(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A METHODOLOGY FOR HANDLING DATA ERRORS AND INCONSISTENCIES IN DATABASE CONVERSIONS

by

Mark Robert Hendrickson

June 1990

Thesis Advisor: Vincent Y. Lum

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL *(If applicable)* Code 37
7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11 TITLE *(Include Security Classification)*

A METHODOLOGY FOR HANDLING DATA ERRORS AND INCONSISTENCIES IN DATABASE CONVERSIONS

12. PERSONAL AUTHOR(S)
Hendrickson, Mark Robert

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Master's Thesis	FROM ______ TO ______	1990 June	128

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Database conversion; integrity constraints; errors; inconsistencies; DBMS.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)* A database management system (DBMS) can have numerous errors and inconsistencies in its data. Examples of errors and inconsistencies that may be contained in a DBMS are: Referential integrity violations, logical inconsistencies, redundancies and out-of-range values. During a conversion of database management systems, the errors and inconsistencies in the source system must be corrected so the data entered into the new target DBMS will be accurate. The goal of this thesis is to examine a source database management system to determine what errors and inconsistencies are possible, to propose a methodology to detect them, and to correct such errors and inconsistencies prior to entering the data into the target DBMS. In applying my proposals, the thesis will examine the specific systems utilized by the United States Military Academy (USMA) at West Point, New York. The Academy uses a UNISYS 1100/72 mainframe computer in support of its existing network model DBMS. West Point proposes to convert from its current network model to a relational model system. The thesis will also address the general applicability of this methodology to other database management system conversions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Vincent Y. Lum	22b. TELEPHONE *(Include Area Code)* 408-646-3091	22c. OFFICE SYMBOL Code 52Lm

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

* U.S. Government Printing Office: 1986—606-24.

Unclassified

Approved for public release; distribution is unlimited.

A METHODOLOGY FOR HANDLING DATA ERRORS AND INCONSISTENCIES IN DATABASE CONVERSIONS

by

Mark Robert Hendrickson
Captain, United States Army
B.S., Middle Tennessee State University, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: [illegible]
Mark Robert Hendrickson

Approved By: [illegible]
Vincent Y. Lum, Thesis Advisor

[illegible]
C. Thomas Wu, Second Reader

[illegible]
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

A database management system (DBMS) can have numerous errors and inconsistencies in its data. Examples of errors and inconsistencies that may be contained in a DBMS are: Referential integrity violations, logical inconsistencies, redundancies and out-of-range-data values. During a conversion of database management systems, the errors and inconsistencies in the source system must be corrected so the data entered into the new target DBMS will be accurate.

The goal of this thesis is to examine a source database management system to determine what errors and inconsistencies are possible, to propose a methodology to detect them, and to correct such errors and inconsistencies prior to entering the data into the target DBMS. In applying my proposals, the thesis will examine the specific systems utilized by the United States Military Academy (USMA) at West Point, New York. The Academy uses a UNISYS 1100/72 mainframe computer in support of its existing network model DBMS. West Point proposes to convert from its current network model to a relational model system. The thesis will also address the general applicability of this methodology to other database management system conversions.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I wish to thank Dr. Vincent Y. Lum for his support and guidance as my advisor throughout the past year. His helpful assistance during my project has been immeasurable. Not only is Dr. Lum an internationally known computer scientist, but he is also a great teacher and friend.

I must also thank Mr. Bob Nelson and his DOIM-CSD staff at West Point, New York. Because of their efforts this project became reality.

Finally, and most importantly, I thank my wife, Kathy, for her patience and support during our two years at the Naval Postgraduate School. As is always the case, without her my personal achievements would not be possible.

I. INTRODUCTION

Information is a primary product of our civilization. For business and industry, information is synonymous with profitability. For service organizations and educational institutions, the method in which information is used directly influences efficiency and quality of service. Whether it concerns finance, personnel, transportation, or logistics, civilian and military organizations use database management systems to organize information and power their organizations. Data from these numerous information applications is created, observed and recorded on a daily basis. The volume of this data is generally immense; therefore, the storage, maintenance and retrieval of information is an enormous undertaking, and unmanageable if done manually.

The benefit gained by organizations using computers to manage information for various applications is enhanced by today's technology. When computers are used, the data is entered directly into a database. Multiple users can access common information simultaneously through an integrated database management system, improving the efficiency of the organization and the consistency within its information base.

The United States Military Academy (USMA) at West Point, New York is an organization that uses a database management system to manage its information. Since its founding in 1802, the charter of USMA has been to educate, train and prepare cadets to serve their country as military leaders. Maintaining information for the past 188 years, one can imagine the size to which the USMA database has grown and the importance of preserving its data correctly. The USMA database is used for, among other things, storing cadet personnel records, scheduling classes, and maintaining a field force of USMA graduates for recruiting purposes. With the advent of technology and the ultimate improvements in DBMS software, USMA decided in 1989 to convert from a network DBMS model (source system) to a relational DBMS model (target system). In general, the source system is the one presently in use by an organization while the target is the system to which the organization is converting. This thesis is not intended to explain the different existing DBMS models or to address the rationale in making such a change. Rather, it is intended to discuss error detection and correction methods as applied to the USMA database management system in the conversion process. Elmasri and Navathe [Ref. 1:pp. 133-352] provides an excellent description of existing DBMS models.

There are many problems to be faced by the USMA Database Administrator (DBA) prior to converting to a different database management system. The first is to ensure that erroneous data is not loaded from the old source system into the new target system. I contend that numerous errors and inconsistencies potentially exist in any source database management system. Some of these errors and inconsistencies are quite simple in nature; for example, an out-of-range-attribute value, like age equal to 200. Others are more complex in that they involve logical inconsistencies in their implications. In modern database management systems some of the errors can be detected by the source DBMS. For example, attribute domains can be checked as they are entered. In general, even modern database management systems do not check logical inconsistencies and integrity constraint violations. For example, if an object in one table (say, EMPLOYEE SSN) is referred to elsewhere in another table (DEPENDENT ESSN), and the particular object being referred to (John Smith 123456789) is deleted, the database management system generally does not check the validity of such an action. This action, however, will create a dangling reference, i.e., DEPENDENT ESSN no longer refers to an existing EMPLOYEE SSN with the same value. More complex logical implications are generally not checked by any database management system currently in existence. During the DBMS conversion process, errors and inconsistencies must be detected and resolved prior to entering the data into the new target system so that the data in the target system is accurate. Such error detection and correction is the focus of this thesis.

A companion thesis [Ref. 2], written by fellow Naval Postgraduate School students CPT Daniel Guilmette and CPT Georgette Wilson, centers around designing the relational database schema for the USMA target system, developing a functioning database prototype, and loading the prototype with USMA data for selected applications. Together these two theses mirror the process that any database conversion should follow.

Following the Introduction to this thesis, Chapter Two describes examples of common data errors and methods to detect and correct them. Chapter Three provides an overview of common integrity constraint violations and inconsistencies. Chapter Four discusses the specific applications of the USMA database management system. Chapter Five reviews potential errors, integrity constraint violations and inconsistencies contained in the USMA database system. Chapter Six describes a generalized methodology for error checking and correction, provides a specific method for resolving the issues at hand, and presents selected examples of the implementation. The final chapter includes conclusions and recommendations.

II. COMMON DATA ERRORS AND HOW TO DETECT AND CORRECT THEM

A. INTRODUCTION

The terms accuracy, validity and correctness, when used in DBMS contexts, relate to the integrity of the data or information within a database management system. Experience tells us that some degree of error and inconsistency invariably exists in the data of a DBMS. This is true because the data contained in any DBMS is only as accurate as the information entered and the degree of checks and control enforced by the system. Since no person or system is infallible, errors occur. Consider a bank statement or the automated personnel records maintained by employers. From time to time we find errors. As customers we ask, "How is it possible that these errors exist in what appears to be a sophisticated automated system?" Chapter Two of this thesis will explore the circumstances that cause common data errors to occur.

To ensure the accuracy of the data in a DBMS, the database manager must establish safeguards against invalid updates. Invalid updates result from data entry errors, mistakes by system operators or application programmers, system-induced errors, or security violations. Regardless of the cause of the error, the result to the consumer, and often the organization, ranges from inconvenience to significant monetary loss. Therefore, system managers are obligated to establish strategies for minimizing data errors.

In this research many examples of common data errors have been found. Common data errors are normally easy for an individual to identify, but difficult for a computer to detect. These common errors can be classified into five types: Out-of-range values, incompatible data types, subset-set discrepancies, redundancies and arithmetic mistakes.

B. CAUSES OF COMMON DATA ERRORS

Out-of-range-value errors occur in a variety of ways. The most frequent cause is unintentional typographical mistakes. Potential errors of this form could result because a manager filled out a time card incorrectly or because a secretary could not read the supervisor's writing. Such an unintentional mistake could lead to an employee's height being entered as 90 inches, when the correct value is 70 inches. Or the number of hours worked entered into the system could be 95 when the actual hours worked by the employee is only 45. In these examples, one could see how a seven could be misread as a nine, or a nine mistakenly typed instead of a four. An out-of-range-value error stays in the system when, during DBMS design, no specification is given to constrain the range of values within the data type. This allows the system to accept any entry which fits the particular data type. The value may be wrong, but the system accepts it. This explains how intentional errors are allowed to occur. In tampering with the system, an employee could alter his or her annual salary from $10,000 to $100,000 simply by entering an additional zero. While such breaches occur infrequently, they are possible as a form of out-of-range-value error. When out-of-range-value errors are entered and no constraints are specified in the system, the DBMS has no mechanism for recognizing them as mistakes. As a result, these errors reside in the system until some source outside the system identifies them for correction.

Incompatible data types may be present in the source system or may be encountered during conversion to the target system. They are caused by design specification choices made by DBMS designers in both the source and target systems, or through operational data entry errors. During the source DBMS specification, designers may decide that a valid requirement exists to store specified fields in separate files using different data types. This requirement is perfectly acceptable. For example, in one file dates are represented by an alphanumeric data type in day, month, year format (10Mar90), and in another file, the dates are represented by an integer data type in month, day, year format (031090). Each file is independent of the other with fields representing identical data. However, potential problems arise when trying to compare the consistency of redundant fields with different data types. Like comparing apples and oranges, it is difficult to check whether 10Mar90 and 031090 actually mean the same thing. Or perhaps one file will read 10Mar90 and the other 031190. The original design decision to duplicate data fields with different data types may lead to potential redundancy errors. In the previous example, one of the dates is obviously incorrect. The problem then becomes deciding which value to transform during conversion. This decision is complicated further when the target system requires a completely different data type than either of those used in the source system.

Operational data type errors occur because the data type selected during DBMS design is subject to error without the user's knowledge. In many cases, the specific data type may have been selected for a valid reason (it best meets the user's needs). For example, the DBMS designer has a valid requirement for the social security number (SSN) to be an 11 position alphanumeric field. This allows dashes to be entered as separators (123-45-6789). Since dashes are alpha characters, this data type specification enables operators to enter other alphabetic characters to the SSN field and increases the possibility of errors (ABC-DE-FGHI). Such an entry is obviously invalid, but it, or some more subtle error involving alphabetic characters, could occur (123-45-678A) because of the data type specified. Additionally, because 11 characters are specified, a SSN value could be way out of range (99999999999) again without the operator realizing a mistake had been made. If these operational data type errors are not corrected prior to conversion, they can be carried into the target system if the target system calls for identical data types.

A third type of common data error is a subset-set discrepancy. These discrepancies occur because the user decides during database design to maintain a shortened version of a longer field, as well as the longer field. In the military, for example, an individual's complete social security number is stored in the DBMS. However, in retrieving data about the service member, the individual is most often asked to provide only the last four digits of the SSN together with his or her name. The *last four*, as the military calls it, is a subset of the larger SSN set. Another example occurs in storing names in a DBMS. A full or long name, such as John David Smith, is often stored in one file with a short name, like Smith John D stored in another file. The long name would be used for formal references, as in diplomas, certificates or awards, while the short name would be used for other applications, such as the paycheck or a course roster. Again, the short name is a subset of the long name set. The problem associated with subset-set discrepancies is that similar data is stored in two places, a duplication that may lead to potential error. Errors occur when one field is changed or updated, leaving the other field in an inconsistent state, or when either the set or subset field has been incorrectly entered into the system.

Redundancy of data, which was alluded to in the previous paragraphs, means the same data is stored in two or more places. Redundancies lead to several problems: First, identical data must be entered multiple times, once for each file containing the redundancy. Second, storage space is wasted because the same information is maintained in several locations. The third and most serious problem is that files containing the same information can easily become inconsistent. This inconsistency occurs when one file is updated and another is not. For example, two files contain names and addresses for employees. In one file, a change is made to an employee's address, while the other file remains unchanged. As a result, the address of the employee is inconsistent among the files.

A fifth category of common data errors, arithmetic errors, occur when there are mistakes in the routine the system follows to compile data for a specific field. For example, when a graduate student sees that his or her graduate-level grade point average (GPA) is equal to 3.52, how does the student know whe*her this value is correct? To check its accuracy, the student could add the graduate level quality points earned and divide by the total graduate hours passed. This exercise could be accomplished with a hand-held calculator in less then ten minutes. One would expect this calculation to be a simple process for a database system. Yet, if the DBMS mistakenly adds in undergraduate-level course grades, the graduate GPA value will be in error. This error would also occur if individual grades are changed but the grade point average is not recomputed.

The area of military logistics provides a second example of the importance of maintaining accurate arithmetic fields. If the Army supply DBMS reflects that there are 10,000 tanks in the active Army, how does the logistician know whether this value is correct? Only by physically looking up each unit to determine how many tanks it has on hand. This task would be monumental given the number of Army units with tanks and the quantity of machinery assigned to each. The key to avoiding arithmetic errors in a system is to ensure that the routine used to fill in a DBMS field is accurately capturing the required data and that any change in the component data is followed by recalculation of the arithmetic.

C. HOW TO DETECT COMMON DATA ERRORS

How are the common data errors mentioned above detected? Normally, data errors are detected when someone complains, like the employee whose height is listed as 90 inches instead of 70, or the senior logistician who states he is positive the correct number of Army tanks is 12,000. It stands to reason that most errors will be identified by the person on whom the data is maintained or by the manager for a particular field or record. But prior to DBMS conversion, procedures must be established to detect errors not identified by a user or manager. Potential out-of-range-value fields can be checked for accuracy upon entry in the source DBMS through the use of restrictive data types and range constraints, if the system is so designed. For example, using restrictive data types and range constraints, the computer could recognize a height of 100 inches or a grade of *Z* as an incorrect entry. However, most systems are not set up initially to accomplish that function. In such cases these fields can be compared against a target value range. For example, a GPA must range between 0.00 and 4.33. (A grade of A+ at the Military Academy earns a numerical value of 4.33 quality points.) A comparison of the GPA field against the GPA range would be accomplished with the aid of an application program written to perform this specific function. Out-of-range-value errors would then be marked for correction.

A similar application program would be run on the fields where possible incompatible data types are found. In the source system, this program would be required to convert redundant fields with different data types so that the data could be compared for consistency. Data with different data types must be converted into a common format for comparison and checking. Also, this program would check the SSN to ensure that it contained nine integer digits with dashes in the correct positions. If required by the target DBMS, it could then remove these dashes for loading into an intermediate file. Again, errors would be marked for correction.

The subset-set discrepancy can be checked, again through an application routine, to ensure that each subset mirrors a like part of the set. For example, the *last four* subset would be checked to ensure it matches exactly the corresponding positions in the complete SSN set. Errors would be corrected prior to conversion.

Redundancies can be checked through the use of an application program to see that all redundant fields contain identical information. The application program would read each file that contained duplicate fields and compare the fields of corresponding records for accuracy. Errors would be marked for validation and correction.

Arithmetic fields would be checked to ensure the method used to calculate the field is accurate. In the GPA example, a program could be run that actually calculates the GPA based on the student's grades and then compares that value against the DBMS calculated field. This program could confirm the DBMS calculation to be correct.

D. HOW TO CORRECT COMMON DATA ERRORS

Once the common data errors have been detected and the information validated, what means exist to correct them? Manual correction is the most common method for rectifying errors. Manual correction is used when the end user complains of a mistake in the DBMS. This seems to be the most immediate and least costly way to achieve DBMS accuracy, and most of the time this may be the only way.

A second method for error correction is called the "majority rules" method. In this method, the DBMS is asked to determine whether there may be other fields that store the same information. If there are, the DBMS corrects the field with the error to be consistent with the data in the majority fields. For example, three fields contain the SSN. One SSN field is detected to have an error, so it is updated with the data from one of the majority fields. This method saves operator time but does not allow for cases where the minority field houses the correct value. This entire issue is resolved if the DBMS is designed so that only one field contains the SSN. In this case redundancies cannot occur. Guilmette and Wilson's target database design [Ref. 2] supports the redundancy issue by not allowing duplicate fields.

A final error correcting mechanism would be to build a generic knowledge-based expert system that would essentially do all the correcting work for the user. This expert system would be built on a set of facts. If it encountered an out-of-range-value error, it could analyze the problem, see how it handled the problem previously, and take corrective action. This type of system sounds quite pleasing but would be very expensive. Its drawback being that it would be designed for a specific DBMS conversion, used one time, and then discarded. Additionally, the expert system does not guarantee accurate data in all cases. For example, two files contain identical first and last names, SSN and addresses. Obviously the individuals in the records are the same person. But the names have different middle initials. The expert system would not be able to tell which record was the correct one. At best, it would have to guess.

E. CONCLUSION

The preceding paragraphs have offered several examples of common data errors that may exist in a DBMS. This is only a brief list of potential errors. There are many more. The key point is that any DBMS has a potential for errors. The database manager must implement methods to detect and correct them prior to conversion to the target system.

III. COMMON INTEGRITY CONSTRAINTS AND INCONSISTENCIES

A. INTRODUCTION

A database management system is designed to depict relationships that exist in the real world. For example, in a business application, relationships exist between employees and the departments to which they are assigned, between departments and the projects they administer, and between departments and their specific locations. In many cases, however, there are conditions that exist in the real world that cannot be stated explicitly in a relation as part of the DBMS. These conditions are known as integrity constraints. The purpose of an integrity constraint is to state the conditions among the different relations in the DBMS that are necessary due to policy, fact or logic. Integrity constraints are used in the DBMS to keep inconsistencies from occurring in the data. The importance of integrity constraints cannot be overstated. Date suggests that the specification of integrity constraints could account for as much as 80 percent of a typical DBMS description [Ref. 3:p. 36].

This chapter discusses the different types of integrity constraints available in most modern database management systems and reviews several examples of logical implications that cannot be enforced automatically by the DBMS. As was the case in Chapter Two, these integrity constraints must be checked and the data validated prior to moving the data from the source to the target DBMS. Since most database management systems do not support automatic enforcement of most integrity constraints, it becomes extremely important that data be checked for consistency before conversion.

B. INTEGRITY CONSTRAINTS

Integrity constraints serve a vital purpose within the DBMS. They provide a means of ensuring that changes made to information in the database will not result in a loss of data consistency. When the DBMS designer develops schemata for an application, one of the most important activities is to define the conditions, or integrity constraints, that must hold on the database. The designer would like to specify as many of the conditions as possible to the DBMS, and if possible, have the DBMS assume responsibility for automatically enforcing them. Problems arise, however, because there exists no automatic enforcement for most types of integrity constraints by the DBMS.

One type of integrity constraint that normally is considered part of the DBMS is called an entity integrity constraint. The idea behind the entity integrity constraint is that a primary key for a relation cannot contain a null value. This is a very important point. Primary keys perform a unique identification function between the individual objects in a relation. A primary key is a field (or attribute) whose values uniquely identfy an object (or tuple), i.e., social security number, employee number, part number, etc. A primary key value that was null would mean that there was an object that did not have a unique identification. This object would not be distinguishable from other objects; and if two objects are not distinguishable from each other, then there are not two objects but only one [Ref. 4:p. 89]. For example, in the student relation SSN is designated as the primary key and null values are allowed. Two students named Smith and Jones are part of the student relation with null values stored as their SSNs. Because Smith and Jones' primary keys are not distinguishable, a change to Smith's record would also map to Jones' record.

Let us examine entity integrity from another perspective. To allow a primary key to store null values, or any non-unique (identical) values, violates the basic definition of a relation. A relation is defined as a set of tuples. Since all elements of a set must be distinct, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes [Ref. 1:p. 141]. More importantly, for a given relation, a primary key value must uniquely and functionally determine all the other attribute values in the relation for this tuple. This dependence, called functional dependence, forms the theoretical foundation of relations and cannot be violated. If the primary key (e.g., SSN) can contain null values, tv'o distinct tuples, say John Brown and George Johnson, will mean that a given primary key, namely null values, map to both John Brown and George Johnson. Such existence totally violates the functional dependency concept and cannot be allowed.

A second type of integrity constraint that normally is not part of the DBMS is known as a referential integrity constraint. This constraint is specified between two relations, whereas entity integrity constraints are specified on an individual relation. Referential integrity constraints are used to maintain consistency among the tuples of the relation. In general, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. A more formal definition of referential integrity is provided by Date:

> Let R_1 be a relation with an attribute A that is defined on a primary domain D. At any given time, each value of A in R_1 must be either null or equal to V, where V is the primary key value of some tuple in another relation R_2 (R_1 and R_2 are not necessarily distinct) with primary key defined on D [Ref. 4:p. 89].

An example of referential integrity constraints is displayed in Figure 1. Referential integrity constraints can be displayed by drawing an arc from a relation tc the relation it references (or refers to).

In database management systems there can be many referential integrity constraints. To specify these constraints, database designers must have a thorough understanding of the meaning that each attribute plays in the different schemata of the database.

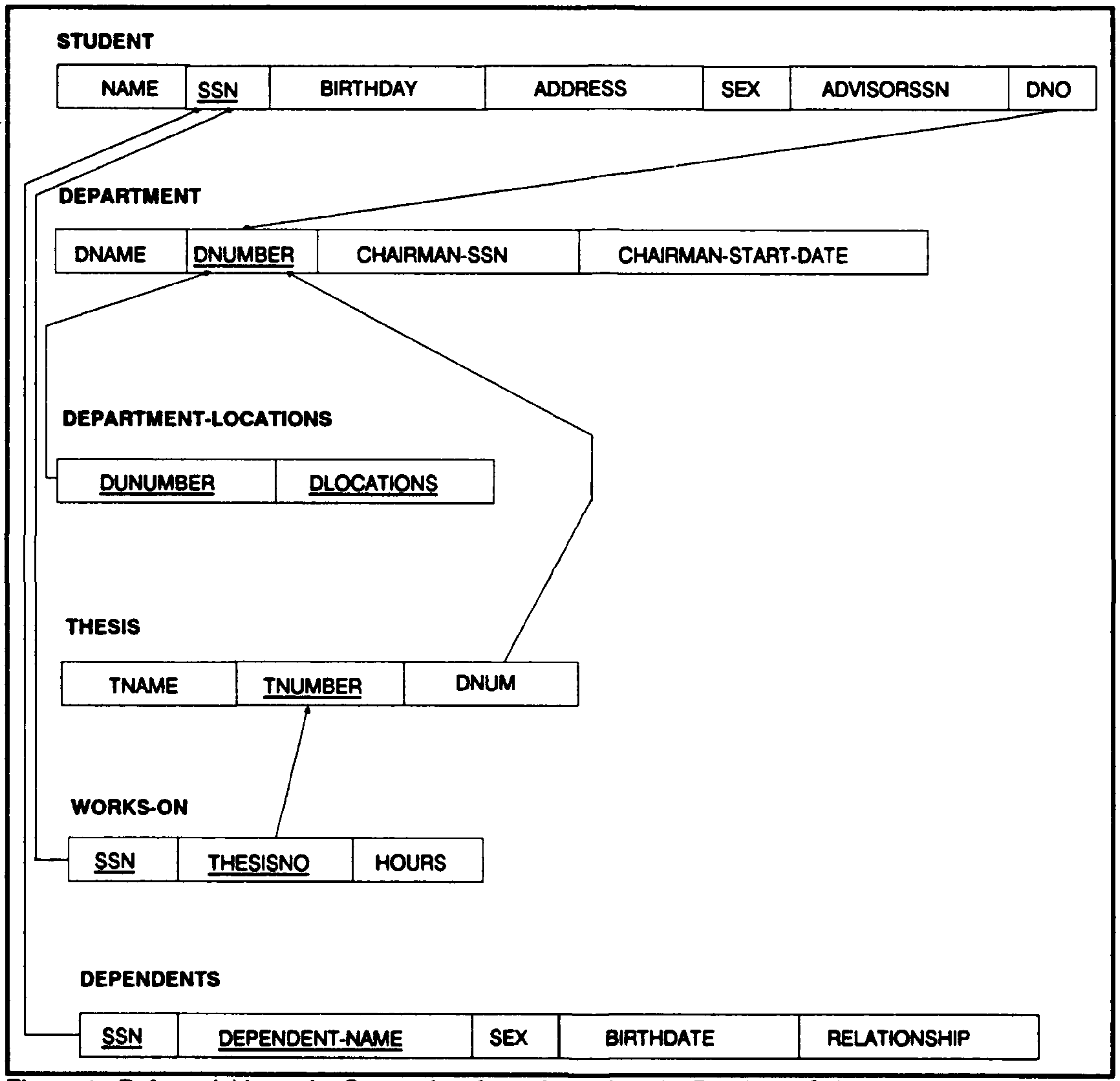

Figure 1. Referential Integrity Constraints from the University Database Schema.

Let us look at some examples of referential integrity. Consider the university database of Figure 1. In the STUDENT relation, the attribute DNO references the

DEPARTMENT where a student is assigned (like the Computer Science Department). This means the value of DNO in any tuple t_i of the STUDENT relation must match a value of the primary key of the DEPARTMENT relation, the DNUMBER attribute, in some tuple t_j of the DEPARTMENT relation. Or the value of DNO can be null if the student does not yet belong to an academic department. If tuple t_1 of the STUDENT relation has a DNO attribute value equal to 368, one would expect to see a tuple in the DEPARTMENT relation with a DNUMBER value equal to 368. In this example, the referential integrity constraint would hold.

The reader may ask "What is the significance of referential integrity constraints?" Consider a second example, again using Figure 1. In the DEPENDENTS relation, the attribute SSN refers to the STUDENT SSN that the DEPENDENT is a dependent of. Let's say that Kathy O'Keefe (with SSN equal to 123456789) was a student at the university. Her personal information would be contained in the STUDENT relation of the university database. Let's also assume that Kathy has two dependents named Mike and Katelyn. These two dependents of Kathy's would be part of the DEPENDENTS relation. Kathy's SSN would be stored in the DEPENDENTS relation as a reference to her SSN in the STUDENT relation. If Kathy were to leave the university and her tuple is deleted in the STUDENT relation, but her dependents were still part of the database, there would be no way to determine whose dependents they were. This results in a dangling reference because DEPENDENTS SSN no longer references an existing STUDENT SSN with the same value. The same sort of problem occurs in the previous example if the department with a DNUMBER equal to 368 were to disband, but the t_1 tuple of the STUDENT relation still refers to department 368.

Regardless of whether entity integrity and referential integrity constraints are provided for by the DBMS, it is imperative to ensure that, prior to DBMS conversion, both of these constraints are checked for consistency, validation as required, and correction before moving information from the source to the target DBMS.

C. LOGICAL INCONSISTENCIES

The integrity constraints listed above exclude a much larger class of constraints, known as logical or semantic integrity constraints. Examples of several simple, logical integrity constraints (simple because they require accessing and checking only one relation or file) are listed below:

1. Every graduate student must have a bachelors degree.
2. A student's age is non-decreasing.
3. The average annual salary of professors in the Computer Science Department is between $40,000 and $45,000.
4. The number of female instructors is non-decreasing.
5. No student may graduate if his/her cumulative GPA is below 3.00.

More complex logical inconsistencies would require the access and checking of two or more relations or files. Examples of complex logical integrity constraints that require the checking of values in several relations are:

1. An employee's quarterly bonus is based on whether the employee meets his personal quarterly sales quota, whether his particular department shows a profit for the quarter, and if the corporation as a whole shows a quarterly profit. If these conditions are met, then the present bonus can be no less than the bonus from the corresponding quarter of the previous year.
2. A professor's gross annual earnings are based on a percentage of the monetary value of research grants he receives for the year and whether this value is more than the previous year, plus his base salary. For an energetic professor, these research grants can be quite large. However, no professor may earn more than the superintendent of the university.

This type of constraint should be specified during DBMS design and enforced upon implementation. Unfortunately, there are few systems that automatically support and enforce these logical integrity constraints. According to Elmasri and Navathe, "support and enforcement of integrity constraints in general is a weak point of many existing database management systems." [Ref. 1:p. 597] A select few systems may provide some means of logical constraint enforcement through procedurally coding the constraints, assertions and triggers.

Constraint coding can be accomplished in an efficient manner, but this technique places a great burden on the programmer who must understand all the constraints a transaction may violate. A tiny error or omission can lead to an inconsistency. Assertions and triggers are appealing to users and programmers because of their simplicity and flexibility. Unfortunately, both have proved to be difficult to implement due to inefficient and complex integrity control subsystems. A more complete review of constraint coding, assertions and triggers can be found in [Ref. 1:pp. 599-602].

It should be apparent from the above discussion that due to the difficulty in enforcing logical inconsistencies, an effort must be made prior to DBMS conversion to first identify the logical integrity constraints that must hold on the system, and then to develop application programs to support checking the potential logical inconsistencies. As was the case with common data errors, logical inconsistencies can be marked for validation and correction.

The task to identify the logical integrity constraints is in itself a complicated and arduous undertaking. In any organization there may be a myriad of regulations and rules of operation that are related to these constraints. In many cases, an organization's rules are not explicitly stated; in fact, many times they are unwritten. The mission of specifying the constraints the organization wishes to hold is a significant task, and identifying the relations affected by the constraints is an even larger one.

D. CONCLUSION

The paragraphs above have reviewed the ideas of integrity constraints and possible logical inconsistencies that may occur in the DBMS. Significant numbers of potential errors exist in this area. It is imperative, then, that the database manager review the integrity constraints and logical implications that must hold on the database and take steps to check them prior to moving the data to the target system.

IV. SPECIFIC APPLICATIONS OF THE WEST POINT SYSTEM

A. INTRODUCTION

The charter of the United States Military Academy is to prepare young men and women to serve their country in uniform. Prior to acceptance into West Point, and during the cadet's four years there, USMA maintains a multitude of data on every student. The West Point database that maintains this information is called the Cadet Information Database (CIDB). The purpose of this chapter is to ,resent selected specific applications from the CIDB. This information will establish the foundations for Chapter Five's discussion of potential errors and inconsistencies contained in the data from the CIDB. A complete application of the West Point system can be found in Guilmette and Wilson's companion thesis [Ref. 2]. The task to develop the complete USMA application was a joint effort (Hendrickson, Guilmette and Wilson). This was a time and labor intensive endeavor given the amount of preliminary data gathering, research and analysis requisite to developing the application. We estimate that at a minimum, three months were spent trying to gain a thorough understanding of the CIDB using the documents provided to us. This task was made especially challenging by the absence of a USMA organizational manual to describe the functions of the DBMS users, data dictionaries with cryptic, confusing or nonexistent comments, data fields that were named in such a way that it was not obvious what the data field represented, and in general, conflicting and inconsistent information contained in the USMA reference documents.

B. BACKGROUND

The Computer Systems Division (CSD) of the Directorate of Information Management (DOIM) is responsible for all USMA data processing. Included in this broad mission statement is the responsibility to operate and maintain the CIDB. Three organizations at USMA play a major role in updating the CIDB and are also proponents for their specific areas of the CIDB. These three organizations are: The Director of Admissions (DAD), responsible for recruiting future cadets to USMA; The Office of the Dean (Dean), responsible for the normal registrar duties, like scheduling courses, posting grades, and issuing transcripts; and the Commandant, United States Corps of Cadets (USCC), responsible for military-related information, such as personal data, military training received, leadership scores, athletic and physical abilities, and disciplinary records. The Computer Systems Division works in a direct support role with all three proponents to ensure that the CIDB is maintained properly.

The recruiting mission of USMA and the Director of Admissions involves a lengthy process that begins a year and a half before the class start date. In February of each year DAD mails inquiries to approximately 60,000 potential applicants. From April through March, application packets are returned to West Point. The number of application packets is normally near 14,000. In December, a USMA admissions committee meets to rank order the applicant packages received to date. In January and February approximately 6,000 nominations are received from congress. Of the 6,000 applicants nominated, USMA selects approximately 1,800 as qualified applicants and sends them offers to attend West Point. Normally around 1,300 of the qualified applicants accept the offer and actually arrive at USMA in June for the first day of training. In late May, over 100 of the data fields from the DAD's portion of the CIDB are copied to fields maintained by the Dean and USCC. The data copied is the cadet candidate information that relates to the 1,300 cadets expected to report for the first day of school. At any given time there are approximately 4,400 cadets attending USMA with records maintained in the CIDB. Let us examine some of the specific responsibilities and applications of the DAD, Dean and USCC.

C. DIRECTOR OF ADMISSIONS

The Director of Admissions is responsible for recruiting, testing and appointing applicants to USMA. As application packets are received from potential cadets, the process of entering applicant data into the CIDB begins.

Once the data has been entered, the applicant is known as a cadet candidate. In addition to maintaining information on each cadet candidate, the DAD maintains data on its recruiters, also known as the field force. The field force is composed of two groups of individuals, the admission participants and the educators. The Director of Admissions also keeps information about high schools, as the high school is the primary source for recruiting cadets to attend USMA. Data on physical aptitude examination (PAE) test sites is also kept by the DAD. Finally, the DAD maintains information on the individuals (senators and congressmen) who have nomination authority of USMA applicants.

The information entered into the CIDB on a candidate includes the social security number, name, address, telephone number, sex, height, weight, race and birthdate. The projected USMA graduation year and several test scores are also maintained (ACT, SAT and PAE). High school information about the candidate, to include the school's Princeton identification number is maintained. The CIDB also stores the admission participant's identifier for the candidate as well as the test site identifier of where the candidate took the PAE. Finally, NCAA athletic information is maintained if the candidate has the desire and ability to participate in intercollegiate sports.

Admission Participants (AP) or liaison officers, make up the first group of the field force. An AP is a graduate of USMA and is retired or in a reserve status. The AP's job is to recruit qualified young men and women to attend USMA. The data to be entered on the AP includes a unique AP identifier, name, SSN, address, telephone number, rank, branch of service, USMA graduation year, and the month and year the AP joined the field force program.

Educators constitute the second group of the field force. Normally, educators are instructors or guidance counselors from junior and senior high schools, but other interested individuals may also participate as educators in the USMA recruitment process (for example, local news media). Educator information consists of a SSN, name, address, month and year joining the field force, the AP identifier of the admission participant who is responsible for working with the educator, and the test site identifier to which the educator's applicants will be assigned for PAE testing.

The Director of Admissions uses the high school as one of its primary means of recruiting. The USMA mails information packets, catalogs and other promotional materials to high schools in an effort to recruit quality personnel. The data maintained on high schools includes the Princeton identifying number, school name, address, AP identifier for the field force representative who handles the admissions interests at the school, and the test site identifier that applicants from that school would be assigned when taking the PAE.

Applicants must take the PAE before submitting an application packet to USMA. Locations where the PAE is given are called test sites. Normally, armories, gymnasiums, or high schools are used as test sites. The information entered to the CIDB about test sites includes the unique test site identifier, the test site name, name of the individual responsible for the site, address of the site, its capacity, telephone number, and dates and times of the PAE.

The Director of Admissions also maintains data on those individuals who have the authority to nominate applicants to attend West Point. The nomination of an applicant normally comes from a senator or congressman. The information stored on these individuals (nominating authority) includes a unique identifying number, title, name, address, telephone number, and data concerning the number of nominations authorized, filled and vacant.

Finally, the Director of Admissions keeps a record of AP identifiers and test site identifiers that coincide with a particular zip code. This listing makes a convenient cross reference to determine the AP and test site for each zip code.

D. OFFICE OF THE DEAN

The major responsibilities of the Office of the Dean are to oversee the academic education received by cadets, to schedule classes, and to maintain cadet grades. The Dean keeps information on courses to be taught, classrooms available for each course (for example, ensuring a chemistry lab is not scheduled in an English classroom), books to be used for each class, course schedules for students, and grades received.

All USMA cadets take the same courses during their first two years at West Point. These include core courses such as math, English, and chemistry. Many cadets also have identical course schedules during their last two years. Prior to the end of the sophomore year, cadets must select a field of study (similar to a major). At that time they must forecast the remaining courses they wish to take and when they wish to take them during the last two years. The cadet's graduation year determines the total number of classes the cadet must take. Currently, 40 academic and eight physical education courses are required for graduation. The 40 academic courses are broken into 31 core courses and nine field of study courses.

To facilitate course enrollment, the following data is maintained on each course USMA offers: Course name and number, the year and terms the course is offered, credit hours for the course, labs (if required), number of students enrolled, and any prerequisites for the course. Information concerning the required texts is also stored in the CIDB. This data includes a unique identifying number and issue code for each book, title, author, price, number of books on hand, number of books ordered, and an estimated delivery date. Finally, classroom information is maintained to assist the scheduling process. This data includes building name, room number, room capacity, classroom type (lecture or lab), and the department usually associated with the classroom.

The final course grades received by each cadet are entered to the CIDB by the instructors at the end of the term. In addition to course grades, the Dean maintains quality points, credit hours and QPA for each cadet on a term, yearly and cumulative basis. At West Point the grade point average is called QPA for quality point average. The CIDB stores a distinguished cadet indicator as well as a probation flag to indicate superior or problematic performance, as appropriate.

E. COMMANDANT, UNITED STATES CORPS OF CADETS

The United States Corps of Cadets (USCC) has primary responsibility for the military training received by cadets, as well as leadership, discipline, physical, and athletic training. In addition to many of the CIDB fields copied from the DAD prior to enrollment (like name, SSN, sex and blood type), USCC maintains individual information, such as a specific cadet number (composed of the graduation year and five digit alpha number), as well as current height and weight, permanent company and regiment, and physical fitness test results. Many personal data fields are stored, such as cadet long name (up to 60 characters in length), and parents' rank/title, name, address and phone number.

The United States Corps of Cadets stores prior college, prior service and graduation information in the CIDB. Prior college information is maintained on all colleges attended and includes the name and address of the college and the number of months in attendance there. Prior service data includes whether the cadet attended a prep school, the prior service component, number of months of service, and military occupational specialty. The graduation information is entered during the last term prior to graduation and includes graduation date, commissioning date, basic and detail branches, and Graduate Record Exam scores.

The United States Corps of Cadets is responsible for maintaining cadet illness and injury records. Data maintained includes the time a cadet went on sick call, the time returned, and the date and disposition of the illness. Injury information includes the date of the injury, activity the cadet was participating in when injured, and the nature of the injury.

Leadership development is vital to a cadet's success upon graduation. Leadership records are maintained to document leadership positions held, ratings received in those positions, and summer assignments that reflect where the cadet served and the dates of the summer assignment. Summer assignments may include regular Army-type training like Ranger, Airborne, and Air Assault Schools, as well as Cadet Troop Leadership Training (CTLT), where cadets go to active Army units to train and lead platoons for six weeks.

Disciplinary records are also stored and reflect the number of demerits and disciplinary actions taken against the cadet as punishment. Demerit data is maintained on a daily, monthly and yearly basis and includes an offense code and number to delineate between occurrences of an infraction.

A final area of responsibility for USCC is to track the athletic and extra-curricular activities in which cadets may participate (for example, varsity football, intramural basketball, debate team, glee club, etc.). The activity start date, number of trips taken while a member of the activity, number of days in the activity and the type of award received are data fields for which USCC has oversight. Additionally, extra-curricular trip information is charted by cadet on a weekly basis and includes an identifying code for each trip taken. Other information stored is the city, state, and zip code of the location of the trip and the individuals in charge of the trip.

F. CONCLUSION

The preceding paragraphs are intended to provide an overview of the applications of the Cadet Information Database. The discussions above are purposely not all inclusive, but rather are meant to provide the reader with a clear and concise picture of the types of information maintained in the CIDB and an understanding of which office is responsible for maintaining the data. Although certain parts of the database may be difficult to understand and some attributes quite cryptic, the West Point database management system is very similar to the DBMS of any civilian university. Social security numbers, names, addresses, classes taken, grades, and quality point averages are just a few of the similarities between the CIDB and a university DBMS.

This chapter was developed from USMA Regulation 25-5 [Ref. 5] and the USMA Cadet Information Database Dictionary [Ref. 6].

V. POTENTIAL ERRORS, INTEGRITY CONSTRAINT VIOLATIONS AND INCONSISTENCIES IN THE WEST POINT SYSTEM

A. INTRODUCTION

Chapters Two and Three of this thesis provided examples of potential errors and inconsistencies that may be found in any database management system. The purpose of this chapter is twofold: First, and most importantly, to outline specific examples where possible errors, integrity constraint violations and inconsistencies can occur in the present USMA system. This listing is not all-inclusive, but is intended to be comprehensive enough to assist the DOIM CSD staff in its data validation efforts prior to system conversion. Painstaking work of a tedious nature was involved in formulating the listing that follows. The process required delving into a data dictionary in excess of 400 pages, that in many cases was not well commented, searching for and documenting all the instances where a particular field contained a potential out-of-range value (like SSN), and then starting over and looking for all the instances of the next out-of-range field (like height and weight). This procedure was repeated numerous times in search of as many potential errors as possible. The process of dissecting the data dictionary and compiling lists of potential errors is analogous to that of a detective hunting for clues to solve a mystery. Secondly, this chapter is intended to provide the backdrop for proposing a methodology to solve the issues at hand. The examples described will be accompanied, when applicable, by figures to illustrate the potential problem. Where appropriate, field names will include the unique three character USMA identifier for reference. The Military Academy uses these three character identifiers to designate field names or attributes. For example, the cadet candidate social security number field is identified by the code AAB, and the cadet SSN field is identified by the code CAA. See Appendix A for a glossary of USMA field identifier codes not defined in the text that potentially contain errors and inconsistencies.

B. COMMON DATA ERRORS

1. Out-of-Range Values

a. Throughout the entire database, with the exception of the Educator-ID field (FPA), social security numbers have the potential to contain out-of-range values (99999999999).

b. In the cadet candidate portion of the database, the following fields may store out-of-range values:

(1) Several Julian dates are used that allow the numerical day entry to reach 999, though the maximum Julian date is 366.

(2) The height and weight fields allow entries as low as 00 inches and 000 pounds or as high as 99 inches and 999 pounds, though the lower and upper extremes represent improbable values.

(3) The birth month and day fields accept entries up to 99, but the maximum month is 12 and the maximum day is 31.

(4) Fields for physical activity exam and admission scores should range between 200 and 800, but the system allows entries as low as 000 or as high as 999.

(5) The system accepts ACT scores ranging from 00 to 99 and SAT scores from 000 to 999, though actual ACT scores must range between 1 and 36 and SAT scores from 200 to 800.

(6) The value for years of work experience during high school, which normally should not exceed four, may equal nine.

(7) Single character fields such as sex, race, ethnic background, and the flags indicating whether certain USMA application forms have been received, could possibly contain values from A to Z, though M or F, or Y or N are the appropriate values.

(8) The entry for month joined the field force could equal up to 99, while the actual maximum value is 12.

(9) The system allows values for nomination vacancies allowed, filled, and authorized for the current academic year to equal up to 999, though no nominating authority could possibly have this many vacancies.

(10) The nomination record contains a nomination selection score that could reach 9999, but a score this high is not possible.

(11) The test site record uses a nine character test date that allows day, month and year to be out of range (99XXX9999), and a four digit test time that would allow a time of 9999 to be entered to the system, when the maximum allowable time should be 2359.

c. In the scheduling portion of the CIDB, the following fields could contain out-of-range values:

(1) Julian dates allow the numerical day entry to reach 999, while the maximum Julian date is 366.

(2) Book quantity on hand and requested quantity can both equal 99999, but with 4,400 cadets, it is unlikely USMA would maintain such quantities of a single book.

(3) The system accepts a classroom capacity entry of 999, but no USMA classroom accommodates this number.

(4) By system constraints, the course population could equal 9999, but the actual maximum could not exceed the cadet enrollment.

(5) Course enrollment reflects minimum, maximum and desired enrollment per course. System entry could be 999, but no class is this large.

(6) There are numerous flags and single character fields throughout this application that contain a single character designator, like Y or N; however, characters from A to Z are accepted.

d. In the cadet record portion of the CIDB out-of-range values could occur in the following fields:

(1) Height and weight fields can store values as low as 00 inches and 000 pounds or as high as 99 inches and 999 pounds. Both lower and upper extremes represent improbable values.

(2) The Army physical fitness test maximum score is 300, but the field can reach 999.

(3) The entrance class size, graduation class size, class size at start of term and class size at end of term may all, by system standards, be as large as 9999, but no class will ever be this large.

(4) Graduate Record Exam scores may be stored up to 999, but a value this high is not possible.

(5) While the system accepts an input of 99 minutes, 99 seconds, the entry for the 1.5 mile entrance run should not exceed 59 minutes, 59 seconds.

(6) The trip departure and return times cannot be greater than 2359, but the system allows 9999.

(7) The system accepts a cadet illness date and excused to date of 999999, though if such an entry were used, both the month and day would be out of range. Illness time in and time out could be 9999, when the maximum should be 2359.

(8) Cadet days participating in an activity should not exceed 366, but entries up to 999 are accepted.

(9) Several orders of merit (OM) are found in the grades section of a cadet's record. These OM entries are allowed to be as large as 9999, though a class size could never be this large.

(10) Quality point averages should be a maximum of 4.330, but are accepted by the system up to 9.999.

2. Incompatible Data Types

a. As was the case for the out-of-range values described above, all social security numbers stored in the CIDB may contain errors due to the data type selected. Since the SSN data type is alphanumeric, both characters and integers are acceptable entries (ABC-45-FGHI or 123-ZZ-6789).

b. Following are two examples of redundant fields using different data types. From the individual record of cadet candidate, height-of-individual and weight-of-individual are declared as two and three character alphanumeric fields respectively, while in the entrance and high school record, the entrance-height and entrance-weight are declared as two and three digit integer values. Also from the individual record, the transcript-grad-year is declared as a two character alphanumeric field. In the cadet class record, the class-graduation-year is declared as a two digit integer value. By selecting an alphanumeric data type, the cadet candidate height and weight can have incorrect values like height equal to 7C inches and weight of 18E, while the entrance-height is equal to 72 inches and entrance-weight is 184. The problem with redundant data fields is that it may be difficult to tell which field holds the correct value. Obviously a height of 7C inches is incorrect, but there is no way to know if entrance-height equal to 72 inches is accurate.

c. From the cadet candidate area of the CIDB, the following fields may contain errors in the data because of incompatible data types:

(1) Several Julian dates are declared as four character alphanumeric fields allowing data like ABCD to be entered to the system.

(2) Zip codes and telephone numbers are typed as nine and ten alphanumeric characters respectively. This enables operators to enter information such as 93943ABCD for the zip code and 408647ABCD for the phone number.

(3) Height and weight fields allow two or three alphanumeric characters respectively. The height and weight values could therefore be entered as GB inches and AHE pounds, for example.

(4) The transcript graduation year is declared as a two character alphanumeric field. One would expect to see values like 89 or 90, but HI or IO is possible.

(5) The entrance senator or district number field is declared as two alphanumeric characters, but values are always integers.

(6) The source sequence number field is declared as a single alphanumeric character, but it must contain an integer value.

(7) The percent onto college field is declared as three alphanumeric characters. However, it should be stored as an integer, otherwise values for a percentage could look like ABC or 9B2.

(8) The month and year joining the field force, USMA class year, and training year fields are typed as two alphanumeric characters, but only integers should be allowed.

d. From the scheduling portion of the database, the following fields may store erroneous information due to the data type used:

(1) Telephone numbers are typed as ten alphanumeric characters. This declaration allows entries like 615DEB5373.

(2) The master course number field always contains a three digit integer value, but it is declared as three alphanumeric characters.

(3) The permanent regiment is declared as a single alphanumeric character but can only hold integer values from one to four.

e. From the cadet record area of the CIDB, the fields listed below may contain inconsistent data because of the data type declared:

(1) Zip codes and telephone numbers are declared as nine and ten character alphanumeric fields. Therefore, entries like 9C9D3E000 for the zip code and ABC384EFGH for the telephone number are possible.

(2) Course number is declared as storing three alphanumeric characters. This allows values such as ABC or D3Z to be stored when three integers like 100 or 101 are expected.

3. Subset-Set Discrepancies

In the entire CIDB, only three examples of subset-set discrepancies can be found. The first two examples come from fields within the database. In the individual record of the cadet candidate, the preferred-name-individual (ACC) is a ten character field consisting of as many letters of the full name as possible, starting with the last name. It is a subset of the 27 character name-individual (ACB) set that is in last, first and middle initial sequence. In the cadet record, cadet-short-name (CBE) is a 27 character field that contains as many letters of the name as possible, again in last, first and middle initial sequence. Cadet-short-name is a subset of the 60 character cadet-long-name (CIF) which comes from the cadet personal data record. The cadet-long-name is in first, middle and last name sequence. A third and more subtle example of this subset-set discrepancy that must be checked comes from the fact that prior to the class start date, the DOIM CSD staff copies much of the information from the cadet candidate individual record (AAA) into the cadet record (CAA), the entrance and high school record (CEA), and the cadet personal data record (CHA) for the 1,300 cadet candidates that USMA expects to begin school. Once school starts, the 1,300 cadet records which constitute the subset, but may include late arrivals, should be compared against the larger set of 14,000 cadet candidate individual records to ensure that every cadet record comes from the larger set of cadet candidate individual records.

4. Redundancies

Duplication of data fields occurs frequently in the CIDB. As mentioned in Chapter Four and in the preceding paragraph, the duplication of data fields starts when many of the cadet candidate fields are copied into the cadet and schedule portions of the database. A few examples of fields that are copied are SSN, name, sex, height, weight, ethnic background, race and birthday. The problems associated with redundant data fields are twofold. First, if the information is accurate in the first file, it should be accurate when applied to the second file. If an update is made to the second file and not the first, the two files become inconsistent. Second, if the first file contained erroneous information and was copied to the second file, the data in the second file would also be incorrect. If the second file was updated to correct the inaccuracy, the two files would again be inconsistent and the user would not know which, if either, was correct. These problems are further exacerbated if more than two files contain identical data. Prior to system conversion, redundant data fields must be checked for accuracy and corrections must be made to inconsistent fields so the information moved to the target DBMS will be clean.

To further illustrate the duplication of data in the CIDB, Figures 2 and 3 are provided. Figure 2 displays the commonality between the CADET RECORD and the SCHEDULE CADET RECORD. Only the duplicate fields from these two records are shown. Over 50 percent of the fields from the two records are redundant. Figure 3 shows the similarity between the CADET VALIDATION RECORD and the SCHEDULE CADET RECORD. Every field from these two records is duplicated.

CADET RECORD	SCHEDULE CADET RECORD
CADET-SSAN	SCHED-CADET-SSAN
CADET-GRAD-YEAR	SCHED-CADET-NAME
CADET-SHORT-NAME	SCHED-CADET-GRAD-YEAR
CADET-SEX-FLAG	SCHED-CADET-SEX-CODE
CADET-SEPARATION-FLAG	SCHED-CADET-PERM-COMPANY
CADET-TURN-COME-BACK-FLAG	SCHED-CADET-PERM-REGIMENT
CADET-DEFERRED-TURN-BK-FLAG	SCHED-CADET-FIELD-OF-STUDY
CADET-PERM-COMPANY	SCHED-CADET-2ND-FIELD-OF-STUDY
CADET-PERM-REGIMENT	SCHED-CADET-PREREQ-CHECK-FLAG
CADET-FIELD-OF-STUDY	SCHED-CADET-GRAD-CHECK-FLAG
CADET-SECOND-FIELD-OF-STUDY	SCHED-CADET-FOS-CHECK-FLAG
CADET-CRSE-PREREQUISITE-CHECK	SCHED-CADET-TURN-COME-BACK-FLAG
CADET-CRSE-GRADUATION-CHECK	SCHED-CADET-DEF-TURN-BACK-FLAG
CADET-FIELD-OF-STUDY-CHECK	SCHED-CADET-SEPARATION-FLAG

Figure 2. Redundancies Between Cadet Record and Schedule Cadet Record

Fields in both figures are displayed vertically to assist in identifying the redundances. While potentially leaving the system in an inconsistent state, redundancies also require multiple entries (one for each record containing the duplicate data), and waste valuable system storage space.

CADET VALIDATION RECORD	SCHEDULE CADET VALIDATION RECORD
CADET-VALIDATION-COURSE-DESC	SCHED-CADET-VALID-CRSE-DESC
CADET-VALIDATION-COURSE-YEAR	SCHED-CADET-VALID-CRSE-YEAR
CADET-VALIDATION-COURSE-TERM	SCHED-CADET-VALID-CRSE-TERM
CADET-VALIDATION-COURSE-TYPE	SCHED-CADET-VALID-CRSE-TYPE

Figure 3. Redundancies Between Cadet Validation Record and Schedule Validation Cadet Record

5. Arithmetic Errors

Throughout the CIDB there are data fields that are mathematically manipulated to provide a result for another particular field. It is possible that the computed result stored in that field is inaccurate. Data fields that are computed must be recalculated and validated prior to data transfer. Examples of computed fields from the CIDB that could lead to arithmetic errors are:

a. The daily cadet-demerits-awarded (TTF) to a cadet are added together to input to a monthly total. In turn, the monthly-demerits-received (TPF) are totaled to input to a yearly-demerits-received (TVH) total. In a similar fashion, the cadet-demerits-area-tours-awarded (TTG) and cadet-demerits-room-tours-awarded (TTH) are added together to determine a monthly-special-penalty-tour (TPG) total. This monthly disciplinary tour total is then added to the yearly-special-penalty-tour (TVI) total to provide a yearly disciplinary tour total.

b. In the cadet academic grades record there are several data fields used to determine information for the cadet academic year and term record. For example, for each course a cadet takes in a term, the grades-course-credit-hours (DLB) and the grades-course-letter-grade (DLL) (which is converted to a numerical value), are multiplied together to provide the quality points for that course. All of the course quality point values for the term are added together to provide a term-academic-quality-point (DGF) value. Credit hours for the term are also added together to provide the term-academic-credit-hour (DGG) value. The term-academic-credit-hour value is then divided into the term-academic-quality-point value to provide the term-academic-quality-point-average (DGH). Similar calculations are made to determine the year-summary-academic-quality-points (DHF), year-summary-academic-credit-hours (DHG), year-summary-academic-QPA (DHH) values, cumulative-academic-quality-points (DIH), cumulative-academic-credit-hours (DIF), and the cumulative-academic-QPA (DII) values. Several in-depth calculations are used to determine the term, year and cumulative orders of merit, and the term, year and cumulative academic percentiles.

C. INTEGRITY CONSTRAINT VIOLATIONS

1. Entity Integrity Constraints

The CIDB contains fields that are designed to store primary keys. Examples are cadet candidate social security number (AAB), admission participant identifier (FBA), educator identifier (FPA), test site identifier (FHA), high school Princeton number (FMA), and cadet social security number (CAA). As primary keys, these fields must contain unique values. If a primary key was allowed to contain identical values, the system would have no way to differentiate between the objects that contained the same primary key value. Any non-unique values for the primary key (this includes null values) will cause problems for the system and the user. Because the USMA system does not support entity integrity, it is imperative that the primary key fields be checked for uniqueness and that errors be validated prior to system conversion.

2. Referential Integrity Constraints

Two examples of referential integrity are provided in Figures 4 and 5. Figure 4 displays a simple example that shows how the CADET-PERM-COMPANY (CBN) and CADET-PERM-REGIMENT (CBO) of the CADET RECORD refer to the PERM-COMPANY (TEB) and the PERM-REGIMENT (TEC) of the PERMANENT COMPANY RECORD, respectively. It also shows how the CADET-GRAD-YEAR (CBC) of the CADET RECORD refers to the CLASS-GRADUATION-YEAR (TCA) of the CADET CLASS RECORD. If the cadet's permanent company and regiment was equal to G2, we would expect to find a value of G2 in the PERMANENT COMPANY RECORD. However, since the West Point system does not check for referential integrity, one would not know if this constraint held.

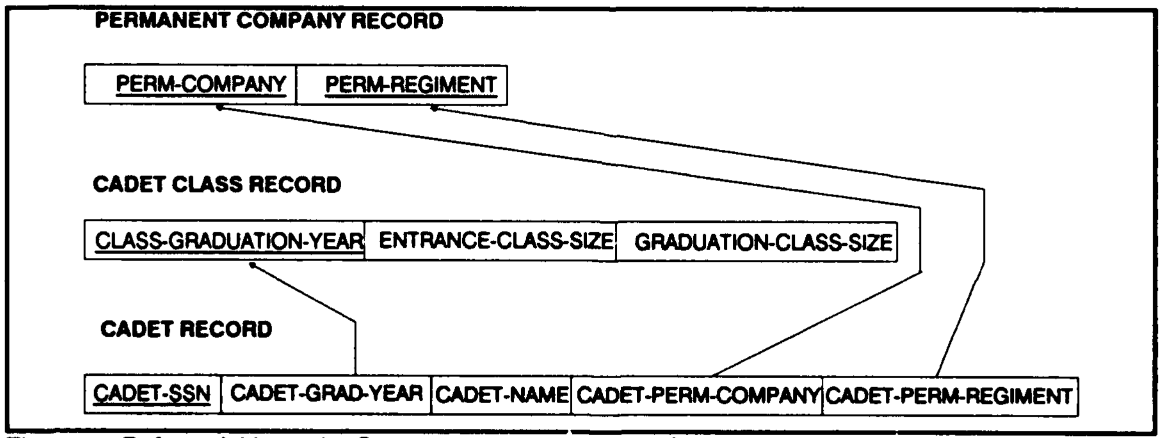

Figure 4. Referential Integrity Constraints from the CIDB Schema

Figure 5 displays a more involved example of referential integrity from the cadet candidate portion of the CIDB. From the INDIVIDUAL RECORD: AP-

IDENT (AQF) refers to AP-IDENT (FBA) of the ADMISSION PARTICIPANT record; HS-ETS-CODE (AHB) refers to the PRINCETON-NO (FMA) of the HIGH SCHOOL record, and the APPLICANT-TEST-SITE-CODE (AQK) refers to the SITE CODE (FHA) of the TEST SITE record. From the EDUCATOR record: EDUCATOR-AP-IDENT (FTI) refers to the AP-IDENT (FBA) of the ADMISSION PARTICIPANT record, and EDUCATOR-SITE-IDENT (FTJ) refers to the SITE CODE (FHA) of the TEST SITE record. Finally, from the HIGH SCHOOL record: HS-AP-IDENT (FOS) refers to the AP-IDENT (FBA) of the ADMISSION PARTICIPANT record, and the HS-SITE-IDENT (FOT) refers to the SITE CODE (FHA) of the TEST SITE record. As an example of referential integrity, if the

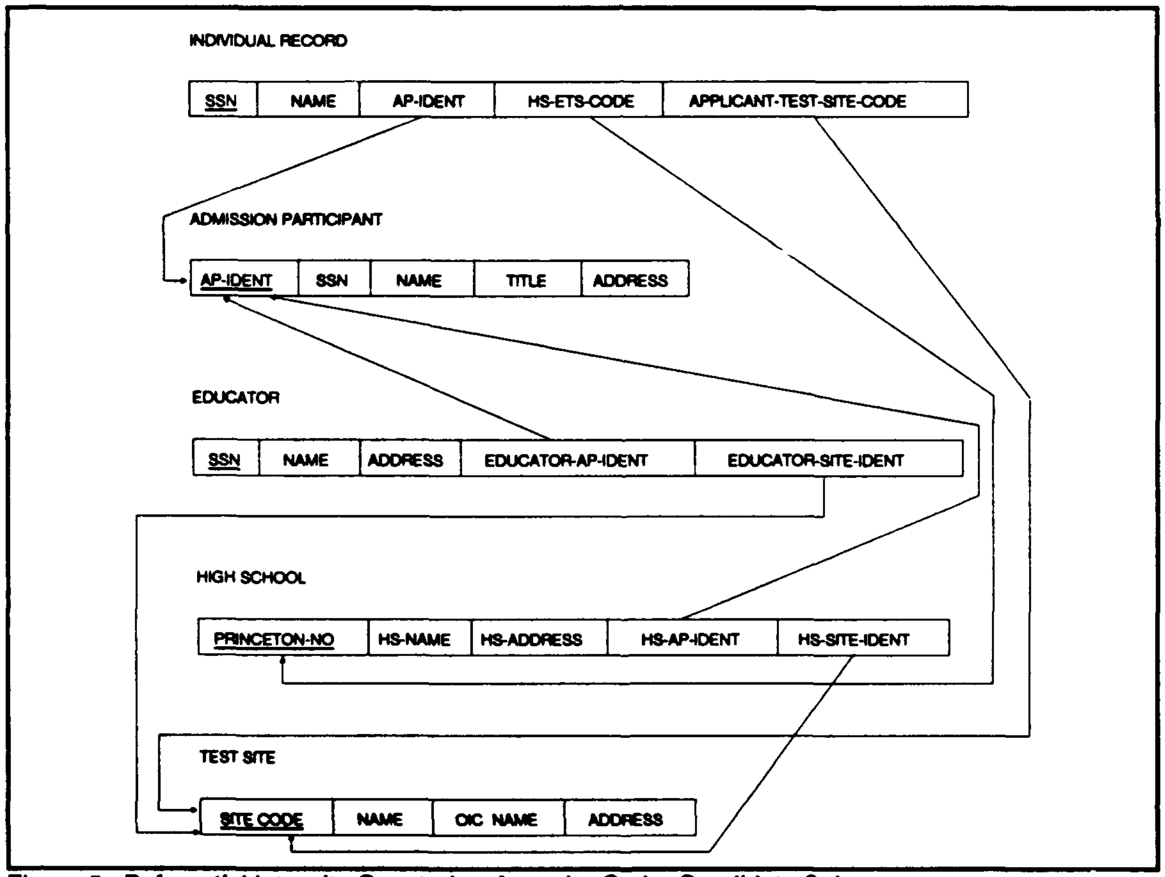

Figure 5. Referential Integrity Constraints from the Cadet Candidate Schema

applicant's test site code was equal to TN02, we would expect to find this value in the SITE CODE field of the TEST SITE record. As was the case in the previous example, the USMA user would not know whether the referential integrity constraint held because the USMA system does not check referential integrity constraints.

D. LOGICAL INCONSISTENCIES

As was discussed in Chapter Three, an organization may have significant numbers of logical constraints that it wishes to hold on the DBMS. With a database as large as the Military Academy's CIDB, the task of identifying all the semantic integrity constraints could be quite time consuming and would require the skills of an extremely knowledgeable team of individuals. The following is a partial list of logical constraints that must hold on the CIDB:

1. No cadet shall have a term QPA less than 1.67. If the term QPA falls below 1.67, place the cadet on academic probation.

2. No freshman cadet (plebe, class year equal to 4) shall have a cumulative QPA less than 1.70 following the second term of the freshman year. If the cumulative QPA is below 1.70 after the second term, place the cadet on academic probation.

3. No sophomore cadet (yearling, class year equal to 3) shall have a cumulative QPA less than 1.80 following the first term of the sophomore year or a cumulative QPA less than 1.85 following the second term of the sophomore year. If the cumulative QPA is below 1.80 at the end of the first term or below 1.85 after the second term, place the cadet on academic probation.

4. No junior cadet (cow, class year equal to 2) shall have a cumulative QPA less than 1.95 following either term of the junior year. If the cumulative QPA is less than 1.95 after either term, place the cadet on academic probation.

5. No senior cadet (firstie, class year equal to 1) shall have a cumulative QPA less than 2.00 following either term of the senior year. If the cumulative QPA is less than 2.00 after the first term, then place the cadet on academic probation. If the cumulative QPA is below 2.00 following the second term, do not allow the cadet to graduate.

6. No cadet shall enter USMA whose age is less than 17 or greater than 22 by the plebe class start date.

7. No cadet shall enter USMA without a high school diploma.

8. No cadet shall receive more than 14 demerits for an offense without being awarded an area or room disciplinary tour.

9. No cadet shall attend USMA for a period of more than four years without adjusting the graduation year.

10. To be eligible to underload (take less than the required six classes per term), cadets must have a cumulative QPA less than 2.00 or be a varsity corps squad athlete.

11. To be eligible to overload (take more than the required six courses per term), cadets must be class year 3, 2, or 1 (sophomore, junior or senior) and have made the dean's list in the preceding term. Or the cadet must be class year 1 (senior) with a cumulative QPA greater than 2.30.

12. No cadet will graduate from USMA without successfully completing 40 academic and eight physical education courses. This includes successful completion of 31 core curriculum courses and nine specific field of study (FOS) or major courses. Again, the cumulative QPA for graduation must be greater than 2.00.

13. No cadet will be considered a Distinguished Cadet unless in the top 5 percent of the class (from the class order of merit list).

14. No cadet will be considered for the dean's list unless the cadet's term QPA is greater than 3.00. Cadets are not eligible for the dean's list if they failed a course during the term, received an I (incomplete) for a course during the term, withdrew from USMA, are retaking a field of study course, or are underloading during the term.

15. To be eligible for the Superintendent's Award, a cadet's yearly QPA must be greater than 3.00, the cadet must be in the top third of the class (OM), and must pass all P.E. courses and the APFT during the year.

16. The book quantity on hand (QOH) value must be greater than the number of cadets scheduled for the course planning to use the book.

17. The estimated delivery date for books must be prior to the class start date.

18. If the number of classes taken during a term by a cadet is less than the minimum allowed for the class (minimum academic load), then alert the cadet's tactical officer with a message unless the cadet is eligible to underload.

19. If the cadet class is equal to 3 and the number of classes completed is less than 12, then alert the tactical officer with a message.

20. If the cadet class is equal to 2 and the number of classes completed is less than 24, then alert the tactical officer with a message.

21. If the cadet class is equal to 1 and the number of classes completed is less than 36, then alert the tactical officer with a message.

22. The classroom capacity must be greater than or equal to the number of cadets scheduled for the class.

23. Two classes cannot be scheduled for the same classroom at the same time.

24. No cadet will be scheduled for more than one class per hour.

25. No professor will teach more than one class per hour.

26. No male cadet will enter USMA with height less than 60 inches or greater than 80 inches or weight less than 100 pounds or more than 280 pounds.

27. No female cadet will enter USMA with height less than 58 inches or greater than 80 inches or weight less than 90 pounds or greater than 201 pounds.

28. No cadet company will exceed 120 cadets.

29. No cadet company will have less than 20 seniors, 20 juniors, 20 sophomores and 20 freshmen.

30. A cadet who validates a course should not be scheduled to take that course.

31. A cadet's high school class ranking cannot be greater than the total number of students in the high school class.

32. The test site fill for the PAE cannot be larger than the capacity of the test site.

33. An admission participant must be a USMA graduate.

E. CONCLUSION

This chapter has discussed specific examples of possible common data errors, integrity constraint violations and logical inconsistencies that may be present in West Point's CIDB. It was developed from the USMA CIDB Dictionary [Ref. 6] and the

USMA Academic Redbook [Ref. 7]. The idea that must be stressed is that any DBMS has the potential to contain errors in its data. In this regard, the USMA DBMS is not unique. Now that these potential errors have been identified, the focus shifts to the task of checking, validating and correcting them prior to transferring the data to the target system.

VI. PROPOSED METHOD OF RESOLVING THE ISSUES

A. INTRODUCTION

Chapter Five outlined specific examples from the CIDB of potential errors, integrity constraint violations, and logical inconsistencies that may be contained in the data. It is from these specific examples that this chapter is derived. The purpose of this chapter is threefold: First, to describe a generalized methodology that any organization could follow to identify errors and inconsistencies in its DBMS. Second, to outline a specific method that the USMA DOIM staff can follow to check the CIDB for possible errors and inconsistencies that are potentially stored in the database. Third, to discuss several implementations using the specific method described above as a guide. These implementations will be done on selected applications from the CIDB. The code that supports these implementations is written in PASCAL and can be found in Appendix B. It is not the intent of this thesis to exhaustively test every possible application from the CIDB, but to select examples to demonstrate what is done and how it is done to ensure that the information contained in a particular application is clean. The implementations selected come from a representative set of examples of potential errors contained in the CIDB, including an out-of-range value check, an incompatible data type check, and a referential integrity check.

B. GENERAL METHODOLOGY

There are many possible alternatives to choose from when deciding upon a general methodology for error detection and correction. Four methods to check the information stored in any DBMS are described below. Realistically, one of these methods, or some other similar method, must be followed on every potential inconsistency in the database. Only through checking each possible error can the organization confirm that the data is without fault. Let's look at these four general methods:

The first method involves checking the data directly in the source system. After errors are detected, validated and corrected, the data can be moved to the target system. If one has a thorough understanding of the source system and the programming language that supports it, this method might be quite attractive. For the USMA DOIM staff, fluent in the inner workings of the system, making checks directly in the source system could be the best means for error detection. The disadvantage to this method is that to apply it a programmer must be completely familiar with the programming language of the system: COBOL in the case of West Point. Additionally, one must have easy access to the source system. For these reasons, i.e., proficiency in COBOL and distance from West Point, using this first general method is not well suited to our needs.

A second method follows these general steps:

1. For a given application, unload the required information from the source system into intermediate files. An intermediate file will be generated for the appropriate corresponding file in the source system. For example, the database has N files where the social security number is stored. This means that N intermediate files would be generated. To check that the SSN field contains integer values, the intermediate files would store SSNs in positions one through eleven in a long column.

2. Run a specific application program on each intermediate file to check for and identify potential errors. The application program may require reading in and checking against more than one intermediate file. For example, in checking a potential subset-set error, at least two intermediate files must be read by the application program. In this case, the subset intermediate file and the set intermediate file are read by the application program and each record from the subset is checked against a like record from the set.

3. As errors are identified, mark the record containing the error in the intermediate file with an integer (1, 2, 3,. . .), and generate two files. The first file, known as the error record file, will list the records that contain errors. (Again marked with corresponding integer values.) Records are marked with unique values because the primary key may contain an error and therefore must be corrected. If the primary key is the SSN and it is changed, we must be able to find the corresponding record in the intermediate file. In the example above, the error record file would contain those SSNs that did not have exactly nine integers. For example, the error record file would possibly contain a column of values similar to the following: 1 12345678A, 2 394**9826, etc. The second file, known as the error message file, will identify the specific errors contained in the error record file. From the example, the error message file would store information like the following: Record number 1, position nine of the SSN contains the letter A. Record number 2, positions four and five of the SSN contain a *, etc.

4. Using the error message file in conjunction with the error record file, validate the records that contain errors. This may require using an external source to find the correct data.

5. Once the correct information has been located, make corrections to the error record file. Once all corrections have been made, the error record file becomes the corrected error record file.

6. Overwrite the corrected error record file into the intermediate file.

7. The intermediate file is now ready for loading into the target system.

An advantage to generating these two files is that the error messages are separated from the records with the errors. This enables an operator to validate the errors and correct the mistakes directly in the error record file, thereby turning the error record file into a corrected error record file. The operator can then overwrite the corrected error record file into the intermediate file simply by locating the records with the same integer values. A second advantage to using an intermediate file method is that the application programs can be written in any programming language. In our case PASCAL. This means that the individual writing the applications does not have to learn a new programming language. Being able to work with a language with which one is familiar is a significant advantage.

A third method, which is a variation of the two methods previously described allows an operator to make on-line corrections to errors that are detected. This means that errors would be printed to the screen, and then an operator could access the necessary file to make corrections with an editor. An on-line correction method would be beneficial if the number of corrections to be made were minimal. However, this method would be most cumbersome if the number of errors per application exceeded three or four. Imagine trying to write down 100 errors that printed to the screen. Operator frustration with this method would quickly occur. Another disadvantage to this method would be trying to make on-line corrections to errors that were difficult to validate. An operator might sit idly waiting for a telephone call from a source capable of validating the information, unable to continue until the validated data is provided. Finally, this on-line method would be the most time-consuming of the four general methods discussed.

A fourth method is a variation of the second method described above. This method has two differences when compared with the second general method. The first difference is that three files are generated instead of two. These three files are known as the good data file, the error record file and the error message file. The second difference is that rather than overwriting the corrected error record file into the intermediate file, this method merges the corrected error record file with the good data file. The advantages of this methodology are similar to the advantages for method two.

The general methods described above can have many different variations. Many combinations are possible. These examples illustrate some simple means to detect and correct database errors. There are many others. It is not the scope of this thesis to study and report on all the methods available, nor is it my intent to provide statistical data on the most cost effective, fast, or efficient method. I have selected a method for accomplishing the error detection task in a manner that best fits the needs of this thesis.

C. SPECIFIC METHOD

In terms of this thesis, the best methodology for checking the information contained in the CIDB follows closely to the last general method previously discussed above. This method's advantages are:

1. Application programs supporting the method can be written in the PASCAL programming language.
2. Error records are separated from the error messages. This will enable operators to validate the error records in any order, thus making the error correction process quite flexible. As validated information is provided, error records can be corrected.
3. The merging of the good data file and the corrected error record file is straightforward and easy to follow.
4. This method can be performed here in Monterey as easily as in New York. Intermediate data files are readily transferred through the ARPANET's file transfer protocol.
5. Record correction can be accomplished using a familiar text editor.

Figure 6 is provided as a means to help understand the specific methodology. Appendix C contains examples of the intermediate file, good data file, error record file, error message file and the corrected good data file from the application programs. Let us review the steps to be followed:

1. Unload all required information from the source system into intermediate files. One intermediate file will be generated for each corresponding file in the source system.
2. Run the specific applicatiôn program on each intermediate file to check for and identify errors. Count the number of intermediate file records read by the application program.

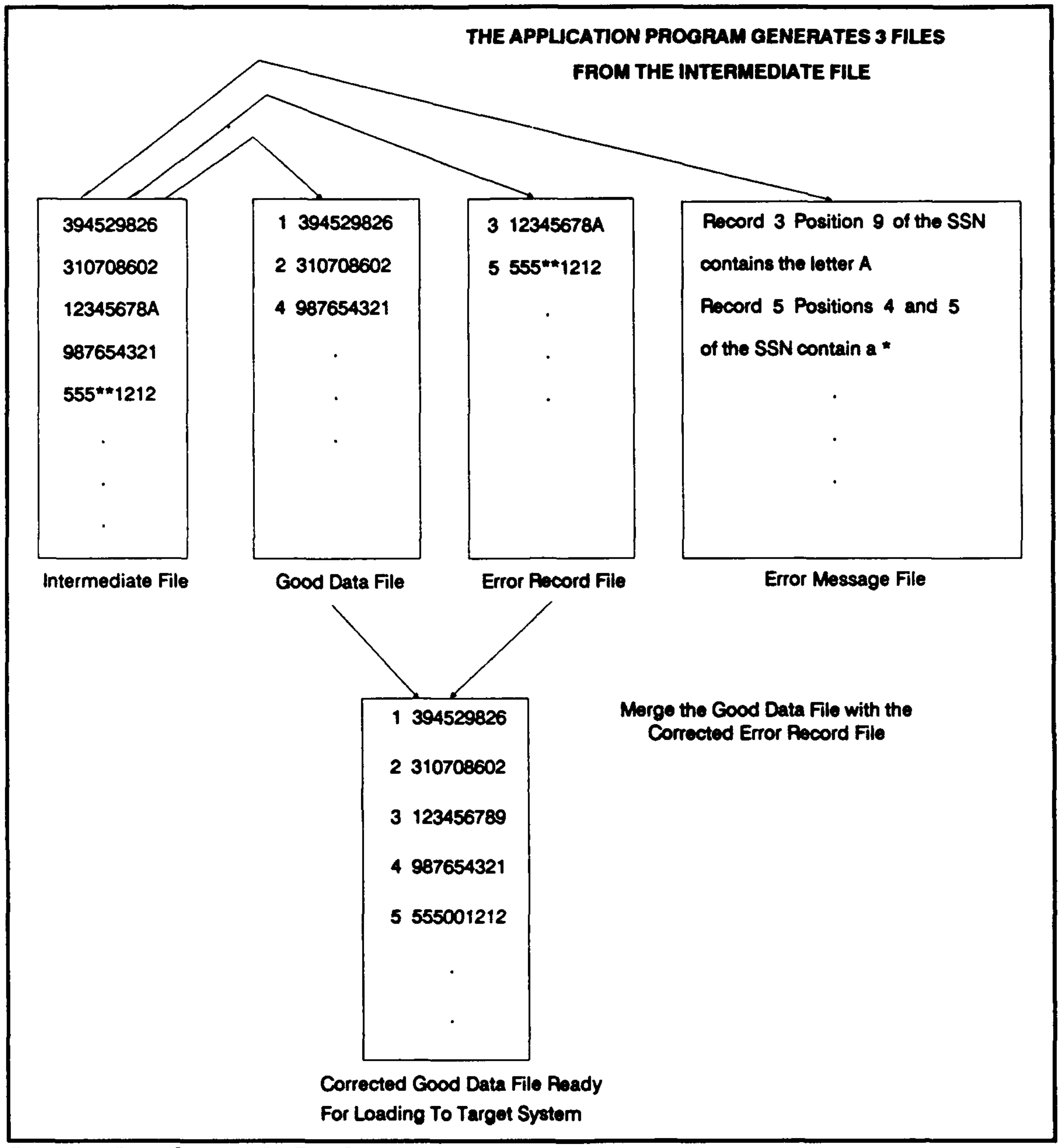
THE APPLICATION PROGRAM GENERATES 3 FILES
FROM THE INTERMEDIATE FILE
394529826
310708602
12345678A
987654321
555**1212
1 394529826
2 310708602
4 987654321
3 12345678A
5 555**1212
Record 3 Position 9 of the SSN contains the letter A
Record 5 Positions 4 and 5 of the SSN contain a *
Intermediate File
Good Data File
Error Record File
Error Message File
1 394529826
2 310708602
3 123456789
4 987654321
5 555001212
Merge the Good Data File with the Corrected Error Record File
Corrected Good Data File Ready For Loading To Target System

Figure 6. Files Created By The Specific Method

3. As records are read by the application program, three files are generated. The first file contains the records that are clean and is called the good data file. Each record is marked with an appropriate integer value (1, 2, 4, . . .). The second file contains those records with errors and is called the error record file. The records in error are also marked by the appropriate integer value (3, 5, . . .). Finally, an error message file is generated to assist the operator in the error validation process.

4. Using the error message file in conjunction with the error record file, validate the records that contain errors. This may require using an external source to obtain the correct data.

5. Once the correct information has been located, make corrections to the error record file. Once all corrections have been made, the error record file becomes the corrected error record file.

6. Merge the corrected error record file with the good data file and count the total number of records in the corrected good data file.

7. Ensure the number of records in the corrected good data file is equal to the number of records stored in the intermediate file.

8. The corrected good data file is ready for loading into the target system.

D. IMPLEMENTATIONS

1. Out-of-Range Values

The purpose of this implementation is to perform range checks on the sex, height, weight and birthdate fields of the individual record from the cadet candidate portion of the CIDB. The following algorithm will be used:

a. By class, load the SSN, name, height, weight, sex, and birthdate into an intermediate file. This check must be accomplished by class because the range check on the birthdate field will change depending on the entrance year. Following the format of the source system, positions one through eleven of the intermediate file will store the SSN, positions 12 through 38 the name, positions 39 and 40 the height, positions 41 through 43 the weight, position 44 the sex, and positions 45 through 50 will contain the birthdate.

b. An application program will be run on the intermediate file to identify out-of-range values for the four fields mentioned above.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

2. Incompatible Data Types

The purpose of this implementation is to ensure that the social security number has nine integers contained in it for each file storing the SSN. The following algorithm will be used:

a. For each file containing the SSN field, load the SSN into a separate intermediate file. The SSN will be stored in positions one through eleven of the intermediate files. For this application, the check will be made on the SSN from the individual record of the cadet candidate.

b. An application program will be run on the intermediate f̈⁻ to identify any incompatible data types it may contain. Because West Point normally only uses the first nine positions of the SSN field in its source system, this check will look for either nine integers in positions one through nine, with blanks in positions ten and eleven, or if dashes are used for separators, will look for dashes in positions four and seven with integers stored in the other nine positions.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

3. Redundancies

The purpose of this implementation is to ensure that redundant fields contained in the CIDB store the same information for a given cadet. This check will look at a cadet's entrance height and weight from the entrance and high school record to ensure that they are the same as the cadet's height and weight from the individual record. This check will be accomplished by class. The following algorithm will be used:

a. By class, load the SSN, name, height and weight from the individual record and the SSN, name, height and weight from the entrance and high school record into intermediate files. Both files will store the SSN in positions one through eleven, the name in positions 12 through 38, the height in positions 39 and 40, and the weight in positions 41 through 43.

b. Run an application program on the intermediate files to ensure that for each cadet the height and weight stored in the entrance and high school record intermediate file has identical entries in the individual record intermediate file. Those SSNs that do not have identical heights and weights will be identified, as well as those records found in the entrance and high school record but not in the individual record.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

4. Referential Integrity

The purpose of this implementation is to ensure that referential integrity holds for the company and regiment to which a cadet is assigned. This check will look at a cadet's company and regiment from the cadet record to ensure that the company and regiment are contained in the permanent company record. This check will be accomplished by class. The following algorithm will be used:

a. By class, load the SSN, name, company and regiment from the cadet record and the companies and regiments from the permanent company record into intermediate files. The cadet record file will store the SSN in positions one through eleven, the name in positions 12 through 38, and the company and regiment in positions 39 through 40, while the permanent company file will contain the company and regiment data in positions one and two.

b. Run an application program on the intermediate files to ensure that for each cadet the company and regiment stored in the cadet record intermediate file has an identical entry in the permanent company intermediate file. A null value is allowed for the cadet's company and regiment values. Those SSNs that do not have a company and regiment that is referenced will be identified.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

5. Entity Integrity

The purpose of this implementation is to ensure that entity integrity holds for the SSN field of the cadet candidate individual record. This check will look at a cadet's SSN from the individual record to ensure that there are no duplicate SSNs contained in the individual record. This check will be accomplished by class. The following algorithm will be used:

a. By class, load the SSN from the individual record into an intermediate file. The individual record file will store the SSN in positions one through eleven.

b. Run an application program on the intermediate file to ensure that for each SSN there are no duplications stored in the individual record intermediate file. Those SSNs that are duplicated will be identified. Null values are not allowed in the primary key field.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

6. Logical Inconsistencies

The purpose of this implementation is to ensure the validity of the logical implication that a cadet's high school ranking cannot be greater than the number of students in the cadet's high school class. This check will look at the value of a cadet's high school ranking from the entrance and high school record to ensure that it is not larger than the value of the number of students in the cadet's high school graduating class. This check will be accomplished by USMA class. The following algorithm will be used:

a. By class, load the SSN, name, high school ranking and the high school number in class from the entrance and high school record into an intermediate file. The intermediate file will store the SSN in positions one through eleven, the name in positions 12 through 38, the high school ranking in positions 39 through 42, and the high school number in class in positions 43 through 46.

b. Run an application program on the intermediate file to ensure that for each cadet the high school ranking is less than the high school number in class in the intermediate file. Those SSNs that have a high school ranking that is larger than their high school number in class will be identified.

Steps c through h are identical to steps 3 through 8 of the specific method above and are not repeated.

E. CONCLUSION

The implementations described in this chapter represent examples of algorithms designed to identify potential errors that may be present in the CIDB. These implementations are not by any means an all-inclusive set. Rather, they are designed to demonstrate selected applications where possible errors and inconsistencies exist in the CIDB. Appendix B contains the PASCAL programs written to support these applications. Sample output from each of the programs can be found in Appendix C. A discussion of the results of these sample program runs can be found in Chapter Seven of this thesis.

VII. CONCLUSIONS AND RECOMMENDATIONS

The specific programs of this thesis were developed to run on selected data provided by the USMA DOIM staff. These programs identified three errors that were contained in the data. The errors were: A cadet's height value equal to 30 inches (an out-of-range value), a cadet's sex with no value entered, and a cadet's birthdate that was left blank. While a total of three errors may seem a negligible amount, when multiplied across a database as large as the CIDB, this number becomes significant. These three errors support the basic premise of this thesis: In any database there are potentially many errors that must be checked for and corrected prior to system conversion. Overall, the data that was checked by the thesis' programs looked acceptable, the three errors notwithstanding. However, the DOIM staff should beware that many more errors are possible. This statement is based on the fact that only a small portion of CIDB data was actually checked.

This thesis, together with the companion work of Guilmette and Wilson [Ref. 2], was part of a project to design and convert the existing source database in network form to a target relational database management system for the United States Military Academy at West Point. The three of us worked together closely for the USMA project, but then split apart to develop two separate theses. This thesis has shown that many potential errors and inconsistencies are possible in any DBMS and in particular the current USMA system. These potential errors must be checked before the system is converted. Additionally, once the data has been checked and ultimately moved into the new target system, it stands to reason that the same type of errors and inconsistencies may occur in the target system as in the source if steps are not taken by the DOIM staff to strictly enforce the ideas discussed in Chapters Two and Three of this thesis. The target DBMS must have constraints implemented to keep the data clean.

In summation, the following recommendations are made. These recommendations apply not only to West Point, but to any organization contemplating system conversion.

1. All DBMS potentially contain many errors in the data the system stores. Steps must be taken prior to system conversion to check the information stored in the DBMS to ensure that it is without error. If there is "garbage" in the source system, there will be "garbage" in the target system when the data is converted.

2. Most modern systems do not support automatic enforcement of integrity constraints. Consequently, the new target system, when it is fielded by West Point, must have range checks developed, restrictive data types specified, integrity constraints built with triggers or procedural coding, and other defensive measures taken that will decrease the opportunity for errors in the system. Without the development of an error checking package in the target system, efforts to clean the data prior to system conversion will be for naught.

The process of database management system conversion can yield significant improvements to an organization's system and benefits to its users. To arrive at a target system that ensures integrity and minimal opportunity for error requires planning and communication between the database manager and system users. The process of designing and developing integrity constraints and applying them to the source and target systems is as important to DBMS conversion as is developing the code to implement the new system.

A team approach such as the one used by Hendrickson, Guilmette and Wilson appears to be an ideal way to approach the conversion task. By designating team members to oversee particular functions, such as system design, coding and integrity maintenance, each operates as a specialist yet understands the overall goals for the conversion.

Before any conversion can take place, the data in the source system must be checked for validity and accuracy. If this task is done properly, and adequate planning and communication are in place, the database manager can be confident of a smooth transition toward an enhanced and error free target DBMS.

APPENDIX A

USMA FIELD IDENTIFIERS NOT DESCRIBED IN TEXT

I. OUT-OF-RANGE VALUES

A. Cadet Candidate

1. SSN
   a. AAB/Individual-SSAN-Service-Number
   b. FBH/AP-LO-SSAN
   c. NHA/Nomination-Candidate-SSAN

2. Julian Dates
   a. ABF/Individual-Status-Date
   b. ABG/Offer-of-Admission-Date
   c. ABH/Status-Elaboration-Date
   d. AFC/Record-Creation-Date
   e. AFD/Record-Last-Update-Date
   f. ASC/Academic-Status-Date
   g. ASD/Physical-Aptitude-Status-Date
   h. ASE/Medical-Status-Date
   i. ASF/Leadership-Status-Date
   j. ASG/Second-Step-Kit-Sent-Date
   k. ASH/5-413-Date-5-480-Date
   l. ASI/Special-Letter-One-Date
   m. ASJ/Special-Letter-Two-Date

3. Height and Weight
   a. ADD/Height-of-Individual
   b. ADE/Weight-of-Individual

4. Birth month and day
   a. ADI/Birth-Month
   b. ADJ/Birth-Day

5. Physical Activity Exam and Admissions Scores
   a. AGG/Physical-Activity-Exam-Score
   b. AGH/PAE-Score2
   c. AGI/PAE-Score3
   d. AGK/Leadership-Potential-Score
   e. AMB/PAE-Event-One-Score
   f. AMC/PAE-Event-Two-Score
   g. AMD/PAE-Event-Three-Score
   h. AME/PAE-Event-Four-Score
   i. AMF/PAE-Event-Five-Score

j. APB/Extracurricular-Activities-Score
k. APC/Athletic-Activities-Score
l. APD/Faculty-Appraisal-Score
m. APE/High-School-Class-Rank-Score

6. ACT and SAT Scores
 a. AJB/SAT-Math
 b. AJC/SAT-Verbal
 c. AJD/Second-SAT-Math
 d. AJE/Second-SAT-Verbal
 e. AJF/SAT-Math-Average
 f. AJG/SAT-Verbal-Average
 g. AKB/ACT-Math-Score
 h. AKC/ACT-English-Score
 i. AKD/ACT-Natural-Science-Score
 j. AKE/ACT-Social-Science-Score
 k. AKF/Second-Math-Score
 l. AKG/Second-English-Score
 m. AKH/Second-Natural-Science-Score
 n. AKI/Second-Social-Science-Score
 o. AKJ/ACT-Math-Average
 p. AKK/ACT-English-Average
 q. AKL/ACT-Natural-Science-Average
 r. AKM/ACT-Social-Science-Average

7. Work Experience
 AOG/Work-Experience-Years

8. Single Character Fields
 a. ADC/Sex-of-Individual
 b. ADF/Individual-Ethnic-Background
 c. ADG/Race-of-Individual
 d. ARB/Interview-on-File-Flag
 e. ARC/Candidate-Personal-Statement-Code
 f. ARD/Employers-Evaluation-Code
 g. ARE/Activities-Record-DD-1868-Flag
 h. ARF/Personal-Data-Record-DD-1867-Flag

9. Month Joined the Field Force
 a. FEF/AP-Month-Joined
 b. FTB/ED-Month-Joined

10. Nomination Vacancies
 a. NEG/NA-Vacancies-Allowed
 b. NEH/NA-Vacancies-Filled
 c. NEI/NA-Nominations-Authorized-Current-AY

11. Nomination Record
 NIC/Nomination-Selection-Score

12. Test Site Record
    a. FJI/Test-Date-1
    b. FJJ/Test-Date-2
    c. FJK/Test-Date-3
    d. FJL/Test-Date-4
    e. FJM/Test-Date-5
    f. FJN/Test-Date-6
    g. FJO/Test-Date-7
    h. FJP/Test-Time-1
    i. FJQ/Test-Time-2
    j. FJR/Test-Time-3
    k. FJS/Test-Time-4
    l. FJT/Test-Time-5
    m. FJU/Test-Time-6
    n. FJV/Test-Time-7

B. Schedule

1. SSN
    a. HPA/Sched-Cadet-SSAN
    b. HTA/Sched-Term-Plan-Cadet-SSAN

2. Julian Dates
    a. KCF/Sched-Book-Transaction-Date
    b. KCK/Sched-Book-Est-Delivery-Date

3. Book Quantities
    a. KCJ/Sched-Book-Quantity-On-Hand
    b. KCN/Sched-Book-Request-Quantity

4. Classroom Capacity
    HOE/Sched-Room-Capacity

5. Course Population
    a. HKS/Sched-Master-Crse-Population
    b. KBG/Sched-Trm-Ipd-Crse-Count

6. Course Enrollment
    a. KBD/Sched-Trm-Ipd-Max-Enrollemnt
    b. KBE/Sched-Trm-Ipd-Min-Enrollment
    c. KBF/Sched-Trm-Desired-Enrollment

7. Flags
    a. HQY/Sched-Cadet-Flag-Change
    b. HRA-HRK/Sched-Cadet-Group-Flags
    c. HOF/Sched-Room-Type
    d. HOG/Sched-Instruct-Period-Hour

C. Cadet

1. SSN
   a. CAA/Cadet-SSAN
   b. TJA/Trip-SSAN
   c. TVA/Yearly-Demerit-SSAN
   d. TPA/Monthly-Demerit-SSAN
   e. TKB/Commandant-SSAN
   f. DFB/Academic-SSAN

2. Height and Weight
   a. CAD/Cadet-Current-Height
   b. CAE/Cadet-Current-Weight
   c. CEH/Cadet-Entrance-Height
   d. CEI/Cadet-Entrance-Weight
   e. TKO/Cadet-Profile-Height
   f. TKP/Cadet-Profile-Weight

3. Physical Fitness Test
   CON/Cadet-Assign-APFT-Score

4. Class Size
   a. TCC/Entrance-Class-Size
   b. TCD/Graduation-Class-Size
   c. THF/Class-Size-Start-of-Term
   d. THG/Class-Size-End-of-Term

5. Graduate-Record-Exam-Scores
   a. DEP/GRE-Verbal
   b. DEQ/GRE-Quantitative
   c. DER/GRE-Analytical

6. Entrance Runs
   a. CEM/Entrance-Run-1
   b. CEN/Entrance-Run-2
   c. COO/Cadet-Assign-Run-Time

7. Trip Departure and Return Times
   a. TGQ/Trip-Departure-Time
   b. TGV/Trip-Return-Time

8. Cadet Illness
   a. TYB/Cadet-Illness-Date
   b. TYG/Cadet-Illness-Excused-to-Date
   c. TYC/Cadet-Illness-Time-Out
   d. TYD/Cadet-Illness-Time-In

9. Cadet Activity Record
   TMH/Cadet-Days-in-Activity

10. Orders of Merit
    a. TQN/Military-Dev-Index-Cum-OM
    b. DBE/Course-Max-OM
    c. DGB/Term-Academic-OM
    d. DGC/Term-General-OM
    e. DHB/Year-Summary-AOM
    f. DHC/Year-Summary-GOM
    g. DIB/Cumulative-AOM
    h. DIC/Cumulative-GOM
    i. DLM/Grades-Course-OM

11. Quality Point Averages
    a. DGH/Term-Acad-QPA
    b. DGI/Term-Gen-QPA
    c. DHH/Year-Summary-Acad-QPA
    d. DHI/Year-Summary-Gen-QPA
    e. DII/Cumulative-Acad-QPA
    f. DIJ/Cumulative-Gen-QPA

II. INCOMPATIBLE DATA TYPES

A. Cadet Candidate

1. SSN
    a. AAB/Individual-SSAN-Service-Number
    b. FBH/AP-LO-SSAN
    c. FPA/Educator-ID
    d. NHA/Nomination-Candidate-SSAN

2. Julian Dates
    a. ABF/Individual-Status-Date
    b. ABH/Status-Elaboration-Date
    c. AFC/Record-Creation-Date
    d. AFC/Record-Last-Update-Date
    e. ASC/Academic-Status-Date
    f. ASD/Physical-Aptitude-Status-Date
    g. ASE/Medical-Status-Date
    h. ASF/Leadership-Status-Date
    i. ASG/Second-Step-Kit-Sent-Date
    j. ASH/5-413-Date-5-480-Date
    k. ASI/Special-Letter-One-Date
    l. ASJ/Special-Letter-Two-Date

3. Zip Codes and Telephone Numbers
    a. ACJ/Address-Zip-Code-Individual
    b. ACK/Telephone-Number
    c. FCJ/AP-Zip-Code
    d. FCL/Home-Phone-Area-Code
    e. FCM/Home-Phone-Number
    f. FCN/Business-Area-Code

g. FOO/Business-Phone-Number
h. FCP/Business-Phone-Ext
i. FOQ/Autovon-Number
j. FCR/Autovon/Extension
k. FSD/Ed-Address-Zip-Code
l. FNF/High-School-Zip
m. NCI/NA-Address-Zip-Code
n. NCJ/NA-Telephone
o. NDH/NA-Asst-Address-Zip-Code
p. NDI/NA-Asst-Telephone
q. FIJ/Test-Site-Zip-Code
r. FIM/Test-Site-Telephone-Number
s. FIN/Test-Site-Telephone-Extension
t. FIO/Test-Site-Autovon-Number
u. FIP/Test-Site-Autovon-Extension
v. FIU/Test-Site-OIC-Zip

4. Height and Weight
   a. ADD/Height-of-Individual
   b. ADE/Weight-of-Individual

5. Graduation Year
   AEB/Transcript-Grad-Year

6. District Number
   AEE/Ent-Senator-or-District-No

7. Sequence Number
   AEF/Ent-Source-Sequence-No

8. Percent onto College
   AHH/Ind-Percent-onto-College

9. Month and Year Joined, Class Year and Training Year
   a. FEF/AP-Month-Joined
   b. FEG/AP-Year-Joined
   c. FEH/USMA-Class-Year
   d. FEK/Training-Year
   e. FTB/Ed-Month-Joined
   f. FTC/Ed-Year-Joined

B. Schedule

1. Telephone number
   HQI/Sched-Cadet-FOS-Advisor-Phone

2. Master Course Number
   HJD/Sched-Master-Crse-Number

3. Permanent Regiment
   HHC/Sched-Perm-Regt

C. Cadet

1. Zip Codes and Telephone Numbers
   a. CFE/Cadet-HS-Zip-Code
   b. CHI/Parent-Zip-Code
   c. CHQ/Second-Parent-Zip-Code
   d. CJF/Prior-College-Zip-Code
   e. TGF/Trip-Address-Zipcode
   f. TGG/Trip-Address-Phone

2. Course Number
   a. DAD/Course-Number
   b. DKC/Grades-Course-Number

APPENDIX B

APPLICATION PROGRAMS

The following computer programs were written in TURBO PASCAL Version 4.0, using a Leading Edge Model D2 (80286) personal computer (IBM compatible).

```
(***************************************************************************)
(* The purpose of this application program is to perform range checks on the*)
(* height, weight, sex and birthdate fields for each cadet candidate indi-  *)
(* vidual record.  Upon entry to USMA, male cadet's must be between 60 and  *)
(* 80 inches tall and weigh between 100 and 280 pounds.  For females, their *)
(* height must be between 58 and 80 inches and their weight must be between *)
(* 90 and 201 pounds.  Sex must be entered as either M for male or F for    *)
(* female.  A cadet's age must be not less than 17 and not older than 22 by *)
(* the class start date.  Out-of-range-value errors will cause two files to *)
(* be generated - an error record file and an error message file.  Records  *)
(* that are clean are written to a good data file.  Corrections are to be   *)
(* made to the error record file, and then it is to be merged with the good *)
(* data file.  The corrected good data file is to be stored for future load-*)
(* ing into the target system.  Use the program RangMerg to merge the good  *)
(* data file and the corrected error record file.                           *)
(***************************************************************************)

Program OutOfRangeValueCheck;
Uses CRT;

Const
  filename1  = 'file.dat';      (*cadet record file*)
  file1      = 'goodp1.dat';    (*good data file*)
  file2      = 'badp1.dat';     (*error record file*)
  file3      = 'emesagp1.dat';  (*error message file*)
  maxcadets  = 99;              (*maximum number of cadet records*)
  one        = 1;               (*minimum number of cadet records*)
  blank      = ' ';             (*blank character*)
  oldage     = '680701';        (*oldest birthadate allowable*)
  youngage   = '730701';        (*youngest birthdate allowable*)

Type
  numssn     = string[11];      (*cadet SSN*)
  personname = string[27];      (*cadet name*)
  inches     = string[2];       (*cadet height*)
  pounds     = string[3];       (*cadet weight*)
  morf       = string[1];       (*male or female*)
  birthdate  = string[6];       (*cadet birthdate*)
  cadet      = record                    (*cadet record*)
                 ssn    : numssn;
                 name   : personname;
                 height : inches;
                 weight : pounds;
                 sex    : morf;
                 date   : birthdate;
               end;

  cadetrec  = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  filein                 : text;      (*files to be read by the program*)
  person                 : cadetrec;  (*variable of type cadetrec*)
  count, error           : integer;   (*counters*)
  gooddata               : text;      (*file to be written by the program*)
  baddata, emessage      : text;      (*file to be written by the program*)
  ok                     : boolean;   (*true or false*)


Procedure ReadCadet(var filein : text; var count : integer);

(* This procedure reads the necessary data from the cadet candidate *)
(* individual record into an intermediate file to be processed. *)

Var
  i : integer; (* counter *)

begin (* ReadCadet *)
  assign(filein, filename1);
  reset(filein); (*reset the file*)
  i := 1; (*initialize variable*)
  count := 0; (*initialize variable*)
    while not eof(filein) do
      begin (*read cadet records into the file*)
        count := count + 1; (*increment the cadet record count*)
        read(filein, person[i].ssn, person[i].name, person[i].height);
        readln(filein, person[i].weight, person[i].sex, person[i].date);
        i := i + 1; (*increment counter*)
      end;
  close(filein); (*close the file*)
end; (* ReadCadet *)


Procedure RangeCheck(var filein : text; var count, error : integer;
                     var ok : boolean; var gooddata, baddata, emessage : text);

(* This procedure checks to insure that the height, weight, sex and birthdate*)
(* of a cadet are within an acceptable range. *)
```

```
Var
  i : integer; (*counters*)

begin (* RangeCheck *)
  error := 0; (*initialize variable*)
  for i := 1 to count do
    begin
      ok := true; (*set boolean flag to true*)
      if not((person[i].sex = 'M') or (person[i].sex = 'F')) then
        begin (*write records with sex out-of-range errors to a file*)
              (*named badp1.dat and error messages to file emesagp1.dat*)
          ok := false; (*set boolean flag to false*)
          write(baddata,i:2,person[i].ssn, person[i].name,person[i].height);
          writeln(baddata,person[i].weight,person[i].sex,person[i].date);
          write(emessage,'Sex value out-of-range!  Check sex for record ');
          writeln(emessage,i,'.');
          writeln(emessage,person[i].ssn, person[i].name,person[i].sex);
          writeln(emessage);
          error := error + 1; (*count the records with errors*)
        end
      else if ((person[i].sex = 'M') or (person[i].sex = 'F')) then
        begin (*check height and weight*)
          if (((person[i].sex = 'M') and
              ((person[i].height < '60') or (person[i].height > '80'))) or
              ((person[i].sex = 'F') and
              ((person[i].height < '58') or (person[i].height > '80')))) then
            begin (*write records with height out-of-range errors to a file *)
                  (*named badp1.dat and error messages to file emesagp1.dat.*)
              ok := false; (*set boolean flag to false*)
              write(baddata,i:2,person[i].ssn, person[i].name);
              write(baddata,person[i].height,person[i].weight,person[i].sex);
              writeln(baddata,person[i].date);
              write(emessage,'Height value out-of-range!  Check height for ');
              writeln(emessage,'record ',i,'.');
              writeln(emessage,person[i].ssn, person[i].name,person[i].height);
              writeln(emessage);
              error := error + 1; (*count the records with errors*)
            end;
          if (((person[i].sex = 'M') and
              ((person[i].weight < '100') or (person[i].weight > '280'))) or
              ((person[i].sex = 'F') and
              ((person[i].weight < '090') or (person[i].weight > '201')))) then
            begin (*write records with weight out-of-range errors to a file *)
                  (*named badp1.dat and error messages to file emesagp1.dat.*)
              if ok = false then
                begin (*a previous error in the record exists*)
                  write(emessage,'Weight value out-of-range!  Check weight ');
                  writeln(emessage,'for record ',i,'.');
                  write(emessage,person[i].ssn,person[i].name);
                  write(emessage,person[i].weight);
                writeln(emessage);
              end
            else if ok = true then
              begin (*no previous errors exist in the record*)
                ok := false; (*set boolean flag to false*)
                write(baddata,i:2,person[i].ssn, person[i].name);
                write(baddata,person[i].height,person[i].weight);
                writeln(baddata,person[i].sex,person[i].date);
                write(emessage,'Weight value out-of-range!  Check weight ');
                writeln(emessage,'for record ',i,'.');
                write(emessage,person[i].ssn,person[i].name);
                write(emessage,person[i].weight);
                writeln(emessage);
                error := error + 1; (*count the records with errors*)
              end;
          end;
        end;
      if (person[i].date < oldage) or (person[i].date > youngage) then
        begin (*write records with birthdate out-of-range errors to a file*)
              (*named badp1.dat and error messages to file emesagp1.dat.*)
          if ok = false then
            begin (*a previous error exists in the record*)
              write(emessage,'Birthdate value out-of-range!  Check ');
              writeln(emessage,'birthdate for record ',i,'.');
              writeln(emessage,person[i].ssn,person[i].name,person[i].date);
              writeln(emessage);
            end
          else if ok = true then
            begin (*no previous errors exist in the record*)
              ok := false; (*set boolean flag to false*)
              write(baddata,i:2,person[i].ssn,person[i].name);
              write(baddata,person[i].height,person[i].weight,person[i].sex);
              writeln(baddata,person[i].date);
              write(emessage,'Birthdate value out-of-range!  Check ');
              writeln(emessage,'birthdate for record ',i,'.');
              writeln(emessage,person[i].ssn,person[i].name,person[i].date);
              writeln(emessage);
              error := error + 1; (*count the records with errors*)
            end;
        end;
      if ok = true then
        begin (*no errors exist in the record*)
          ok := true;
          write(gooddata,i:2,person[i].ssn, person[i].name,person[i].height);
          writeln(gooddata,person[i].weight,person[i].sex,person[i].date);
        end;
    end;
end; (* RangeCheck *)
```

```
begin (* main application - OutOfRangeValueCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata, file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata, file2);
  rewrite(baddata); (*write to a file*)
  assign(emessage, file3);
  rewrite(emessage); (*write to a file*)
  writeln('Out of Range Value Check For Height, Weight, Sex and Birthdate':72);
  ReadCadet(filein, count);
  RangeCheck(filein,count,error,ok,gooddata,baddata,emessage);
  writeln('There are ':36, error, ' errors detected.');
  if error > 0 then
   begin
    writeln('Check files badp1.dat and emesagp1.dat to make corrections.':69);
   end;
  writeln('The number of records read from the input file was ':65,count,'.');
  writeln('This Application Program is Now Finished!':60);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - OutOfRangeValueCheck *)
```

```
(*****************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Out-Of-Range Value Program into one corrected file for future loading    *)
(* into the target system.  Execute this program after running Program1.    *)
(*****************************************************************************)

Program OutOfRangeValueMergeData;
Uses CRT;

Const
  filename1 = 'goodp1.dat';   (*good data file*)
  filename2 = 'badp1.dat';    (*error record file*)
  filename3 = 'corectp1.dat';(*corrected and merged data file*)
  maxcadets = 99;             (*maximum number of cadet records*)
  one       = 1;              (*minimum number of cadet records*)
  blank     = '';             (*blank character*)

Type
  count      = string[2];     (*record count number*)
  numssn     = string[11];    (*cadet SSN*)
  personname = string[27];    (*cadet name*)
  inches     = string[2];     (*cadet height*)
  pounds     = string[3];     (*cadet weight*)
  morf       = string[1];     (*male or female*)
  birthdate  = string[6];     (*cadet birthday*)

  cadet      = record                     (*cadet record*)
                 k      : count;
                 ssn    : numssn;
                 name   : personname;
                 height : inches;
                 weight : pounds;
                 sex    : morf;
                 date   : birthdate;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2      : text;     (*files to be read by the program*)
  master            : text;     (*file to be written by the program*)
  buffer1, buffer2  : cadetrec; (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)
```

```
begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  write(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  write(master,buffer1[i].height,buffer1[i].weight,buffer1[i].sex);
  writeln(master,buffer1[i].date); (*write to file*)
  i := i + 1; (*increment counter*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].height);
  readln(file1,buffer1[i].weight,buffer1[i].sex,buffer1[i].date);
end; (* GoodData *)


Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  write(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  write(master,buffer2[j].height,buffer2[j].weight,buffer2[j].sex);
  writeln(master,buffer2[j].date); (*write to file*)
  j := j + 1; (*increment counter*)
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].height);
  readln(file2,buffer2[j].weight,buffer2[j].sex,buffer2[j].date);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;  (*counters*)
  buffer1, buffer2             : cadetrec; (*variable of type cadetrec*)

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].height);
  readln(file1,buffer1[i].weight,buffer1[i].sex,buffer1[i].date);
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].height);
  readln(file2,buffer2[j].weight,buffer2[j].sex,buffer2[j].date);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
```

```
          begin
            writeln('Both files are empty':50);
          end
        else if (buffer2[j].k = blank) then
          begin (*all records are in the good data file*)
            GoodData(count1, i, buffer1, master, file1);
          end
        else if (buffer1[i].k = blank) then
          begin (*all records are in the error record file*)
            BadData(count2, j, buffer2, master, file2);
          end
        else if (buffer1[i].k < buffer2[j].k) then
          begin (*record in buffer1 goes into correct file*)
            GoodData(count1, i, buffer1, master, file1);
          end
        else if (buffer1[i].k > buffer2[j].k) then
          begin (*record in buffer2 goes into correct file*)
            BadData(count2, j, buffer2, master, file2);
          end
      end;
    count3 := count1 + count2; (*records read should equal value from int. file*)
    until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
    writeln('Number of Records Read Equals ':55, count3);
  end; (* Merge *)


  begin (* main application - OutOfRangeValueMergeData *)
    clrscr; (*clear the screen*)
    assign(file1,filename1);
    reset(file1); (*reset the file*)
    assign(file2,filename2);
    reset(file2); (*reset the file*)
    assign(master,filename3);
    rewrite(master); (*write to a file*)
    writeln('Merging Started':48);
    Merge(file1,file2,master);
    writeln('END of APPLICATION':50);
    close(file1);  (*close the file*)
    close(file2);  (*close the file*)
    close(master); (*close the file*)
  end. (* main application - OutOfRangeValueMergeData *)
```

```
(*************************************************************************)
(* The purpose of this application program is to insure that the Social   *)
(* Security Number field has integers contained in it.  Either nine       *)
(* consecutive digits or nine digits with dashes in positions four and    *)
(* seven are allowed for the SSN value.  Incompatible data type checking is *)
(* required because the USMA system allows any character value to be      *)
(* entered for a SSN.  Incompatible data type violations will cause two   *)
(* files to be generated - an error record file and an error message file. *)
(* Records that are clean are written to a good data file.  Corrections are *)
(* to be made to the error record file, and then it is to be merged with the*)
(* good data file.  The corrected good data file is to be stored for future *)
(* loading into the target system.  Use the program IDTMerge to merge the  *)
(* good data file and the corrected error record file.                    *)
(*************************************************************************)

Program IncompatibleDataTypeCheck;
Uses CRT;

Const
  filename = 'cadet.dat';     (*cadet record file*)
  file1    = 'goodp2.dat';    (*good data file*)
  file2    = 'badp2.dat';     (*error record file*)
  file3    = 'emesagp2.dat';  (*error message file*)
  blank    = ' ';             (*blank character*)
  dash     = '-';             (*dash character*)

Type
  row      = array[1..80] of char; (*max 80 characters per row*)
  numssn   = array[1..11] of char; (*cadet SSN*)
  person   = array[1..27] of char; (*cadet name*)

Var
  filein              : text;     (*file to be read by the program*)
  line                : row;      (*variable of type row*)
  ssn                 : numssn;   (*variable of type numssn*)
  name                : person;   (*variable of type person*)
  i, count, error     : integer;  (*counters*)
  gooddata            : text;     (*file to be written by the program*)
  baddata, emessage   : text;     (*file to be written by the program*)
  ok                  : boolean;  (*true or false*)


Procedure ErrorMessage1(var ok : boolean; var i, count, error : integer;
                        var ssn : numssn; var name : person;
                        var baddata, emessage : text);

(* This procedure writes the records with one error to the error record and *)
(* error message files. *)
```

```
begin (* ErrorMessage1 *)
  ok := false; (*set boolean flag to false*)
  writeln(baddata,count:2,ssn,name);
  write(emessage,'Error in SSN - position ',i,'.  Check SSN for ');
  writeln(emessage,'record ',count);
  writeln(emessage,ssn,name:40);
  writeln(emessage);
  error := error + 1; (*count the records with errors*)
end; (* ErrorMessage1 *)


Procedure ErrorMessage2(var i, count : integer; var ssn : numssn;
                        var name : person; var emessage : text);

(* This procedure writes the records with more than one error to the error *)
(* message file only. *)

begin (* ErrorMessage2 *)
  write(emessage,'Error in SSN - position ',i,'.  Check SSN for ');
  writeln(emessage,'record ',count);
  writeln(emessage,ssn,name:40);
  writeln(emessage);
end; (* ErrorMessage2 *)


Procedure SSNCheck(var count, error : integer; var ssn : numssn;
                   var name : person; var gooddata, baddata, emessage : text);

(* This procedure checks the validity of the data type for the SSN field and *)
(* insures that nine digits for SSN are contained in the record. *)

Var
  i  : integer; (*counter*)
  ok : boolean; (*true or false*)

begin (* SSNCheck *)
  ok := true; (*set boolean flag to true*)
  for i := 1 to 3 do
    begin (*check first three digits of SSN*)
      if not (ssn[i] in ['0'..'9']) then
        begin
          ErrorMessage1(ok, i, count, error, ssn, name, baddata, emessage);
        end;
    end;
  for i := 4 to 4 do
    begin (*check fourth digit of SSN*)
      if not((ssn[4] = dash) or (ssn[4] in ['0'..'9'])) then
        begin
          if ok = false then
```

```
                begin
                  ErrorMessage2(i, count, ssn, name, emessage);
                end
              else if ok = true then
                begin
                  ErrorMessage1(ok, i, count, error, ssn, name, baddata, emessage);
                end;
          end;
      end;
    for i := 5 to 6 do
      begin (*check fifth and sixth digits of SSN*)
        if not(ssn[i] in ['0'..'9']) then
          begin
            if ok = false then
              begin
                ErrorMessage2(i, count, ssn, name, emessage);
              end
            else if ok = true then
              begin
                ErrorMessage1(ok, i, count, error, ssn, name, baddata, emessage);
              end;
          end;
      end;
    for i := 7 to 7 do
      begin (*check seventh digit of SSN*)
        if not((ssn[7] = dash) or (ssn[7] in ['0'..'9'])) then
          begin
            if ok = false then
              begin
                ErrorMessage2(i, count, ssn, name, emessage);
              end
            else if ok = true then
              begin
                ErrorMessage1(ok, i, count, error, ssn, name, baddata, emessage);
              end;
          end;
      end;
    for i := 8 to 11 do
      begin (*check last four digits of SSN*)
        if (ssn[4] = dash) and (ssn[7] = dash) then
          begin (*fourth and seventh digits are dashes*)
            if not(ssn[i] in ['0'..'9']) then
              begin
                if ok = false then
                  begin
                    ErrorMessage2(i, count, ssn, name, emessage);
                  end
                else if ok = true then
                  begin
                    ErrorMessage1(ok,i,count,error,ssn,name,baddata,emessage);
                  end;
              end;
          end
        else if (ssn[4] in ['0'..'9']) and (ssn[7] in ['0'..'9']) then
          begin (*fourth and seventh digits are not dashes*)
            for i := 8 to 9 do
              begin
                if not(ssn[i] in ['0'..'9']) then
                  begin
                    if ok = false then
                      begin
                        ErrorMessage2(i, count, ssn, name, emessage);
                      end
                    else if ok = true then
                     begin
                       ErrorMessage1(ok,i,count,error,ssn,name,baddata,emessage);
                     end;
                  end;
              end;
            for i := 10 to 11 do
              begin (*digits ten and eleven must be blank*)
                if not(ssn[i] = blank) then
                  begin
                    if ok = false then
                      begin
                        ErrorMessage2(i, count, ssn, name, emessage);
                      end
                    else if ok = true then
                     begin
                       ErrorMessage1(ok,i,count,error,ssn,name,baddata,emessage);
                     end;
                  end;
              end;
          end;
      end;
  if ok = true then
    begin (*SSN has no errors*)
      ok := true; (*set boolean flag to true*)
      writeln(gooddata,count:2,ssn,name);
    end;
end; (* SSNCheck *)


Procedure ProcessLine(var line : row; var ssn : numssn; var name : person);

(*This procedure picks off the values for the ssn and the name.*)

Var
  i, j : integer; (*counters*)

begin (* ProcessLine *)
  j := 1; (*initialize variable*)
  for i:= 1 to 11 do
    begin (*assign values to the cadet SSN*)
      ssn[j] := line[i];
      j := j + 1; (*increment counter*)
    end;
  j := 1; (*initialize variable*)
  for i := 12 to 38 do
    begin (*assign values to the cadet name*)
      name[j] := line[i];
      j := j + 1; (*increment counter*)
    end;
end;  (* ProcessLine *)


Procedure ReadPerson(var filein : text; var count, error : integer;
                     var line : row);

(* This procedure reads the data from the cadet's record one character at a *)
(* time into an intermediate file to be processed. *)

Var
  i : integer; (*counter*)

begin (* ReadPerson *)
  assign(filein, filename);
  reset(filein); (*reset the file*)
  count := 0; (*initialize variable*)
  error := 0; (*initialize variable*)
    while not eof(filein) do
      begin (*read the characters into a file*)
        count := count + 1; (*increment the record count*)
        i := 1; (*initialize variable*)
          while not eoln(filein) do
            begin
              read(filein, line[i]);
              i := i + 1; (*increment the counter*)
            end;
        readln(filein);
        ProcessLine(line, ssn, name);
        SSNCheck(count, error, ssn, name, gooddata, baddata, emessage);
        if (count=150) or (count=300) or (count=450) or (count=600) then
        writeln('PROGRAM IS WORKING - STANDBY':53);
      end;
  close(filein); (*close the file*)
end; (* ReadPerson *)
```

```
begin (* main application - IncompatibleDataTypeCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata, file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata, file2);
  rewrite(baddata);  (*write to a file*)
  assign(emessage,file3);
  rewrite(emessage); (*write to a file*)
  writeln('Data Type Check for SSN':50);
  ReadPerson(filein, count, error, line);
  writeln('There are ':28, error, ' records with SSN errors detected.');
  if error > 0 then
   begin
    writeln('Check files badp2.dat and emesagp2.dat to make corrections.':70);
   end;
  writeln('The number of records read from the input file was ':64,count,'.');
  writeln('This Application Program is Now Finished!':61);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - IncompatibleDataTypeCheck *)
```

```
(*************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Incompatible Data Type Program into one corrected file for future loading*)
(* into the target system.  Execute this program after running Program2.    *)
(*************************************************************************)

Program IncompatibleDataTypeMergeData;
Uses CRT;

Const
  filename1 = 'goodp2.dat';   (*good data file*)
  filename2 = 'badp2.dat';    (*error record file*)
  filename3 = 'corectp2.dat';(*corrected and merged data file*)
  maxcadets = 99;             (*maximum number of cadet records*)
  one       = 1;              (*minimum number of cadet records*)
  blank     = '';             (*blank character*)

Type
  count      = string[2];     (*record count number*)
  numssn     = string[11];    (*cadet SSN*)
  personname = string[27];    (*cadet name*)

  cadet      = record                      (*cadet record*)
                 k          : count;
                 ssn        : numssn;
                 name       : personname;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2      : text;      (*files to be read by the program*)
  master            : text;      (*file to be written by the program*)
  buffer1, buffer2  : cadetrec; (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)

begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  writeln(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  i := i + 1; (*increment counter*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
end; (* GoodData *)
```

```
Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  writeln(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  j := j + 1; (*increment counter*)
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;
  buffer1, buffer2      : cadetrec;

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
        begin
          writeln('Both files are empty':50);
        end
      else if (buffer2[j].k = blank) then
        begin (*all records are in the good data file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k = blank) then
        begin (*all records are in the error record file*)
          BadData(count2, j  buffer2, master, file2);
        end
      else if (buffer1[i].k < buffer2[j].k) then
        begin (*record in buffer1 goes into correct file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k > buffer2[j].k) then
        begin (*record in buffer2 goes into correct file*)
          BadData(count2, j, buffer2, master, file2);
        end
    end;
  count3 := count1 + count2; (*records read should equal value from int file*)
  until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
  writeln('Number of Records Read Equals ':55, count3);
end; (* Merge *)


begin (* main application - IncompatibleDataTypeMergeData *)
  clrscr;
  assign(file1,filename1);
  reset(file1); (*reset the file*)
  assign(file2,filename2);
  reset(file2); (*reset the file*)
  assign(master,filename3);
  rewrite(master); (*write to a file*)
  writeln('Merging Started':48);
  Merge(file1,file2,master);
  writeln('END of APPLICATION':50);
  close(file1);  (*close the file*)
  close(file2);  (*close the file*)
  close(master); (*close the file*)
end. (* main application - IncompatibleDataTypeMergeData *)
```

```
(***************************************************************************)
(* The purpose of this application program is insure that redundant fields *)
(* contain the same data.  This program specifically compares the height   *)
(* and weight values contained in the cadet record against the individual  *)
(* record from the cadet candidate area of the CIDB.  Redundant fields that*)
(* do not contain the same information will cause two files to be generated*)
(* - an error record file and an error message file.  Records that are     *)
(* clean are written to a good data file.  Corrections are made to the     *)
(* error record file, and then it is to be merged with the good data file. *)
(* The corrected good data file is to be stored for future loading into the*)
(* target system.  Use the program RedMerge to merge the good data file and*)
(* the corrected error record file.                                        *)
(***************************************************************************)

Program RedundancyCheck;
Uses CRT;

Const
  filename1  = 'person.dat';    (*cadet record file*)
  filename2  = 'cadetc.dat';    (*individual record file*)
  file1      = 'goodp3.dat';    (*good data file*)
  file2      = 'badp3.dat';     (*error record file*)
  file3      = 'emesagp3.dat';  (*error message file*)
  maxcadets  = 99;              (*maximum number of cadet records*)
  mincadet   = 1;               (*minimum number of cadet records*)

Type
  numssn     = string[11];      (*cadet SSN*)
  personname = string[27];      (*cadet name*)
  inches     = string[2];       (*cadet height*)
  pounds     = string[3];       (*cadet weight*)

  cadet      = record              (*cadet record*)
                 ssn    : numssn;
                 name   : personname;
                 height : inches;
                 weight : pounds;
               end;

  cadetrec1  = array[mincadet..maxcadets] of cadet; (*array of cadet records*)
  cadetrec2  = array[mincadet..maxcadets] of cadet; (*array of cadet records*)

Var
  filein1, filein2        : text;      (*files read by the program*)
  person                  : cadetrec1; (*variable of type cadetrec1*)
  individual              : cadetrec2; (*variable of type cadetrec2*)
  error, count, counter   : integer;   (*counters*)
  gooddata                : text;      (*file to be written by the program*)
  baddata, emessage       : text;      (*file to be written by the program*)
```

```
Procedure ReadPerson(var filein1 : text; var count : integer);

(* This procedure reads in the data from the cadet record into an *)
(* intermediate file to be processed. *)

Var
  i : integer;  (*counter*)

begin (* ReadPerson *)
  assign(filein1, filename1);
  reset(filein1); (*reset the file*)
  i := 1; (*initialize variable*)
  count := 0; (*initialize variable*)
    while not eof(filein1) do
      begin (*read the cadet records into the file*)
        count := count + 1; (*increment the record count*)
        read(filein1,person[i].ssn,person[i].name,person[i].height);
        readln(filein1, person[i].weight);
        i := i + 1; (*increment counter*)
      end;
  close(filein1); (*close the file*)
end; (* ReadPerson *)


Procedure ReadIndividual(var filein2 : text; var counter : integer);

(* This procedure reads in the data from the cadet candidate individual *)
(* record into an intermediate file to be processed. *)

Var
  j : integer;  (*counter*)

begin (* ReadIndividual *)
  assign(filein2, filename2);
  reset(filein2); (*reset the file*)
  j := 1; (*initialize variable*)
  counter := 0; (*initialize variable*)
    while not eof(filein2) do
      begin (*read the cadet candidate records into the file*)
        counter := counter + 1; (*increment the record count*)
        read(filein2,individual[j].ssn,individual[j].name);
        readln(filein2,individual[j].height,individual[j].weight);
        j := j + 1; (*increment counter*)
      end;
  close(filein2); (*close the file*)
end; (* ReadIndividual *)
```

```
Procedure RedundancyChecks(var filein1, filein2 : text;
                           var count, counter, error : integer;
                           var gooddata, baddata, emessage : text);

(* This procedure takes a record from the cadet record file and compares its *)
(* height and weight values to the values contained in the cadet candidate   *)
(* individual record.                                                        *)

Var
  i, j      : integer; (*counters*)
  ok, found : boolean; (*true or false*)

begin (*RedundancyCheck*)
  error := 0; (*initialize variable*)
  for i := 1 to count do
    begin (*loop to compare SSN from cadet record file and find a match in *)
          (*the cadet candidate file, if one exists*)
      j := 1; (*initialize variable*)
      found := false; (*set boolean flag to false*)
      ok := true;     (*set boolean flag to true*)
      while not found do
        begin
          if person[i].ssn = individual[j].ssn then
            begin (*SSNs are the same*)
              found := true; (*set boolean flag to true*)
              if (person[i].height = individual[j].height) and
                 (person[i].weight = individual[j].weight) then
                begin (*heights and weights are the same*)
                  write(gooddata,i:2,person[i].ssn,person[i].name);
                  writeln(gooddata,person[i].height,person[i].weight);
                end;
              if (person[i].height <> individual[j].height) then
                begin (*heights not the same*)
                  ok := false; (*set boolean flag to false*)
                  write(baddata,i:2,person[i].ssn,person[i].name);
                  writeln(baddata,person[i].height,person[i].weight);
                  write(emessage,'Height values are different!  ');
                  writeln(emessage,'Check data for record ',i:2,'.');
                  write(emessage,person[i].ssn,person[i].name);
                  writeln(emessage,person[i].height);
                  write(emessage,'The cadet candidate height value is equal');
                  writeln(emessage,' to ',individual[j].height,'.');
                  writeln(emessage);
                  error := error + 1; (*increment error count*`
                end;
              if (person[i].weight <> individual[j].weight) then
                begin (*weights not the same*)
                  if ok = false then
                    begin (*record contains a previous error*)
```

```
                      write(emessage,'Weight values are different!  ');
                      writeln(emessage,'Check data for record ',i:2,'.');
                      write(emessage,person[i].ssn,person[i].name);
                      writeln(emessage,person[i].weight);
                      write(emessage,'The cadet candidate weight value is ');
                      writeln(emessage,'equal to ',individual[j].weight,'.');
                      writeln(emessage);
                    end
                  else if ok = true then
                    begin (*no previous error exists*)
                      ok := false; (*set boolean flag to false*)
                      write(baddata,i:2,person[i].ssn,person[i].name);
                      writeln(baddata,person[i].height,person[i].weight);
                      write(emessage,'Weight values are different!  ');
                      writeln(emessage,'Check data for record ',i:2,'.');
                      write(emessage,person[i].ssn,person[i].name);
                      writeln(emessage,person[i].weight);
                      write(emessage,'The cadet candidate weight value is ');
                      writeln(emessage,'equal to ',individual[j].weight,'.');
                      writeln(emessage);
                      error := error + 1; (*increment error count*)
                    end;
                end;
              j := j + 1; (*increment counter*)
            end
          else if (person[i].ssn <> individual[j].ssn) then
            begin (*SSNs are not the same*)
              found := false; (*set boolean flag to false*)
              j := j + 1; (*increment counter*)
              if j = counter + 1 then
                begin (*SSNs do not match from either file*)
                  write(baddata,i:2,person[i].ssn,person[i].name);
                  writeln(baddata,person[i].height,person[i].weight);
                  write(emessage,'No match for record ',i,' found in the ');
                  write(emessage,'Cadet Candidate file.  ');
                  writeln(emessage,'Please validate the ');
                  write(emessage,'height and weight for ');
                  write(emessage,person[i].ssn,person[i].name);
                  writeln(emessage,person[i].height,person[i].weight);
                  writeln(emessage);
                  error := error + 1; (*increment error count*)
                  found := true; (*set boolean flag to true*)
                end;
            end;
        end;
    end;
end; (* RedundancyChecks *)
```

```
begin (* main application - RedundancyCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata,file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata,file2);
  rewrite(baddata);  (*write to a file*)
  assign(emessage,file3);
  rewrite(emessage); (*write to a file*)
  writeln('Height and Weight Redundancy Check':57);
  ReadPerson(filein1, count);
  ReadIndividual(filein2, counter);
  RedundancyChecks(filein1, filein2, count, counter, error, gooddata, baddata,
                   emessage);
  writeln('The number of records read from the input file was ':65,count,'.');
  writeln('There are ':30,error,' records with errors detected.');
  if error > 0 then
   begin
    writeln('Check files badp3.dat and emesagp3.dat to make corrections.':71);
   end;
  writeln('This Application Program is Now Finished!':61);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - RedundancyCheck *)
```

```
(*****************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Redundancy Program into one corrected file for future loading into the   *)
(* target system.  Execute this program only after running Program3.        *)
(*****************************************************************************)

Program RedundancyMergeData;
Uses CRT;

Const
  filename1 = 'goodp3.dat';  (*good data file*)
  filename2 = 'badp3.dat';   (*error record file*)
  filename3 = 'corectp3.dat';(*corrected and merged data file*)
  maxcadets = 99;            (*maximum number of cadet records*)
  one       = 1;             (*minimum number of cadet records*)
  blank     = '';            (*blank character*)

Type
  count       = string[2];   (*record count number*)
  numssn      = string[11];  (*cadet SSN*)
  personname  = string[27];  (*cadet name*)
  inches      = string[2];   (*cadet height*)
  pounds      = string[3];   (*cadet weight*)

  cadet       = record                     (*cadet record*)
                  k        : count;
                  ssn      : numssn;
                  name     : personname;
                  height   : inches;
                  weight   : pounds;
                end;

  cadetrec    = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2      : text;     (*files to be read by the program*)
  master            : text;     (*file to be written by the program*)
  buffer1, buffer2  : cadetrec; (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)

begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  write(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  writeln(master,buffer1[i].height,buffer1[i].weight);
  i := i + 1; (*increment counter*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].height);
  readln(file1,buffer1[i].weight);
end; (* GoodData *)


Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  write(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  writeln(master,buffer2[j].height,buffer2[j].weight);
  j := j + 1; (*increment counter*)
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].height);
  readln(file2,buffer2[j].weight);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;  (*counters*)
  buffer1, buffer2             : cadetrec; (*variables of type cadetrec*)

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].height);
  readln(file1,buffer1[i].weight);
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].height);
  readln(file2,buffer2[j].weight);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
        begin (*both files contain no data*)
          writeln('Both files are empty':50);
        end
      else if (buffer2[j].k = blank) then
        begin (*all records are in the good data file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k = blank) then
        begin (*all records are in the error record file*)
          BadData(count2, j, buffer2, master, file2);
        end
      else if (buffer1[i].k < buffer2[j].k) then
        begin (*record in buffer1 goes into correct file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k > buffer2[j].k) then
        begin (*record in buffer2 goes into correct file*)
          BadData(count2, j, buffer2, master, file2);
        end
    end;
  count3 := count1 + count2; (*records should equal value from int. file*)
  until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
  writeln('Number of Records Read Equals ':55, count3);
end; (* Merge *)


begin (* main application - RedundancyMergeData *)
  clrscr; (*clear the screen*)
  assign(file1,filename1);
  reset(file1); (*reset the file*)
  assign(file2,filename2);
  reset(file2); (*reset the file*)
  assign(master,filename3);
  rewrite(master); (*write to a file*)
  writeln('Merging Started':48);
  Merge(file1,file2,master);
  writeln('END of APPLICATION':50);
  close(file1);  (*close the file*)
  close(file2);  (*close the file*)
  close(master); (*close the file*)
end. (* main application - RedundancyMergeData *)
```

```
(*****************************************************************************)
(* The purpose of this application program is to insure that referential    *)
(* integrity holds for the company and regiment that a cadet is assigned.   *)
(* Companies are lettered A through I, while regiments are numbered 1 thru  *)
(* 4, for a total of 36 companies.  Each cadet is to be assigned to a       *)
(* referenced company (null values are allowed if no company is assigned).  *)
(* Each company must be part of the cadet brigade.  Referential integrity   *)
(* violations will cause two files to be generated - an error record file   *)
(* and an error message file.  Records that are clean are written to a good *)
(* data file.  Corrections are to be made to the error record file, and then*)
(* it is to be merged with the good data file.  The corrected good data file*)
(* is to be stored for loading into the target system.  Use the program     *)
(* RIMerge to merge the good data file and the corrected error record file. *)
(*****************************************************************************)

Program ReferentialIntegrityCheck;
Uses CRT;

Const
  filename1  = 'ccc.dat';       (*cadet record file*)
  filename2  = 'company.dat';   (*permanent company file*)
  file1      = 'goodp4.dat';    (*good data file*)
  file2      = 'badp4.dat';     (*error record file*)
  file3      = 'emesagp4.dat';  (*error message file*)
  maxcadets  = 99;              (*maximum number of cadet records*)
  maxcompany = 36;              (*maximum number of cadet companies*)
  one        = 1;               (*minimum number of cadet/company records*)
  blank      = '';              (*blank character*)

Type
  numssn     = string[11];      (*cadet SSN*)
  personname = string[27];      (*cadet name*)
  unitname   = string[2];       (*cadet company*)

  cadet      = record                         (*cadet record*)
                 socsecnum : numssn;
                 name      : personname;
                 comp      : unitname;
               end;

  companyrec = record                         (*company record*)
                 company   : unitname;
               end;

  cadetrec  = array[one..maxcadets] of cadet;       (*array of cadet records*)
  unitrec   = array[one..maxcompany]of companyrec;  (*array of company records*)

Var
  filein1, filein2       : text;      (*files to be read by the program*)
  person                 : cadetrec;  (*variable of type cadetrec*)
  compname               : unitrec;   (*variable of type unitrec*)
  count, counter, error : integer;   (*counters*)
  gooddata              : text;      (*file to be written by the program*)
  baddata, emessage     : text;      (*file to be written by the program*)
  found                 : boolean;   (*true or false*)


Procedure ReadPerson(var filein1 : text; var count : integer);

(* This procedure reads the necessary data from the cadet record into an *)
(* intermediate file to be processed. *)

Var
  i : integer; (*counter*)

begin (* ReadPerson *)
  assign(filein1, filename1);
  reset(filein1); (*reset the file*)
  i := 1; (*initialize variable*)
  count := 0; (*initialize variable*)
    while not eof(filein1) do
      begin (*read the cadet records into the file*)
        count := count + 1; (*increment the record count*)
        readln(filein1, person[i].socsecnum, person[i].name, person[i].comp);
        i := i + 1; (*increment the counter*)
      end;
  close(filein1); (*close the file*)
end; (* ReadPerson *)


Procedure ReadCompany(var filein2 : text; var counter : integer);

(* This procedure reads the necessary data from the permanent company record *)
(* into an intermediate file to be processed. *)

Var
  i : integer; (*counter*)

begin (* ReadCompany*)
  assign(filein2, filename2);
  reset(filein2); (*reset the file*)
  i := 1; (*initialize variable*)
  counter := 0; (*initialize variable*)
    while not eof(filein2) do
      begin (*read company records into the file*)
        counter := counter + 1; (*increment the company count*)
        readln(filein2,compname[i].company);
        i := i + 1; (*increment the counter*)
      end;
  close(filein2); (*close the file*)
end; (* ReadCompany *)
```

```
Procedure RefIntCheck(var count, counter, error : integer; var found : boolean;
                      var filein1, filein2, gooddata, baddata, emessage :text);

(* This procedure checks to insure that the company a cadet is assigned to *)
(* is a valid company. *)

Var
  i, j : integer; (*counters*)

begin (* RefIntCheck *)
  error := 0; (*initialize variable*)
  for i := 1 to count do
    begin (*loop to check cadet company value against permanent company value*)
      j := 1; (*initialize variable*)
      found := false; (*set boolean flag to false*)
      while not found do
        begin
          if (person[i].comp = compname[j].company) or
             (person[i].comp = blank) then
            begin  (*write records with no errors or null values to a file*)
                   (*named goodp4.dat*)
              found := true; (*set boolean flag to true*)
              write(gooddata, i:2, person[i].socsecnum, person[i].name);
              writeln(gooddata, person[i].comp);
              j := j + 1; (*increment counter*)
            end
          else if person[i].comp <> compname[j].company then
            begin (*values are not equal so increment and try next value*)
              found := false; (*set boolean flag to false*)
              j := j + 1; (*increment counter*)
              if j = counter + 1 then
                begin (*write records with errors to a file named badp4.dat*)
                      (*and error messages to a file named emesagp4.dat*)
                  write(baddata, i:2, person[i].socsecnum, person[i].name);
                  writeln(baddata, person[i].comp);
                  writeln(emessage, 'Value for Cadet Company is incorrect!');
                  write(emessage, 'Check data for record ', i:2);
                  write(emessage, person[i].socsecnum:16, person[i].name);
                  writeln(emessage, person[i].comp);
                  writeln(emessage);
                  j := j + 1; (*increment counter*)
                  error := error + 1; (*increment error counter*)
                  found := true; (*set boolean flag to true*)
                end;
            end;
        end;
    end;
end; (* RefIntCheck *)
```

```
begin (* main application - ReferentialIntegrityCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata, file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata, file2);
  rewrite(baddata); (*write to a file*)
  assign(emessage, file3);
  rewrite(emessage); (*write to a file*)
  writeln('Referential Integrity Check For Cadet Company':63);
  ReadPerson(filein1, count);
  ReadCompany(filein2, counter);
  RefIntCheck(count,counter,error,found,filein1,filein2,
              gooddata,baddata,emessage);
  writeln('There are ':36, error, ' errors detected.');
  if error > 0 then
    begin
     writeln('Check files badp4.dat and emesagp4.dat to make corrections.':69);
    end;
  writeln('The number of records read from the input file was ':65,count,'.');
  writeln('This Application Program is Now Finished!':60);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - ReferentialIntegrityCheck *)
```

```
(*************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Referential Integrity Program into one corrected file for future loading *)
(* into the target system.  Execute this program after running Program4.    *)
(*************************************************************************)

Program ReferentialIntegrityMergeData;
Uses CRT;

Const
  filename1 = 'goodp4.dat';  (*good data file*)
  filename2 = 'badp4.dat';   (*error record file*)
  filename3 = 'corectp4.dat';(*corrected and merged data file*)
  maxcadets = 99;            (*maximum number of cadet records*)
  one       = 1;             (*minimum number of cadet records*)
  blank     = '';            (*blank character*)

Type
  count      = string[2];    (*record count number*)
  numssn     = string[11];   (*cadet SSN*)
  personname = string[27];   (*cadet name*)
  unitname   = string[2];    (*cadet company*)

  cadet      = record                         (*cadet record*)
                 k         : count;
                 ssn       : numssn;
                 name      : personname;
                 company   : unitname;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2       : text;     (*files to be read by the program*)
  master             : text;     (*file to be written by the program*)
  buffer1, buffer2   : cadetrec; (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)

begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  write(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  writeln(master,buffer1[i].company); (*write to file*)
  i := i + 1; (*increment counter*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].company);
end; (* GoodData *)
```

```
Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  write(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  writeln(master,buffer2[j].company); (*write to file*)
  j := j + 1; (*increment counter*)
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].company);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;  (*counters*)
  buffer1, buffer2             : cadetrec; (*variable of type cadetrec*)

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name,buffer1[i].company);
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name,buffer2[j].company);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
        begin
          writeln('Both files are empty':50);
        end
      else if (buffer2[j].k = blank) then
        begin (*all records are in the good data file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k = blank) then
        begin (*all records are in the error record file*)
          BadData(count2, j, buffer2, master, file2);
        end
      else if (buffer1[i].k < buffer2[j].k) then
        begin (*record in buffer1 goes into correct file*)
          GoodData(count1, i, buffer1, master, file1);
        end
```

```
          else if (buffer1[i].k > buffer2[j].k) then
            begin (*record in buffer2 goes into correct file*)
              BadData(count2, j, buffer2, master, file2);
            end
        end;
    count3 := count1 + count2; (*records read should equal value from int. file*)
    until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
    writeln('Number of Records Read Equals ':55, count3);
  end; (* Merge *)


  begin (* main application - ReferentialIntegrityMergeData *)
    clrscr; (*clear the screen*)
    assign(file1,filename1);
    reset(file1); (*reset the file*)
    assign(file2,filename2);
    reset(file2); (*reset the file*)
    assign(master,filename3);
    rewrite(master); (*write to a file*)
    writeln('Merging Started':48);
    Merge(file1,file2,master);
    writeln('END of APPLICATION':50);
    close(file1);  (*close the file*)
    close(file2);  (*close the file*)
    close(master); (*close the file*)
  end. (* main application - ReferentialIntegrityMergeData *)
```

```
(**************************************************************************)
(* The purpose of this application program is to insure that entity      *)
(* integrity holds for the primary key field (SSN) of a cadet's individual *)
(* record.  Each SSN must be unique.  This also means that null values for *)
(* the SSN are not allowed.  Entity integrity violations will cause two    *)
(* files to be generated - an error record file and an error message file. *)
(* Records that are clean are written to a good data file.  Corrections are *)
(* to be made to the error record file, and then it is to be merged with the*)
(* good data file.  The corrected good data file is to be stored for loading*)
(* into the target system.  Use the program EntMerge to merge the good data *)
(* file and the corrected error record file.                              *)
(**************************************************************************)

Program EntityIntegrityCheck;
Uses CRT;

Const
  filename1  = 'entity.dat';   (*cadet record file*)
  file1      = 'goodp5.dat';   (*good data file*)
  file2      = 'badp5.dat';    (*error record file*)
  file3      = 'emesagp5.dat'; (*error message file*)
  maxcadets  = 99;             (*maximum number of cadet records*)
  one        = 1;              (*minimum number of cadet records*)
  null       = '           ';  (*blank character*)

Type
  numssn     = string[11];     (*cadet SSN*)
  personname = string[27];     (*cadet name*)

  cadet      = record          (*cadet record*)
                 ssn  : numssn;
                 name : personname;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  filein1          : text;     (*files to be read by the program*)
  person           : cadetrec; (*variable of type cadetrec*)
  count, error     : integer;  (*counters*)
  gooddata         : text;     (*file to be written by the program*)
  baddata, emessage : text;    (*file to be written by the program*)


Procedure ReadPerson(var filein1 : text; var count : integer);

(* This procedure reads the necessary data from the individual record into *)
(* an intermediate file to be processed. *)

Var
  i : integer; (*counter*)
```

```
begin (* ReadPerson *)
  assign(filein1, filename1);
  reset(filein1); (*reset the file*)
  i := 1; (*initialize variable*)
  count := 0; (*initialize variable*)
    while not eof(filein1) do
      begin (*read the individual records into the file*)
        count := count + 1; (*increment the record count*)
        readln(filein1, person[i].ssn, person[i].name);
        i := i + 1; (*increment the counter*)
      end;
  close(filein1); (*close the file*)
end; (* ReadPerson *)


Procedure EntityCheck(var count, error : integer;
                      var filein1, gooddata, baddata, emessage :text);

(* This procedure checks to insure that the cadet's SSN is unique. *)

Var
  i, j : integer; (*counters*)
  ok   : boolean; (*true or false*)

begin (* EntityCheck *)
  error := 0; (*initialize variable*)
  ok := false; (*set boolean flag to false*)
  for i := 1 to count do
    begin (*loop to check the cadet's SSN against the other SSNs in the file*)
      j := i + 1; (*initialize variable*)
      repeat
        if (person[i].ssn = null) and (ok = false) then
          begin (*SSN field is null*)
            ok := true; (*set boolean flag to true*)
            writeln(baddata,i:2,person[i].ssn,person[i].name);
            writeln(emessage,'Cadet SSN is field is null.');
            write(emessage,'Check data for record ',i:2);
            writeln(emessage,' ',person[i].ssn,person[i].name);
            writeln(emessage);
            error := error + 1; (*increment error counter*)
          end;
        if (person[i].ssn = person[j].ssn) then
          begin (*write records with duplicate SSNs to a file named*)
                (*badp5.dat and error messages to a file named emesagp5.dat*)
            if ok = false then
              begin (*first duplicate SSN found*)
                ok := true; (*set boolean flag to true*)
                writeln(baddata,i:2,person[i].ssn,person[i].name);
                writeln(emessage,'Cadet SSN is redundant.');
                write(emessage,'Check data for record ',i:2);
                writeln(emessage,' ',person[i].ssn,person[i].name);
                write(emessage,'with record ',j:2,' ',person[j].ssn);
                writeln(emessage,person[j].name);
                writeln(emessage);
                j := j + 1; (*increment counter*)
                error := error + 1; (*increment error counter*)
              end
            else if ok = true then
              begin (*more than one duplicate SSN has been found*)
                writeln(emessage,'Cadet SSN is redundant.');
                write(emessage,'Check data for record ',i:2);
                writeln(emessage,' ',person[i].ssn,person[i].name);
                write(emessage,'with record ',j:2,' ',person[j].ssn);
                writeln(emessage,person[j].name);
                writeln(emessage);
                j := j + 1; (*increment counter*)
              end;
          end
        else if person[i].ssn <> person[j].ssn then
          begin (*values are not equal so increment and try next value*)
            j := j + 1; (*increment counter*)
            if (j >= count + 1) and (ok = false) then
              begin (*write records with no redundant SSNs to a file named*)
                   (*goodp5.dat*)
                writeln(gooddata,i:2,person[i].ssn,person[i].name);
                ok := false; (*set boolean flag to false*)
              end;
          end;
      until j >= count + 1;
      ok := false; (*set boolean flag to false*)
    end;
end; (* EntityCheck *)


begin (* main application - EntityIntegrityCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata, file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata, file2);
  rewrite(baddata); (*write to a file*)
  assign(emessage, file3);
  rewrite(emessage); (*write to a file*)
  writeln('Entity Integrity Check For Cadet SSN':58);
  ReadPerson(filein1, count);
  EntityCheck(count,error,filein1,gooddata,baddata,emessage);
  writeln('There are ':36, error, ' errors detected.');
  if error > 0 then
    begin
      writeln('Check files badp5.dat and emesagp5.dat for corrections.':68);
    end;
  writeln('The number of records read from the input file was ':65,count,'.');
  writeln('This Application Program is Now Finished!':60);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - EntityIntegrityCheck *)
```

```
(*****************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Entity Integrity Program into one corrected file for future loading into *)
(* the target system.  Execute this program only after running Program5.    *)
(*****************************************************************************)

Program EntityIntegrityMergeData;
Uses CRT;

Const
  filename1 = 'goodp5.dat';   (*good data file*)
  filename2 = 'badp5.dat';    (*error record file*)
  filename3 = 'corectp5.dat';(*corrected and merged data file*)
  maxcadets = 99;             (*maximum number of cadet records*)
  one       = 1;              (*minimum number of cadet records*)
  blank     = '';             (*blank character*)

Type
  count      = string[2];     (*record count number*)
  numssn     = string[11];    (*cadet SSN*)
  personname = string[27];    (*cadet name*)

  cadet      = record                      (*cadet record*)
                 k          : count;
                 ssn        : numssn;
                 name       : personname;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2       : text;      (*files to be read by the program*)
  master             : text;      (*file to be written by the program*)
  buffer1, buffer2   : cadetrec;  (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)

begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  wri eln(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  i := i + 1; (*increment counter*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
end; (* GoodData *)
```

```
Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  writeln(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  j := j + 1; (*increment counter*)
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;  (*counters*)
  buffer1, buffer2             : cadetrec; (*variables of type cadetrec*)

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  readln(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
  readln(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
        begin (*both files contain no data*)
          writeln('Both files are empty':50);
        end
      else if (buffer2[j].k = blank) then
        begin (*all records are in the good data file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k = blank) then
        begin (*all records are in the error record file*)
          BadData(count2, j, buffer2, master, file2);
        end
      else if (buffer1[i].k < buffer2[j].k) then
        begin (*record in buffer1 goes into correct file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k > buffer2[j].k) then
```

```
        begin (*record in buffer2 goes into correct file*)
          BadData(count2, j, buffer2, master, file2);
        end
    end;
  count3 := count1 + count2; (*records should equal value from int. file*)
  until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
  writeln('Number of Records Read Equals ':55, count3);
end; (* Merge *)


begin (* main application - EntityIntegrityMergeData *)
  clrscr; (*clear the screen*)
  assign(file1,filename1);
  reset(file1); (*reset the file*)
  assign(file2,filename2);
  reset(file2); (*reset the file*)
  assign(master,filename3);
  rewrite(master); (*write to a file*)
  writeln('Merging Started':48);
  Merge(file1,file2,master);
  writeln('END of APPLICATION':50);
  close(file1);  (*close the file*)
  close(file2);  (*close the file*)
  close(master); (*close the file*)
end. (* main application - EntityIntegrityMergeData *)
```

```
(****************************************************************************)
(* The purpose of this application program is to ensure that the logical   *)
(* implication holds for the cadet's high school class ranking being less  *)
(* than the number in the high school graduating class.  Logical inconsis- *)
(* tency violations will cause two files to be generated - an error record *)
(* file and an error message file.  Records that are clean are written to a *)
(* good data file.  Corrections are to be made to the error record file, and*)
(* then it is to be merged with the good data file.  The corrected good data*)
(* file is to be stored for loading into the target system.  Use the program*)
(* LogMerge to merge the good data file and the corrected error record file.*)
(****************************************************************************)

Program LogicalInconsistencyCheck;
Uses CRT;

Const
  filename1  = 'logic.dat';     (*cadet record file*)
  file1      = 'goodp6.dat';    (*good data file*)
  file2      = 'badp6.dat';     (*error record file*)
  file3      = 'emesagp6.dat';  (*error message file*)
  maxcadets  = 99;              (*maximum number of cadet records*)
  one        = 1;               (*minimum number of cadet records*)
  null       = '';              (*blank character*)

Type
  numssn     = string[11];      (*cadet SSN*)
  personname = string[27];      (*cadet name*)
  number     = integer;         (*high school rank/number in class*)

  cadet      = record           (*cadet record*)
                 ssn    : numssn;
                 name   : personname;
                 hsrank : number;
                 hsnum  : number;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  filein1            : text;     (*files to be read by the program*)
  person             : cadetrec; (*variable of type cadetrec*)
  count, error       : integer;  (*counters*)
  gooddata           : text;     (*file to be written by the program*)
  baddata, emessage  : text;     (*file to be written by the program*)


Procedure ReadPerson(var filein1 : text; var count : integer);

(* This procedure reads the necessary data from the individual record into *)
(* an intermediate file to be processed. *)
```

```
Var
  i : integer; (*counter*)

begin (* ReadPerson *)
  assign(filein1, filename1);
  reset(filein1); (*reset the file*)
  i := 1; (*initialize variable*)
  count := 0; (*initialize variable*)
    while not eof(filein1) do
      begin (*read the individual records into the file*)
        count := count + 1; (*increment the record count*)
        read(filein1,person[i].ssn,person[i].name,person[i].hsrank);
        readln(filein1,person[i].hsnum);
        i := i + 1; (*increment the counter*)
      end;
  close(filein1); (*close the file*)
end; (* ReadPerson *)


Procedure LogicCheck(var count, error : integer;
                     var filein1, gooddata, baddata, emessage :text);

(* This procedure checks to insure that the cadet's high school ranking is *)
(*less than the number in their high school class. *)

Var
  i  : integer; (*counters*)

begin (* LogicCheck *)
  error := 0; (*initialize variable*)
  for i := 1 to count do
    begin (*loop to check the cadet's hs rank against the hs number in class*)
      if person[i].hsrank <= person[i].hsnum then
        begin (*rank is less than number in class - write records to a *)
              (*file named goodp6.dat*)
          write(gooddata,i:2,person[i].ssn,person[i].name,person[i].hsrank:4);
          writeln(gooddata,' ',person[i].hsnum:4);
        end
      else if person[i].hsrank > person[i].hsnum then
        begin (*write records with hs rank greater than number in hs class*)
              (*to a file named badp6.dat and error messages to a file    *)
              (*named emesagp6.dat*)
          write(baddata,i:2,person[i].ssn,person[i].name);
          writeln(baddata,person[i].hsrank:4,' ',person[i].hsnum:4);
          write(emessage,'Cadet HS rank is greater than the number in ');
          writeln(emessage,'the HS graduating class.');
          write(emessage,'Check data for record ',i:2);
          write(emessage,' ',person[i].ssn,person[i].name);
          writeln(emessage,person[i].hsrank:4,' ',person[i].hsnum:4);
          writeln(emessage);
          error := error + 1; (*increment error counter*)
        end;
    end;
end; (* LogicCheck *)


begin (* main application - LogicalInconsistencyCheck *)
  clrscr; (*clear the screen*)
  assign(gooddata, file1);
  rewrite(gooddata); (*write to a file*)
  assign(baddata, file2);
  rewrite(baddata); (*write to a f'le*)
  assign(emessage, file3);
  rewrite(emessage); (*write to a file*)
  writeln('Logical Inconsistency Check For Cadet High School Rank':67);
  ReadPerson(filein1, count);
  LogicCheck(count,error,filein1,gooddata,baddata,emessage);
  writeln('There are ':36, error, ' errors detected.');
  if error > 0 then
    begin
      writeln('Check files badp6.dat and emesagp6.dat for corrections.':68);
    end;
  writeln('The number of records read from the input file was ':65,count,'.');
  writeln('This Application Program is Now Finished!':60);
  close(gooddata); (*close the file*)
  close(baddata);  (*close the file*)
  close(emessage); (*close the file*)
end. (* main application - LogicalInconsistencyCheck *)
```

```
(****************************************************************************)
(* The purpose of this program is to merge the two data files created by the*)
(* Logical Inconsistency Program into one corrected file for future loading *)
(* into the target system. Execute this program only after running Program6.*)
(****************************************************************************)

Program LogicalInconsistencyMergeData;
Uses CRT;

Const
  filename1 = 'goodp6.dat';   (*good data file*)
  filename2 = 'badp6.dat';    (*error record file*)
  filename3 = 'corectp6.dat';(*corrected and merged data file*)
  maxcadets = 99;             (*maximum number of cadet records*)
  one       = 1;              (*minimum number of cadet records*)
  blank     = '';             (*blank character*)

Type
  count      = string[2];     (*record count number*)
  numssn     = string[11];    (*cadet SSN*)
  personname = string[27];    (*cadet name*)
  number     = integer;       (*hs ranking/number in hs class*)

  cadet      = record                   (*cadet record*)
                 k      : count;
                 ssn    : numssn;
                 name   : personname;
                 hsrank : number;
                 hsnum  : number;
               end;

  cadetrec   = array[one..maxcadets] of cadet; (*array of cadet records*)

Var
  file1, file2       : text;     (*files to be read by the program*)
  master             : text;     (*file to be written by the program*)
  buffer1, buffer2   : cadetrec; (*variable of type cadetrec*)


Procedure GoodData(var count1, i : integer; var buffer1 : cadetrec;
                   var master, file1 : text);

(* This procedure writes the records from the good data file to the *)
(* corrected data file. *)

begin (* GoodData *)
  count1 := count1 + 1; (*increment the record count*)
  write(master,buffer1[i].k,' ',buffer1[i].ssn,buffer1[i].name);
  writeln(master,buffer1[i].hsrank:4,' ',buffer1[i].hsnum:4);
  i := i + 1; (*increment counter*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
  readln(file1,buffer1[i].hsrank,buffer1[i].hsnum);
end; (* GoodData *)


Procedure BadData(var count2, j : integer; var buffer2 : cadetrec;
                  var master, file2 : text);

(* This procedure writes the records from the corrected error record file *)
(* to the corrected data file. *)

begin (* BadData *)
  count2 := count2 + 1; (*increment the record count*)
  write(master,buffer2[j].k,' ',buffer2[j].ssn,buffer2[j].name);
  writeln(master,buffer2[j].hsrank:4,' ',buffer2[j].hsnum:4);
  j := j + 1; (*increment counter*)
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
  readln(file2,buffer2[j].hsrank,buffer2[j].hsnum);
end; (* BadData *)


Procedure Merge(var file1, file2, master : text);

(* This procedure merges the good data file and the corrected error record *)
(* file into a corrected good data file in the correct numeric sequence.   *)

Var
  i, j, count1, count2, count3 : integer;  (*counters*)
  buffer1, buffer2             : cadetrec; (*variables of type cadetrec*)

begin (* Merge *)
  count1 := 0; (*initialize variable*)
  count2 := 0; (*initialize variable*)
  count3 := 0; (*initialize variable*)
  i := 1; (*initialize variable*)
  j := 1; (*initialize variable*)
  read(file1,buffer1[i].k,buffer1[i].ssn,buffer1[i].name);
  readln(file1,buffer1[i].hsrank,buffer1[i].hsnum);
  read(file2,buffer2[j].k,buffer2[j].ssn,buffer2[j].name);
  readln(file2,buffer2[j].hsrank,buffer2[j].hsnum);
  repeat
    begin (*loop to merge two files*)
      if (buffer1[i].k = blank) and (buffer2[j].k = blank) then
        begin (*both files contain no data*)
          writeln('Both files are empty':50);
        end
      else if (buffer2[j].k = blank) then
        begin (*all records are in the good data file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k = blank) then
        begin (*all records are in the error record file*)
          BadData(count2, j, buffer2, master, file2);
        end
      else if (buffer1[i].k < buffer2[j].k) then
        begin (*record in buffer1 goes into correct file*)
          GoodData(count1, i, buffer1, master, file1);
        end
      else if (buffer1[i].k > buffer2[j].k) then
        begin (*record in buffer2 goes into correct file*)
          BadData(count2, j, buffer2, master, file2);
        end
    end;
  count3 := count1 + count2; (*records should equal value from int. file*)
  until (buffer1[i].k = blank) and (buffer2[j].k = blank); (*both files empty*)
  writeln('Number of Records Read Equals ':55, count3);
end; (* Merge *)


begin (* main application - LogicalInconsistencyMergeData *)
  clrscr; (*clear the screen*)
  assign(file1,filename1);
  reset(file1); (*reset the file*)
  assign(file2,filename2);
  reset(file2); (*reset the file*)
  assign(master,filename3);
  rewrite(master); (*write to a file*)
  writeln('Merging Started':48);
  Merge(file1,file2,master);
  writeln('END of APPLICATION':50);
  close(file1);  (*close the file*)
  close(file2);  (*close the file*)
  close(master); (*close the file*)
end. (* main application - LogicalInconsistencyMergeData *)
```

APPENDIX C

SPECIFIC METHOD FILES

The output that follows was generated by running the application programs from Appendix B on fictitious data files. The made-up data files were used for two reasons: First, to ensure the programs worked properly, and second, to provide representative examples of errors the programs could detect. The programs were also run on actual USMA data files. The results from these runs can be found in Chapter Seven.

A. Out-of-Range Values

Intermediate File
```
394529826  HENDRICKSON MARK R         73185M681103
310708602  O'KEEFE KATHLEEN M         67090F680515
123456789  PONGSUWAN WUTTIPONG O      64145M690325
987654321  HENDRICKSON BETTE J        65142F710221
555121234  HENDRICKSON ROBERT E       71245M690713
415981243  CONNER RYAN C              63109M730627
```

Good Data File
```
 3 123456789  PONGSUWAN WUTTIPONG O      64145M690325
 5 555121234  HENDRICKSON ROBERT E       71245M690713
```

Error Record File
```
 1 394529826  HENDRICKSON MARK R         83185M681103
 2 310708602  O'KEEFE KATHLEEN M         67090F580515
 4 987654321  HENDRICKSON BETTE J        65142 710221
 6 415981243  CONNER RYAN C              63 99M830627
```

Error Message File
```
Height value out-of-range!  Check height for record 1.
394529826  HENDRICKSON MARK R         83

Birthdate value out-of-range!  Check birthdate for record 2.
310708602  O'KEEFE KATHLEEN M         580515

Sex value out-of-range!  Check sex for record 4.
987654321  HENDRICKSON BETTE J
```

```
Weight value out-of-range!  Check weight for record 6.
415981243  CONNER RYAN C              99

Birthdate value out-of-range!  Check birthdate for record 6.
415981243  CONNER RYAN C             830627
```

<u>Corrected Good Data File</u>

```
 1 394529826  HENDRICKSON MARK R           73185M681103
 2 310708602  O'KEEFE KATHLEEN M           67090F680515
 3 123456789  PONGSUWAN WUTTIPONG O        64145M690325
 4 987654321  HENDRICKSON BETTE J          65142F710221
 5 555121234  HENDRICKSON ROBERT E         71245M690713
 6 415981243  CONNER RYAN C                63109M830627
```

B. Incompatible Data Types

<u>Intermediate File</u>

```
A90408656  RITTER JACK L
2B9178539  JOHNSON JERRY J
59C503620  BOONE DANIEL A
608D17163  MARTIN FRANK P
1646 9133  DUDEK ROBERT H
46333E439  WIEMER JOHN S
080-50-70ZZCOWBOY CLINT E
343-6W-0455DAVENPORT ALLEN M
480-13-9999NEWMAN TIMOTHY C
455113460  MERRITT RICHARD R
437881234  ANDERSEN JAMES J
 50355189  NORMAN BUDDY L
```

<u>Good Data File</u>

```
 9 480-13-9999NEWMAN TIMOTHY C
10 455113460  MERRITT RICHARD R
11 437881234  ANDERSEN JAMES J
```

<u>Error Record File</u>

```
 1 A90408656  RITTER JACK L
 2 2B9178539  JOHNSON JERRY J
 3 59C503620  BOONE DANIEL A
 4 608D17163  MARTIN FRANK P
 5 1646 9133  DUDEK ROBERT H
 6 46333E439  WIEMER JOHN S
 7 080-50-70ZZCOWBOY CLINT E
 8 343-6W-0455DAVENPORT ALLEN M
12  50355189  NORMAN BUDDY L
```

<u>Error Message File</u>

```
Error in SSN - position 1.  Check SSN for record 1
A90408656                RITTER JACK L
```

```
Error in SSN - position 2.  Check SSN for record 2
2B9178539                JOHNSON JERRY J

Error in SSN - position 3.  Check SSN for record 3
59C503620                BOONE DANIEL A

Error in SSN - position 4.  Check SSN for record 4
608D17163                MARTIN FRANK P

Error in SSN - position 5.  Check SSN for record 5
1646 9133                DUDEK ROBERT H

Error in SSN - position 6.  Check SSN for record 6
46333E439                WIEMER JOHN S

Error in SSN - position 10.  Check SSN for record 7
080-50-70ZZ              COWBOY CLINT E

Error in SSN - position 11.  Check SSN for record 7
080-50-70ZZ              COWBOY CLINT E

Error in SSN - position 6.  Check SSN for record 8
343-6W-0455              DAVENPORT ALLEN M

Error in SSN - position 1.  Check SSN for record 12
 50355189                NORMAN BUDDY L
```

Corrected Good Data File

```
 1 290408656   RITTER JACK L
 2 239178539   JOHNSON JERRY J
 3 599503620   BOONE DANIEL A
 4 608217163   MARTIN FRANK P
 5 164699133   DUDEK ROBERT H
 6 463330439   WIEMER JOHN S
 7 080-50-7099COWBOY CLINT E
 8 343-65-0455DAVENPORT ALLEN M
 9 480-13-9999NEWMAN TIMOTHY C
10 455113460   MERRITT RICHARD R
11 437881234   ANDERSEN JAMES J
12 150355189   NORMAN BUDDY L
```

C. Redundancies

Intermediate File

```
394529826  HENDRICKSON MARK R          73181
310708602  HENDRICKSON KATHLEEN O      67110
454632123  O'KEEFE SUSAN L             65135
123456789  PONGSUWAN WUTTIPONG O       65150
```

Good Data File

```
 2 310708602  HENDRICKSON KATHLEEN O      67110
```

Error Record File

```
 1 394529826  HENDRICKSON MARK R          73181
 3 454632123  O'KEEFE SUSAN L             65135
 4 123456789  PONGSUWAN WUTTIPONG O       65150
```

Error Message File

```
Weight values are different!  Check data for record  1.
394529826  HENDRICKSON MARK R          181
The cadet candidate weight value is equal to 180.

No match for record 3 found in the Cadet Candidate file.
Please validate the height and weight for 454632123
O'KEEFE SUSAN L              65135

No match for record 4 found in the Cadet Candidate file.
Please validate the height and weight for 123456789
PONGSUWAN WUTTIPONG O        65150
```

Corrected Good Data File

```
 1 394529826  HENDRICKSON MARK R          73180
 2 310708602  HENDRICKSON KATHLEEN O      67110
 3 454632123  O'KEEFE SUSAN L             65125
 4 123456789  PONGSUWAN WUTTIPONG O       65145
```

D. Referential Integrity

Intermediate File

```
394529826  HENDRICKSON MARK R          I4
312455432  O'KEEFE GLORIA S            GG
310708602  O'KEEFE KATHLEEN M          A3
123456789  PONGSUWAN WUTTIPONG O       H1
987654321  CONNER RYAN C               E8
555001212  BADAGNANI DAVID J           C2
410453178  O'KEEFE LOUIS J             B4
333333333  HENDRICKSON R E
666666666  CLAUS SANTA J               Z4
787878787  MCLEAN WILLIAM T            3B
```

Good Data File

```
 1 394529826  HENDRICKSON MARK R          I4
 3 310708602  O'KEEFE KATHLEEN M          A3
 4 123456789  PONGSUWAN WUTTIPONG O       H1
 6 555001212  BADAGNANI DAVID J           C2
 7 410453178  O'KEEFE LOUIS J             B4
 8 333333333  HENDRICKSON R E
```

Error Record File

```
 2 312455432  O'KEEFE GLORIA S           GG
 5 987654321  CONNER RYAN C              E8
 9 666666666  CLAUS SANTA J              Z4
10 787878787  MCLEAN WILLIAM T           3B
```

Error Message File

```
Value for Cadet Company is incorrect!
Check data for record  2 312455432  O'KEEFE GLORIA S          GG

Value for Cadet Company is incorrect!
Check data for record  5 987654321  CONNER RYAN C             E8

Value for Cadet Company is incorrect!
Check data for record  9 666666666  CLAUS SANTA J             Z4

Value for Cadet Company is incorrect!
Check data for record 10 787878787  MCLEAN WILLIAM T          3B
```

Corrected Good Data File

```
 1 394529826  HENDRICKSON MARK R         I4
 2 312455432  O'KEEFE GLORIA S           G1
 3 310708602  O'KEEFE KATHLEEN M         A3
 4 123456789  PONGSUWAN WUTTIPONG O      H1
 5 987654321  CONNER RYAN C              E2
 6 555001212  BADAGNANI DAVID J          C2
 7 410453178  O'KEEFE LOUIS J            B4
 8 333333333  HENDRICKSON R E
 9 666666666  CLAUS SANTA J              A4
10 787878787  MCLEAN WILLIAM T           B3
```

E. Entity Integrity

Intermediate File

```
394529826  HENDRICKSON MARK R
           O'KEEFE GLORIA S
310708602  O'KEEFE KATHLEEN M
123456789  PONGSUWAN WUTTIPONG O
987654321  CONNER RYAN C
555001212  BADAGNANI DAVID J
410453178  O'KEEFE LOUIS J
394529826  HENDRICKSON R E
111111111  CLAUS SANTA J
787878787  MCLEAN WILLIAM T
394529827  HENDRICKSON TODD R
```

Good Data File
```
 3 310708602  O'KEEFE KATHLEEN M
 4 123456789  PONGSUWAN WUTTIPONG O
 5 987654321  CONNER RYAN C
 6 555001212  BADAGNANI DAVID J
 7 410453178  O'KEEFE LOUIS J
 8 394529826  HENDRICKSON R E
 9 111111111  CLAUS SANTA J
10 787878787  MCLEAN WILLIAM T
11 394529827  HENDRICKSON TODD R
```

Error Record File
```
 1 394529826  HENDRICKSON MARK R
 2            O'KEEFE GLORIA S
```

Error Message File
```
Cadet SSN is redundant.
Check data for record  1 394529826  HENDRICKSON MARK R
with record  8 394529826  HENDRICKSON R E

Cadet SSN is field is null.
Check data for record  2            O'KEEFE GLORIA S
```

Corrected Good Data File
```
 1 394529825  HENDRICKSON MARK R
 2 301556789  O'KEEFE GLORIA S
 3 310708602  O'KEEFE KATHLEEN M
 4 123456789  PONGSUWAN WUTTIPONG O
 5 987654321  CONNER RYAN C
 6 555001212  BADAGNANI DAVID J
 7 410453178  O'KEEFE LOUIS J
 8 394529826  HENDRICKSON R E
 9 111111111  CLAUS SANTA J
10 787878787  MCLEAN WILLIAM T
11 394529827  HENDRICKSON TODD R
```

F. Logical Inconsistencies

Intermediate File
```
394529826  HENDRICKSON MARK R        26    350
310991234  O'KEEFE GLORIA S          1     47
310708602  O'KEEFE KATHLEEN M        3     400
123456789  PONGSUWAN WUTTIPONG O     1110  300
987654321  CONNER RYAN C             41    98
555001212  BADAGNANI DAVID J         67    135
410453178  O'KEEFE LOUIS J           8     27
394529829  HENDRICKSON ROBERT E      120   13
111111111  CLAUS SANTA J             1     1
787878787  MCLEAN WILLIAM T          511   575
394529827  HENDRICKSON TODD R        150   295
```

Good Data File

```
 1 394529826  HENDRICKSON MARK R           26  350
 2 310991234  O'KEEFE GLORIA S              1   47
 3 310708602  O'KEEFE KATHLEEN M            3  400
 5 987654321  CONNER RYAN C                41   98
 6 555001212  BADAGNANI DAVID J            67  135
 7 410453178  O'KEEFE LOUIS J               8   27
 9 111111111  CLAUS SANTA J                 1    1
10 787878787  MCLEAN WILLIAM T            511  575
11 394529827  HENDRICKSON TODD R          150  295
```

Error Record File

```
 4 123456789  PONGSUWAN WUTTIPONG O      1110  300
 8 394529829  HENDRICKSON ROBERT E        120   13
```

Error Message File

```
Cadet HS rank is greater than the number in the HS graduating
class.  Check data for record  4
123456789  PONGSUWAN WUTTIPONG O      1110  300

Cadet HS rank is greater than the number in the HS graduating
class.  Check data for record  8
394529829  HENDRICKSON ROBERT E        120   13
```

Corrected Good Data File

```
 1 394529826  HENDRICKSON MARK R           26  350
 2 310991234  O'KEEFE GLORIA S              1   47
 3 310708602  O'KEEFE KATHLEEN M            3  400
 4 123456789  PONGSUWAN WUTTIPONG O       300 1300
 5 987654321  CONNER RYAN C                41   98
 6 555001212  BADAGNANI DAVID J            67  135
 7 410453178  O'KEEFE LOUIS J               8   27
 8 394529829  HENDRICKSON ROBERT E         13  120
 9 111111111  CLAUS SANTA J                 1    1
10 787878787  MCLEAN WILLIAM T            511  575
11 394529827  HENDRICKSON TODD R          150  295
```

LIST OF REFERENCES

1. Elmasri, R., and Navathe, S. B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

2. Guilmette, D. J., and Wilson, G. P., *The West Point Database Conversion Project From a Network to a Relational DBMS*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.

3. Date, C. J., *An Introduction to Database Systems, Volume II*, Addison-Wesley Publishing Company, 1983.

4. Date, C. J., *An Introduction to Database Systems, Third Edition*, Addison-Wesley Publishing Company, 1981.

5. Department of the Army, United States Military Academy Regulation 25-5, *Information Management Systems*, West Point, New York, 11 August 1989.

6. Department of the Army, United States Military Academy Cadet Information Database Dictionary, West Point, New York, 18 December 1989.

7. Department of the Army, United States Military Academy Academic Program AY 1989-1990, West Point, New York, December 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Code CS Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Dr. Vincent Y. Lum Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	5
5.	Dr. C. Thomas Wu Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
6.	CPT Mark R. Hendrickson P.O. Box 432 Gallatin, Tennessee 37066	5
7.	Mr. Robert W. Nelson United States Military Academy ATTN: MAIM-CD West Point, New York 10996-2001	5
8.	CPT Daniel J. Guilmette Box 2800 Chapel Road Bennington, Vermont 05201	1

9.	CPT Georgette P. Wilson 98-1881-D Kaahumanu Street Aiea, Hawaii 96701	1