = mode;
    this.type = type;
}

// public Time(String tm) {
//    mode defaults to coordination
//    type defaults to estimated
//    this.time = Integer.parseInt(tm);
//    }

public Time(String tp, String md) {
    // time defaults to 0
    if (tp.equals(typeStrings[0]))
        type = T_COORDINATION;
    else if (tp.equals(typeStrings[1]))
        type = T_ARRIVAL;
    else if (tp.equals(typeStrings[2]))
        type = T_DEPARTURE;
    else if (tp.equals(typeStrings[3]))
        type = T_BOUNDCROSS;
    else
        type = T_ENROUTE;

    if (md.equals(modeStrings[0]))
        mode = M_ESTIMATED;
    else if (md.equals(modeStrings[1]))
        mode = M_PROPOSED;
    else if (md.equals(modeStrings[2]))
        mode = M_ACTUAL;
    else if (md.equals(modeStrings[3]))
        mode = M_CONTROLLED;
    else
        mode = M_GATE;
}
public String toString()
{
    DecimalFormat twodigits = new DecimalFormat("00");
    return twodigits.format(hour) + twodigits.format(minute);
}

public void setType(int type)
{
    this.type = type;
}

public void setMode(int mode)
{
    this.mode = mode;
}

public void setValue(int hour, int minute)
{
    this.hour = hour;
    this.minute = minute;
}

public void setValue(String timeStr)
{
    this.hour = Integer.parseInt(timeStr.substring(0, timeStr.length() - 2));
    this.minute = Integer.parseInt(timeStr.substring(timeStr.length() - 2));
}

public boolean isType(int type)
{
    return this.type == type;
}

public boolean isMode(int mode)
{
    return this.mode == mode;
}

public int getHours()
{
    return hour;
}

public int getMinutes()
{
    return minute;
}

public boolean equals(Object p1)
{
    if (p1 instanceof Time)
    {
        Time p = (Time) p1;
        return (minute == p.minute) &&
                (hour == p.hour) && (mode == p.mode)
                && (type == p.type);
    }
    return false;
}

public String toXML()
{
    StringBuffer out = new StringBuffer("<time type="");
    out.append(typeStrings[type]).append("" mode="");
    .append(modeStrings[mode]).append(">");
}
xml/TimeInterval.java

package xml;

public class TimeInterval extends Field {

    int timeInterval;

    public TimeInterval(int timeInterval) {
        this.timeInterval = timeInterval;
    }

    public TimeInterval(String timeIntStr) {
        this.timeInterval = Integer.parseInt(timeIntStr);
    }

    public String toString() {
        return "" + timeInterval;
    }

    public boolean equals(Object p1) {
        if (p1 instanceof TimeInterval)
            return ((TimeInterval) p1).timeInterval == timeInterval;
        return false;
    }

    public String toXML() {
        return "";
    }
}

xml/UserClass.java

package xml;

/*@* represent the AC_USER_CLASS field in ETMS
* see Appendix C of ASDI Doc
*//*

public class UserClass extends Field {

    // this works because this class is

}
// immutable and fixed
public static final UserClass
    AIR_TAXI = new UserClass("airtaxi"),
    CARGO = new UserClass("cargo"),
    COMMERCIAL = new UserClass("commercial"),
    GENERAL_AVIATION = new UserClass("general"),
    MILITARY = new UserClass("military"),
    UNKNOWN = new UserClass("unknown");

String userClass;

public UserClass(String userclass) {
    this.userClass = userclass;
}

public String toString() {
    return userClass;
}

public boolean equals(Object p1) {
    if (p1 instanceof UserClass)
        return userClass.equals(((UserClass) p1).userClass);
    return false;
}

public String toXML() { return ""; }
}
Bibliography


