odb/Tools Project Report

by

Fred L. Drake, Jr.

Project report submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

©Fred L. Drake, Jr. and VPI & SU 1995

APPROVED:

____

Lenwood S. Heath, Chairman

______________________________ ______________________________

James A. Arthur Edward A. Fox

August, 1995

Blacksburg, Virginia

odb/Tools Project Report

by

Fred L. Drake, Jr.

Committee Chairman: Lenwood S. Heath

Computer Science

(ABSTRACT)

odb/Tools is a suite of object classes providing scriptable access to the object database facility of Project Envision. It implements an interface to the base ODB library for the Python embeddable language using both a C-language module linked to the host runtime environment and several native Python modules to provide the high level access to the facility.

ACKNOWLEDGEMENTS

I would like to thank my wife, Cathy, for supporting me through the course of executing this project, putting up with the long hours I spent away from our wonderful relationship, and for taking care of our son, William, while I was working. I would like to thank my parents for their support for my continued education. I would also like to thank my advisor, Dr. Heath, for supporting my work on such a tight schedule, and his many contributions to the work described. I would like to thank Dr. Fox for providing thought-provoking advice as well as reading the documents I've written in conjunction with this project. Dr. Arthur I thank for serving on my committee. I would also like to thank Guillermo Averboch for providing some of the foundations for this project, and answering many questions regarding ODB. I would also like to thank Bill Wake for answering questions and proofreading, and Scott Guyer for testing *odb*/Tools and the documentation.

TABLE OF CONTENTS

1 Introduction and Overview **1**
1.1 Motivation 1
1.2 Organization of the *odb*/Tools 3
1.2.1 Examining the Database Structure 4
1.2.2 Accessing Database Entities 5
1.2.3 Change Management 6
1.2.4 Iterating Over the Database 7
1.2.5 Selecting Objects 8
1.3 How the *odb*/Tools are Used 9
1.4 Selection of Language 11
1.5 Efficiency Notes 12
1.6 Python Reference Material 13
1.7 Overview of Modules 14
1.8 Documentation Conventions and Notes 17
1.9 Organization of the Report 18

2 The ODB Object Database **20**
2.1 Organization of Stored Information 20
2.1.1 Field Definitions and Constraints 21
2.2 Indexing Facilities 22
2.3 File Organization on the Persistent Store 23

3 Low-level Database Interface **24**
3.1 Built-in Module `odb` 24
3.2 `odb` Functions 25
3.3 `odb` Public Data and Exceptions 26
3.4 `OdbDatabase` Objects 29
3.4.1 `OdbDatabase` Methods 29
3.4.2 `OdbDatabase` Member Data 35
3.5 `OdbInstance` Objects 37
3.5.1 `OdbInstance` Methods 38
3.5.2 `OdbInstance` Member Data 38

4 The Data Dictionary **40**
4.1 Overview of the Data Dictionary Facilities 41
4.2 Introduction to the Classes 43

4.3 Common Methods 43
4.4 Odb module `DataDictionary` 44
4.4.1 `DataDictionary` Methods 45
4.5 Odb module `ClassDefn` 46
4.5.1 `ClassDefn` Methods 47
4.6 Odb module `FieldDefn` 49
4.6.1 `FieldDefn` Methods 49

5 Accessing the Database **52**
5.1 Odb module `Database` 52
5.1.1 `Database` Methods 53
5.1.2 `Database` Member Data 56

6 The Object Protocol **59**
6.1 Object Protocol Methods 60
6.1.1 Supporting Objects 62
6.2 Field Protocol Methods 63
6.2.1 Supporting Objects 65
6.2.2 Iterating Over Field Values 66

7 Iteration and Traversal Mechanisms **69**
7.1 `Iterators` Module 69
7.2 Sequential Iteration 70
7.3 Depth-first Traversal 72
7.4 Using Iterators Efficiently 73

8 Selection and Traversal Predicates **75**
8.1 Odb module `Predicates` 76
8.1.1 Selection Predicate Classes 76
8.1.2 Compositional Predicate Classes 78
8.2 Odb module `Traversals` 79
8.2.1 Compositional Predicates and Traversal 80
8.3 Implementing New Predicates 80
8.3.1 Functions as Predicates 80
8.3.2 Classes as Predicate Generators 81
8.3.3 Traversal Predicates 86
8.3.4 Invoking Predicates Directly 86

A Installing *odb*/Tools **88**
A.1 Obtaining Python 88
A.2 Obtaining and Installing GDBM 89
A.3 Building and Installing *odb*/Tools 90

A.3.1 Compiler Selection 91
A.3.2 Using the Readline Library 91
A.3.3 Setting the Installation Prefix 91
A.3.4 Running `./configure` 92
A.3.5 Configuring the Built-in Modules 92
A.3.6 Building the Interpreter 93
A.4 Testing the Installation 93
A.5 Version Information 94

B Running *odb*/Tools Scripts **95**

Chapter 1

Introduction and Overview

This chapter discusses the ideas and motivation behind the *odb*/Tools package and provides a brief introduction to its capabilities. Detailed reference information appears in subsequent chapters. *odb*/Tools arose from the need to perform manipulation and reporting on the meta-information related to the document database of Project Envision. The existing low-level interface to the ODB database package made developing small applications for handling the database time consuming, tedious, and error prone. The new interface is designed to encourage the creation of tools as they are needed and to allow prototyping of larger applications should the need arise. The position of *odb*/Tools within the Envision framework is diagrammed in Figure 1.1.

The project included the design of an interface to the ODB library for the Python programming language to allow a rapid development approach to be taken in creating tools used to manipulate and report on data stored in the database populated by DELTO [Ave95] and other tools used to import documents into the Project Envision library. Implementing the interface involved writing a Python module in C to provide access to the existing database functions and creating several modules written in Python to provide the high level, object-oriented interface for application programmers.

1.1 Motivation

Project Envision, an initiative to create a digital library of the computer science literature at Virginia Polytechnic Institute and State University, includes many components tied together in part by a database holding document meta-data [HHN+95]. This meta-data

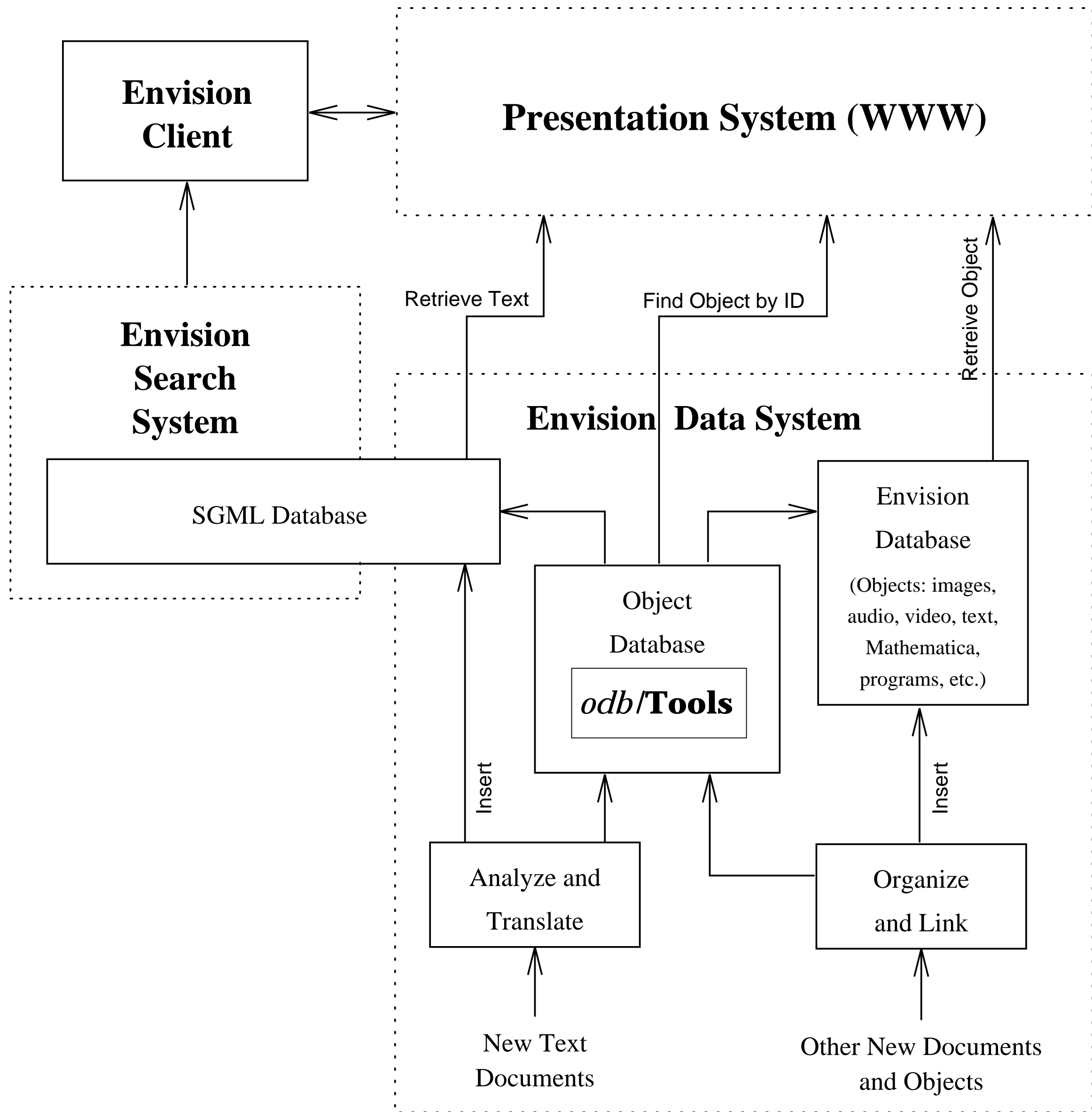

Figure 1.1: *odb*/Tools in Context

covers bibliographic and inter-document relationship information, forming a complex and potentially dynamic information graph with complex semantics. The meta-data is stored in an *object database* implemented using the ODB database library created by Guillermo Averboch. The DELTO document processor, also by Averboch, populates this database with information extracted from documents being prepared for inclusion in the document database [Ave95]. In the Envision database, the relationship information includes links between *Computing Review* categories and documents which have been assigned to each category, people and the organizations with which they are associated, authors and the documents they have written, editors and the documents they have edited, documents and the documents they cite, and keywords and the documents keyed to them.

Minimal tools exist to manipulate entries in the database, perform indexing operations, and allow browsing the database both on a character terminal and World Wide Web interface. These tools are all implemented over a low-level, C language interface provided by the original ODB library.

One cause of the poor availability of general and robust tools for managing the Envision library of documents has been the original interface to the ODB database library. Though the C implementation is fairly efficient in many ways, writing applications to this interface is quite tedious and error-prone. The primary goal of *odb*/Tools is to provide an interface more easily used for rapid application development. To achieve this goal, *odb*/Tools provides a layered, object-oriented wrapper to the underlying ODB library, providing higher-level abstractions, removing the need for explicit memory management in most cases, and encouraging the use of rapid prototyping techniques for initial application design.

1.2 Organization of the *odb*/Tools

The *odb*/Tools approach the creation of the interface over the base ODB library using a layered structure. The only module to actually come in contact with the underlying ODB library is the Python module named `odb`, implemented in C. This module provides a

thin object-oriented layer on top of the C functions in the library. It exports two types of objects, one representing an entire database and the other representing a single database entity. Functions are exported to open a database and provide other information from the ODB library.

Above this a set of high level objects defined in Python provides the application-level view of the database. The primary objects in this set are the `DataDictionary`, representing the class structure defined in the database, and the `Database`, providing access to the data entities and file management aspects of ODB. The `Database` class, defined in a module of the same name, provides a *change dictionary* for databases opened with write permission; this is used to cache database changes to allow a limited form of transaction processing, including support for commit and rollback. The limitation lies in the lack of protection against I/O failures on the mass storage devices associated with the control and data files of the database. The change dictionary is defined in the module `Delta`.

1.2.1 Examining the Database Structure

The structure of an ODB database is defined by a set of *entity classes*, each containing *fields* constrained by data type and the allowed number of values. The set of classes and their fields and constraints form the *data dictionary* of the database. In the *odb*/Tools, this information is provided through an instance of the `DataDictionary` class, which provides information regarding the structure of the database as a group of objects which represent class definitions, which in turn provide field definition objects to represent the constraints placed on each field. These objects provide methods by which various attributes of the definitions may be queried and modified, while limiting the flexibility to modify existing definitions by disallowing restrictive modifications. This limitation is imposed to prevent unintentionally introducing integrity failures in objects already in the database. To allow this limitation to be bypassed, an application may explicitly indicate that this safety check be ignored, acknowledging responsibility for maintaining object integrity. The classes related to the data dictionary are defined in the modules `DataDictionary`, `ClassDefn`, and

`FieldDefn`.

1.2.2 Accessing Database Entities

Objects in a database are accessed by subscripting a corresponding `Database` object with the object ID of the desired object. Methods are provided to query the database for a list of existing object IDs, as well as to determine the presence of a particular object ID. Additional methods support the creation and deletion of objects from the database.

The database objects themselves are implemented as a set of classes, each with a different purpose and level of capability, but all share a common base level of functionality and a common interface. The primary interface to these objects is defined by a pair of protocols. Each protocol defines a minimal set of methods and semantic behavior that is provided by all implementation classes. This interface is defined by the *Object Protocol* and is given a minimal implementation by the `ObjectProtocol` class defined in the module of the same name. This class is used as a base class by all other classes supporting the protocol. The protocol defines means by which the program objects representing database entities may be queried to determine their state, including which fields are represented, the object ID, and the ID of the database class of which it is an element. Two mechanisms are provided to access the data fields of the object, one providing a short-hand notation which is likely to be of use to most application programmers and which provides the most readable form of field access, and another, more general access method. The short-hand method imposes the restriction that the name be a valid Python identifier, and that it not be a Python reserved word. The general mechanism imposes no restrictions.

Distinct classes are used to represent read-only and read/write objects. Subscripting a database object with an object ID yields the appropriate object class for the access permissions of the database. The simpler read-only object is used where possible as a performance boost as well as to reduce the memory footprint of the objects extracted from the database. For write-enabled databases, an object is produced that orchestrates data retrieval from a read-only object and a delta object produced by the change dictionary and

that redirects updates to the change dictionary object as well.

As with database entities, individual fields of entities are represented by objects and share a corresponding interface, known as the *Field Protocol*. The base class for objects supporting this protocol is implemented in module `FieldProtocol`. This protocol provides support for querying the status of a field, determining the number and indexes of values assigned to the field, determining the parent object from which the field was derived, and providing information that can be used to retrieve additional information about the field from the data dictionary. Specific values can be retrieved from a field, or they can be set for fields derived from a database opened with write permission. Field values can be set using simple assignment expressions, with values being managed by the change dictionary and object internals being managed internally. One implementation of the Field Protocol is provided for each class supporting the Object Protocol.

1.2.3 Change Management

The change dictionary used by the `Database` class operates in much the same way as the database, allowing objects to be retrieved, queried, and updated. Objects retrieved from the change dictionary parallel the user-level objects provided by the `Database` methods, including the field-level objects retrieved from the entity objects. The most important difference is that they are created on reference rather than requiring explicit creation. This allows the change dictionary entities to provide objects with an interface that conforms with that defined by the object protocol.

This component of the *odb*/Tools supports a large volume of changes to the database while maintaining a low memory profile. This is achieved by placing change entries in a GDBM database on disk when they are not referenced by any active object in memory. This database is managed entirely by the change dictionary and is removed when the dictionary is closed or de-allocated. The change dictionary is used to drive the commit procedure on the `Database` object and is replaced during a rollback operation.

1.2.4 Iterating Over the Database

The *odb*/Tools support the idea of using high-level iteration mechanisms with distinct selection and operation components to operate over a database. Classes are provided to represent the basic traversal mechanisms and some common selection semantics. A discussion of the selection mechanism follows in the next section, and information on extending the selection system is given in the reference material in Section 8.3. Operations are considered application-specific and must be provided by the *odb*/Tools-based application.

To allow a flexible handling of the database traversal, *odb*/Tools provides two primary iteration mechanisms via the classes `FlatIterator` and `DepthIterator`. Each of these produces a distinctly different result and applies to different situations. Both of these classes, together with a common base class, are defined in the module `Iterators`. The simplest iterator, `FlatIterator`, visits each node in the database without regard for ordering or structure, and passes the node to the operation if the selection mechanism indicates acceptance. This will be most heavily used in searching applications for which no relevant index exists, as well as for programs which build indexes.

A more elaborate mechanism for traversing the database is a depth-first traversal, implemented by the `DepthIterator` class. Unlike the `FlatIterator`, which has no regard for structure in the data, the `DepthIterator` is dominated by structural interpretation of the data. While the `FlatIterator` is controlled only by the selection mechanism, the depth search has a somewhat more complicated set of controls to match the extended semantics of the iteration. These controls are defined in the `Traversals` module. An object called a *traversal predicate* is used to determine if links exhibiting a particular relationship between two nodes of the database are traversed. These predicates are offered in the form of functions with bound state, in some ways similar to the lexical closures of Lisp and Scheme. The function part of a traversal predicate receives information about the entity classes and field definitions of the endpoints of a link and may use these to determine if the link should be followed. The state of a predicate may be empty, statically initialized when the predicate is

defined, dynamically initialized when the predicate is instantiated, or actively mutated over the life of the predicate. Note that the predicate does not receive the actual objects at each end of the link, and has no access to the field data of the objects. The entire decision must be made with no more information than the class and field definitions and the current state of the predicate. The intent in providing objects with this behavior is to allow a means to prevent the `DepthIterator` from traversing links which cannot yield results of interest to the application. Fundamental traversal predicates provided by *odb*/Tools include acceptance based on the class and field identifiers for the source object, as these are expected to be useful to most applications, and the logical operations *and*, *or*, and *not*, which may be used to provide composition of other predicates. Iterators are discussed in detail in Chapter 7.

As an example, consider an application which receives a set of criteria for selecting a document through a dialog with a user. The user indicates that information is needed regarding the authors of the documents selected, including a list of institutions with which they are associated. A `FlatIterator` can be used to locate the document objects which match the search criteria provided by the user. The operation that takes place for each object found by the search can use a `DepthIterator` to descend into the list of authors to retrieve their names and a list of institutions they are affiliated with. This information can be printed as it is found or stored in a data structure for further processing.

1.2.5 Selecting Objects

The previous section intersperses many references to the "selection mechanism" without providing any details. This section will provide those missing details. The material described here is defined in the module `Predicates`.

The selection mechanism is implemented by *selection predicates*, a set of functions with optional bound state information. This mechanism is quite similar to the traversal predicates defined above, though perhaps simpler to understand. Selection predicates are used to determine which database entities are selected using only the information available through

the database object and the state of the predicate. As with traversal predicates, there are no restrictions on the maintenance of state information. Logical compositions using *and*, *or*, and *not* are supported.

For many of the predefined selection predicate classes, the state associated with the predicate is used to store a field name and a string against which values of the field should be matched using some algorithm. A variety of string-matching predicate classes are provided, including approximate string matching. These classes are useful in conjunction with both the `FlatIterator` and `DepthIterator` classes. The `AcceptOnce` predicate allows a depth search to control how many times a node may be retrieved. An extensible demonstration class has been implemented which collects statistical data on the database entities for which it is queried. Detailed information on predicates and predicate classes is available in Chapter 8.

1.3 How the *odb*/Tools are Used

Since *odb*/Tools is a new package, there are few applications that have been created using it. However, some demonstration applications have been created to serve, in part, as examples that are discussed in detail in the reference documentation. This section discusses these applications and describes potential future applications, both in the context of Envision and as a general tool associated with the ODB library.

The simplest applications based on *odb*/Tools will be reporting applications which examine the database for some criteria to present simple list-oriented reports. The existing *odb*/Dict application is an example of this style. The application opens a database and presents a simple report listing all the entity classes defined for the database, each with a list of fields present in the class and constraints placed upon them. The implementation requires under 150 lines of Python code in two modules, including command like processing, report formatting, and source code comments.

A future application related to the maintenance of the data dictionary of an ODB

database could support a graphical interface to allow editing of the entity class definitions, with information about the relationships formed by the data content of link fields displayed. Such an interface would be useful since existing tools are tedious to use, and would provide new functionality by allowing the actual link relationships to be readily examined in the general case. This is important because the ODB database library does not provide for general endpoint constraints on link values: the only restriction is that each endpoint be a link field. For example, tools created using *odb*/Tools can check that all links from a field "author" of entity class "document" terminate at entities of class "person" or "organization."

Future research in digital library applications and technologies will require statistical information regarding the databases used to test new algorithms and techniques in order to determine the applicability of the databases to the problems being solved, or to measure the effectiveness of algorithms. *odb*/Tools supports collection of information across databases; see Section 8.3 for an example of statistic collection.

Another intended application for *odb*/Tools is the creation of an application which analyzes the content of an ODB database to locate possible data faults that may have resulted from failures in the initial import of data from full-text sources. Additional applications may be created to repair problem data; these programs will have to deal with decomposing and merging subgraphs of the database to allow identification errors to be corrected as well as allow simple editing of data values. This undertaking will require further research in repairing faulty information graphs in order to identify the full scope of the problem and determine a general solution.

Applications currently in their design and implementation phases include a simple search utility to operate on the Envision database, a date format analysis and conversion utility to ensure that dates stored in existing databases will be usable beyond the end of the millennium, and a demonstration utility to merge two entities into a unified entity with meaningful links to related entities.

1.4 Selection of Language

The Python programming language was selected as the host environment for a number of reasons. Most importantly, it is a powerful object-oriented language well suited for both medium-sized projects written entirely in itself and for embedding in applications written in other third-generation languages. The ease of extensibility of the interpreter implementation proved valuable in creating the low-level database interface and provides incentive for others to create extensions as well, providing access to a great many facilities from a single programming environment.

Another important aspect of the language selection revolves around usability: the language chosen needed to be easy to learn quickly, or at least easy enough to get useful work done with relatively little delay. For this reason, modularity and popularity were both considered important. Tcl/Tk is a leader in popularity, but does not scale well or offer much modularity in its raw form. Large applications are difficult to develop due to the single name space and high level of overhead achieved through repetitious parsing of static program text. Occasional Tcl programmers find the string-processing foundations of the language difficult to deal with [Con95]. Perl is perhaps even more popular, scales fairly well now that modules and object-oriented programming are supported in the most recent releases, and has a high level of performance. Unfortunately, it remains cryptic and does not enjoy broad acceptance as a general purpose language. Its strength lies in large part in its reputation as a system administration tool. Forms of Lisp and Scheme were considered as well, primarily due to the applicative nature of the language and the availability of interpreters that could be modified and extended in much the same way that the other languages considered could. These were rejected primarily due to lack of documentation on extending the interpreters with new functions provided in C, as well as the lack of familiarity with functional programming among many programmers.

The Python environment offers modularity and object-orientation, is well-accepted for small- to medium-sized projects, and executes quickly since, like Perl, it is byte-compiled

rather than interpreted directly from the source text. Projects as large as twenty thousand source lines have been implemented and considered successful [vRdB91]. This is quite large for interpreted languages.

1.5 Efficiency Notes

A number of factors affect the efficiency of most database applications, and *odb*/Tools-based systems are no exception. There are a number of internal aspects of *odb*/Tools that address efficiency and work to alleviate unnecessary entity retrievals from the database. While these manipulations are handled within *odb*/Tools and should never affect application code, it is important for programmers to be aware of these aspects of internal operation. In all cases, these points are discussed in the reference chapters in which they are appropriate. This section discusses the motivation for the way efficiency issues are addressed.

Improving performance in *odb*/Tools is accomplished primarily through lazy evaluation of queries against the database itself, delaying actual retrieval of objects until they are needed. Evidence of delayed retrieval is seen in the base `odb` module in the result of the `keys()` method of the `OdbDatabase` object, in the arguments to the traversal predicates used in depth-first traversal of a database, and is a result of delayed evaluation of iterators as well.

Limited caching is performed by the `odb` module as well, allowing multiple requests for the same entity to use the program object as long as there is a reference to the original object, alleviating repeated database accesses. Using this cache requires that some reference to the original object exist in the application code; *odb*/Tools will not force objects to remain in the cache without a reference.

The application programmer can gain the most advantage of these measures by creating traversal predicates that make decisions based on the link source rather than the target, or arranging composite predicates in such a way that short-circuiting of boolean evaluation may avoid using information about the target as often as possible. This improves performance

by avoiding the retrieval of the target entity whenever possible. Iterators used in parallel with user interaction may be used to improve performance by avoiding the evaluation of the entire iterator at any single point during program execution. Based on the interaction required, this may be possible, allowing the user to cancel an operation or otherwise interact with the application without having to analyze the entire database.

1.6 Python Reference Material

Though the volume of reference material published on paper for Python is low, the language is beginning to receive an increased level of press in various magazine and journals relating to computer science. More importantly, the material available as part of the standard distribution is of good quality, and exceeds what is available for many interpreted languages used in research environments.

For beginners with Python, the distribution includes a document titled simply *Python Tutorial* [vR95d]. This document provides an introduction to using the Python language and the interpreter, and shows a few simple examples of the language. Detailed information on Python can be found in the *Python Reference Manual* [vR95c], which defines the language constructs formally and in detail, and the *Python Library Reference* [vR95b], which gives information on the many standard modules provided with the interpreter. A number of contributed demonstration programs are provided as well, serving as examples of module usage and demonstrations of common idioms used in crafting Python programs. For advanced users interested in extending the interpreter or embedding it into an application, a supplemental manual titled *Extending and Embedding the Python Interpreter* [vR95a] is provided. This manual is very much a work-in-progress.

The primary Python distribution and supporting material may be found on the Internet for FTP at `ftp.python.org`, with online documentation available as hypertext through the World Wide Web at URL `http://www.python.org/`. This site is home of and maintained by the Python Software Activity. These materials are mirrored in Europe at `ftp.cwi.nl`

and `http://www.cwi.nl/~guido/`. The USENET newsgroup `comp.lang.python` supports ongoing discussion and information about current events and developments in the Python community. Refer to Appendix A for more information about obtaining the Python distribution.

In early 1996, an introductory book on writing Python programs, written by Mark Lutz, will be published by O'Reilly & Associates [Lut96].

1.7 Overview of Modules

The *odb*/Tools interface is defined by several modules, each providing an element of the package's functionality. Many of these can be used with only a few of the others being available, but they are intended to be used together. There is one module that is written in C and must be compiled and linked into the Python interpreter, or at least made available as a shared object for those platforms that support dynamic linking and loading. Other *odb*/Tools modules are written in Python. A graphic display of the relationship of several of the high-level objects to the foundation object in the C module is given in Figure 1.2.

`odb`

This low-level database module, written in C, is required and must be available to the interpreter to use the *odb*/Tools facilities. It should be considered internal to the *odb*/Tools package and not accessed directly. The other modules of the package will work with it as needed, and provide an object-oriented interface to the base facilities available here.

`Database`

Entire databases are presented to application code as instances of the class implemented by this module. Both read-only and read/write support is provided. All entity access and transaction control is handled through this module.

`Delta`

This module provides the Change Dictionary facility used to enable transaction-

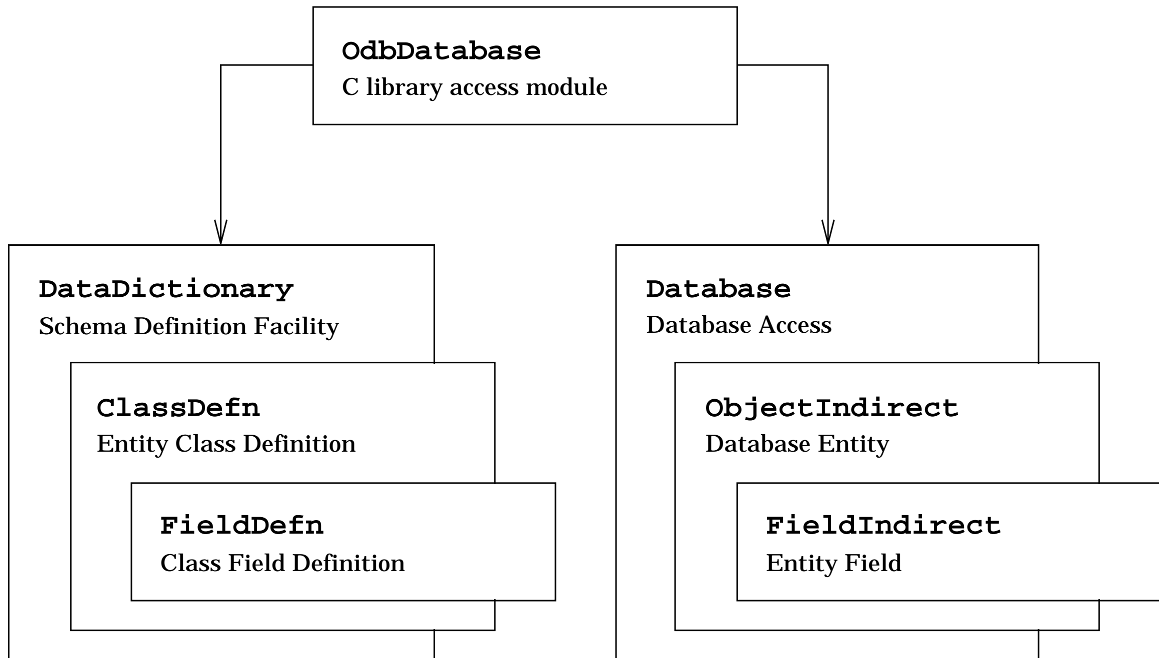

Figure 1.2: Module Relationships in *odb*/Tools

oriented use of ODB databases. This facility allows changes to be made to the database without requiring updates to be stored to disk until a review and confirmation can be performed.

`DataDictionary`

The structure of an ODB database, given by the entity classes and the fields they contain, is represented by the Python class defined in this module. The data dictionary can be interrogated and modified through this interface.

`ClassDefn`

A single ODB class within the database is represented by the Python class in this module. Class attributes made available directly include the class ID, name, alternative ID, the "next field ID" and "other info." Updates to these attributes are performed according to the restrictions imposed by the ODB library and database permissions. Access to field information is provided indirectly through the `FieldDefn` module.

`FieldDefn`

Fields within ODB classes are represented by this Python class. Field attributes made available include the number of occurances, name, ID number, and type. Updates to these attributes are performed according to the restrictions imposed by the ODB library and database permissions.

`DDictReport`

An example use of the data dictionary support provided by the previous three modules is implemented here. This is more application oriented, and serves as an example use of the data dictionary classes defined in the three modules listed immediately above. It receives no further mention in this document.

`Iterators`

Primitive mechanisms for traversing the database are defined in this module. Sequential and depth-first traversals are provided.

`Predicates`

Many basic selection predicates are defined in this module. These are sample predicates for use in iteration over an ODB database, and may be used directly or composed with application-specific predicates as required. Logical composition classes are provided through this module as well.

`CountPred`

The `CountPredicate` class defined in this module is an example of extending the set of available predicates and is discussed in section 8.3.2.

`Traversals`

This module mirrors the `Predicates` module, but provides traversal predicates for manipulating the depth-first search mechanisms. The logical composition predicates exported by `Predicates` are shared and exported by this module as well.

The *odb*/Tools package requires the `gdbm` module, an optional module provided with the standard Python distribution, to use the Change Dictionary. Since the ODB library already requires that the GDBM library be linked with the interpreter, there is little additional code

introduced by adding this module. If the `soundex` module is available, the `Predicates` module is able to take advantage of it, but it is not required. This capability provides approximate string matching for selection predicates, described further in Chapter 8.

1.8 Documentation Conventions and Notes

There are a few typographic conventions used in this manual that readers should be aware of, though most readers will probably be familiar with these. Text is used to communicate all of the information in this manual, though there are several different uses of text in the computing environment being described. Different typefaces are used to indicate information about the specific meaning of each text element in this document. The text "ODB" refers to the base object database package, and "`odb`" refers to the Python interface module of the same name.

Roman	Normal text.
Italic	Terms being defined.
`Courier`	Code fragments, function names in text, module names.
Courier Italic	Variables in code fragments.

At all points, some familiarity with the structure of an ODB database and the Python language is assumed. An introduction to several concepts found in the ODB library and databases it handles is provided in Chapter 2. Some specific points to keep in mind include:

- When calling a Python function, object method, or class method, a `TypeError` exception is always possible if too few or too many parameters are supplied, or if they are of the wrong type. This point is not mentioned for each function or method when exceptional conditions are discussed, but such exceptions are possible.

- A `MemoryError` exception can be raised at any time due to a `malloc()` failure. This should be considered a possibility at any time, including during a call to a function or method provided as part of the ODB interface, regardless of whether the implementation was in C or Python.

- Object methods with names of the form `__*__` are used to implement *disciplines* in the Python environment. These disciplines are used in much the same way that overloaded operators are used in other object-oriented languages. The required semantics for these methods are detailed in the Python reference documentation. [vR95c]

1.9 Organization of the Report

The remainder of this report provides information important in using *odb*/Tools. After a brief review of concepts particular to the ODB database library, the objects and methods in the *odb*/Tools library are covered in detail. Appendices describe the installation procedure and using *odb*/Tools scripts as command-line utilities in the UNIX environment.

Chapter 2 describes several of the underlying concepts reflected in the design of the ODB database library and why they are appropriate for use in Project Envision. The chapter includes information on database organization, structure, and constraints; indexing facilities; and file organization.

Chapter 3 provides reference documentation on the `odb` module and the objects created by the functions in that module. This includes the `OdbDatabase` and `OdbInstance` objects, upon which most other modules in *odb*/Tools build.

Chapter 4 describes the data dictionary supported by ODB databases and the interface used to query and manipulate the dictionary through *odb*/Tools. This includes information on the `DataDictionary`, `ClassDefn`, and `FieldDefn` modules.

Chapter 5 provides reference information on the high-level interface to the database itself, including information on gaining access to the data dictionary and retrieving entities from the database. The transactional support provided by *odb*/Tools and other information particular to managing a database are included in this chapter.

Chapter 6 covers the behavior of individual entities retrieved from the database, and of the fields retrieved from each entity. This includes information on the classes which support the protocols defined here and how to use the protocols to retrieve information

from an entity.

Chapter 7 describes the high-level database traversal mechanisms provided by *odb*/Tools, including both sequential iteration and depth-first traversals. Simple examples are included.

Chapter 8 includes information on the predicate classes provided with *odb*/Tools, using predicates, and extending the set of predicates. This chapter covers the *odb*/Tools modules `Predicates` and `Traversals` and the `CountPred` example module.

Appendix A describes the installation procedure, and lists the versions of various associated software that have been tested with *odb*/Tools in a variety of operating envirionments.

Appendix B covers using *odb*/Tools scripts as stand-alone applications under environments which support UNIX-style interpreted execution from the command line.

Chapter 2

The ODB Object Database

The Envision object database, initially defined by the output of the DELTO document analyzer and managed by the functions of the ODB library created as a part of that project, is a fairly simple database with a number of useful attributes use by Envision [Ave95]. There are a number of structure and administrative aspects of the database which are important for any application which uses the ODB library either independently or as part of Envision. This chapter provides an overview of these concepts and the terminology used to describe an ODB database. Familiarity with these concepts and terms is assumed through the rest of this reference document.

2.1 Organization of Stored Information

The information stored in an ODB database is structured as entities, or objects, of various *classes*, each of which has a set of *fields* with constraints on the type and number of values each may take. Each class and each field can take on certain attributes, some of which are required to be unique within the enclosing larger context. Each entity is identified by a unique string called the *object ID*.

Each class is identified by a string of characters terminated by a trailing zero byte. All upper-case characters are mapped to lower case within the ODB library, making the identifiers case-insensitive. Each class identifier must be unique across the entire database. Also associated with each class is a name, which is case-sensitive and need not be unique. This name provides a way to identify a class in a very high-level fashion, typically as a human-readable name, while the class ID is typically short. Allowing the ID to be short is

important since it is stored with each entity.

Additional attributes of class definitions include an *alternative ID*, a non-negative numeric identifier which must be unique across the entire database, an *other info* text field for holding arbitrary additional information, a number which is used internally to form field identifiers when fields are added to the class, and the actual list of fields defined for the class. The alternative ID is provided as a convenience for application programmers using the C interface to ODB, and is not normally used otherwise.

2.1.1 Field Definitions and Constraints

Field definitions in ODB are somewhat more elaborate than class definitions. Each field has a number of attributes, including a numeric ID which must be unique within the class, but which also is generated by ODB when each field is created rather than being specified by the schema designer as are the class IDs. Each field also has a name which is a zero-terminated character string and must be unique within the enclosing class, and is case-insensitive in the way that the class identifier is insensitive to case. In addition to these attributes, fields have a *type* attribute which specifies the type of data which may be stored in the field. Valid types include STRING, DATE, OBJECT ID, and LINK. The actual data for all field types are zero-terminated strings, but semantic distinctions are defined on these types to allow an application to format data meaningfully.

The STRING and DATE types are self-explanatory, but the OBJECT ID and LINK types require further explanation. The OBJECT ID data type specifies an object ID, identifying a single entity from the database. The LINK data type is handled specially under ODB to form bi-directional relationships between objects. When a value is added to a LINK field, called the `source field`, that value corresponds to an object ID and a field ID where the field is specified as part of the entity referred to by the object ID. This is the *target field*. A value is automatically added to the target field which forms a relationship back to the source field, making the link fully bi-directional. Removing a value from a LINK field removes the corresponding value in the target field as well. Special methods are

required to manipulate LINK fields.

Each field defines constraints on the number of values which may be assigned to the field. These constraints take the form of minimum and maximum values, where the maximum my be unrestricted. The minimum number may be any non-negative integer, and the maximum may be any positive integer or unrestricted. Entities which contain fields where the number of values assigned does not fall within the range specified by the constraints are not stored in the database; storing such an entity would constitute an integrity failure in the database. These integrity constraints are required to hold true after deletion of field values or entire entities as well. An operation which would create an integrity failure is not completed, but returns a failure condition to the application. Entity deletion could cause an integrity failure by causing the number of values of a LINK field referred by the object being deleted to drop below the minimum number set for that field.

2.2 Indexing Facilities

ODB provides an indexing facility to enable an application to rapidly search for records matching a given string value. Matching is performed in a case-insensitive manner. For each string in the database index, a list of object IDs is available which contains each entity explicitly indexed under that string. The indexed strings may be data values from some field of the entity, or they may be computed from the fields of the entity. The use of arbitrary strings is permitted.

The indexes are not maintained automatically, as no mechanism is in place to describe how an index should be built or updated in response to changes in the entities of the database. This is a serious failing in ODB which should be corrected at a future date. Tools are available to generate indexes for the entire database using information provided by an interactive user interface.

2.3 File Organization on the Persistent Store

The ODB library uses the GDBM database library to handle most of the files it uses on external storage devices. This library was developed by the Free Software Foundation in association with the GNU project [Gau94]. All large data files and the entity definition file are GDBM databases. There are a few additional control files which are defined as textual data.

All files in an ODB database can be grouped into one of four categories. Two categories, the *object file group* and index file group, are used to store the database entities and string indexes, respectively. Each of these categories is implemented as a group of files which, taken together, store all the data required for that aspect of the database. Within each group, one file is specified as *current*: this is the file in which new entries for the group are stored. This setting is independent for each of these two file groups, as is the number of files used. For each group, there is also a maximum number of files which may be used, determined by a compile-time constant in the ODB library.

The next category on the persistent store contains a single file used to store entity class definitions. This is a single GDBM file containing an entry for each class definition. One entry is used to store all the information for a class, including all the associated field definitions.

The final category consists of the various control files containing textual information. Files are used to record the "high points" for the object ID identifiers, to allow creating new, unique IDs quickly and responsively, and to record which file in each group is current, to allow this setting to be persistent across invocations of various database tools. Unfortunately, there are several very small files in this group, rather than a single control file.

No facilities are provided in the base ODB library to support transaction-level controls, multiple writers, or entity-level locking. Only one writer is allowed, and this writer is sufficient to block all other access to the database. This is a limitation of the underlying GDBM implementation.

Chapter 3

Low-level Database Interface

The `odb` module provides an interface to the ODB object database library developed by Guillermo Averboch for Project Envision. This interface is intended to allow Python scripts to be created which perform useful operations on the databases generated by DELTO and queried by other Envision programs. The Python modules which use this library are portable to all operating systems provided that the GDBM library is available.

To provide the needed access to the ODB library, this module was implemented in C and must be linked into the Python interpreter along with the base ODB library. Service modules written in Python should be used to provide various aspects of object control and semantic interpretation. Several such modules are described in later chapters of this document, and should be used as the basis for applications using ODB whenever possible.

3.1 Built-in Module `odb`

This module provides a few functions to provide a minimal interface to the base ODB library, and two public object types which provide most of the functionality of the underlying database facility. The interface is object-oriented, but many concessions were made to produce this layer quickly and with emphasis placed on maintaining the integrity of the database.

Functions are available to open an existing database, retrieve a list of the data types available in the ODB library, affect the blocking factor of index entries, and retrieve the text of the global ODB error message. All access to the objects in the databases and the indexes is provided by the `OdbDatabase` and `OdbInstance` objects. The former object type

is used to represent and manipulate the database as a whole, including updating the data dictionary and the later is used to manipulate individual entities in the database.

3.2 `odb` Functions

The `odb` module defines the following functions:

`error_msg()`

This function returns the error message located within the ODB library. This message is not bound to an object as there is no intrinsic association between the message and the operations performed by the ODB library. No matter how many databases are open, there is only one error message buffer. It is generally preferable to use information associated with exceptions raised by the `odb` module to diagnose problems. This error message may be included in the exception's associated information where appropriate. If there is no error message, this function returns an empty string.

`field_types()`

The ODB database library supports a small number of data types for field values. These types are identified in the Python environment by string names. This function returns a list of the valid type names. Currently, each database can only support those type identifiers defined globally and included in the list returned by this function. See also the `field_types()` method of the `OdbDatabase` object.

`index_extra_blocking(` [*blocksize*] `)`

Get or set the size of the block allocated to an index entry. This corresponds directly to reading or setting the *index_extra_blocking* variable defined by the ODB package. The optional *blocksize* parameter sets the value, and must be a non-negative integer. The function always returns the value of the *index_extra_blocking* variable at the end of the function. The only exception which can be raised is a `TypeError`, which can occur if too many parameters are passed or if a single parameter is not a non-negative integer.

`open(`*basename* [, *rwmode* [, *filemode*]] `)`

Open the ODB database named by *basename* if possible. If a relative or absolute pathname is needed, it should be given as part of *basename*. It is important that no file extension be used; the database consists of several files, with extensions used to distinguish among them. In order for the files to be found, the extensions, including the "dot" character, cannot be used. The *rwmode* parameter specifies the desired access to the database; the default value is `'r'` for read-only access. Possible values include `'r'`, `'w'`, and `'rw'`, where the later two both indicate read/write access. The database must already exist; it cannot be created or reset through this function. These capabilities can be easily implemented in Python should an application require the capability. The *filemode* parameter is used to indicate the file permissions which should be requested when adding a file to the database using the `OdbDatabase.add_index_file()` or `OdbDatabase.add_object_file()` methods. In the future it may be used to specify the file mode for database creation as well. The default value is `0666` (global read/write access, adjusted by the user's current `umask(2)` setting). When successful, the `open()` function returns an `OdbDatabase` object. The database object supports a `close()` method, which will be invoked automatically when the object is garbage collected or explicitly by the programmer.

3.3 `odb` Public Data and Exceptions

The public data in the `odb` module includes two type objects, an exception, and several constant definitions which reflect limits in the underlying ODB library. For programmers not familiar with types in Python, a few words are in order. A Python *type* is not congruent with the formal notion of type found in most current programming languages and is especially distinct from that found in languages requiring strong static typing. Not only is Python typing dynamic in a manner very similar to Lisp, but a "type" is actually an object which can be passed like an integer or object instance. These type objects can be acquired from any

object, including other type objects, by using the built-in `type()` function. Unfortunately, this is the only way to retrieve type objects in most circumstances. The standard module `types` provides bindings between identifier and type objects for all types which are provided with the standard interpreter. It creates these bindings by using the `type()` function on examples of each type of object. Unfortunately, this is not necessarily a reasonable approach with any of the objects created by this module. These two type objects are provided by `odb` to avoid the potential need to create instances of these objects simply to gain access to their type objects.

`OdbDatabaseType`

> This is the type object representing the database objects returned by the `open()` function. It is provided to allow testing for the type without having to create a dummy database to use in obtaining the type descriptor.

`OdbInstanceType`

> The objects stored in the databases have this type. Like `OdbDatabaseType`, it is provided for convenience and to allow the avoidance of creating spurious objects that must be destroyed.

`OdbKeysIteratorType`

> This is the type of objects returned by the `keys()` method of the `OdbDatabase` objects. These are sequence objects which are evaluated lazily, yielding keys from the database. Additional information is available with the `keys()` method documentation.

`error`

> Exception raised when an operation fails for some reason specific to the semantics of the ODB library. Errors related to input values and I/O raise the appropriate errors. The documentation for each method of the OdbDatabase and OdbInstance objects includes information on the exact exceptions each may raise. Most of the "associated values" are strings, but many are tuples including an error code from the underlying GDBM library as well as the GDBM or ODB error text. Each of these will have the format `(errcode, 'odbmessage', 'odblib_or_gdbm_message')`. The

exact meanings of the exception values is described under the function or method that raises it, for those cases where the value is not simply a descriptive string.

The remaining data items are constants matching limit definitions of the base ODB library. These are provided as reference values only, but are valid for the version of the ODB package against which this module is compiled.

`MAX_CLASS_ID_LENGTH`

This is the maximum length of a class ID which may be used to create a new class.

`MAX_CLASS_NAME_LENGTH`

This is the maximum length of a class name which may be used to create or update a class definition.

`MAX_CLASS_OTHER_INFO_LENGTH`

This is the maximum length of the "other info" attribute used to create or update a class definition.

`MAX_DATABASE_NAME_LENGTH`

This is the maximum length of the base name of a database, including any absolute or relative path information needed. If a longer name is needed, use a short, relative name here and always switch to the appropriate directory before opening the database or adding files to the database using the `add_index_file()` or `add_object_file()` methods of the `OdbDatabase` objects.

`MAX_ERROR_MSG_LENGTH`

This is the maximum length of the message returned by `error_msg()`.

`MAX_FIELD_NAME_LENGTH`

This is the maximum value of a field name within a entity class definition.

`MAX_INDEX_STR_LENGTH`

This is the maximum length of a string which can be indexed.

`MAX_NUMBER_INDEX_FILES`

This is the maximum number of files which may be created to store index information in a database.

`MAX_NUMBER_OBJECT_FILES`

This is the maximum number of files which may be created to store entity data.

`MAX_OBJ_ID_LENGTH`

This is the maximum length of an object ID.

3.4 `OdbDatabase` Objects

OdbDatabase objects (returned by `open()` above) have several methods and a number of data members. These methods are defined to allow most operations allowed by the ODB library that require the database itself to be operated upon. Some of the data members also allow operations to be performed on the database through assignment, though most are read-only. These are not intended for general use; wrapper classes providing appropriate semantics should be used for that purpose. Several classes associated with the ODB interface are provided in associated modules described in subsequent sections.

It is important to know that in all cases, operations which require that a database be open with write permission will raise an `odb.error` exception with an appropriate message string as an associated value. If this requirement applies to a method, it will be noted at the beginning of the description. Equally important is the requirement that a database remain open when any of its methods are invoked, including such queries as retrieving the list of classes defined in the database. This requirement exists to protect the application from opening the database to gain information, closing it, and re-opening it with the intention to write updated information to the database. While this is not sufficient in itself, it does encourage the careful use of all database information.

3.4.1 `OdbDatabase` Methods

General Methods

`add_index_file()`

[Requires that the database be open with write permission.] This function attempts

to create a new GDBM file and add it to the index group. The access control permissions used to create the file are those specified when the database was opened and made available as the `filemode` attribute. If the file is created successfully, the new number of index files is returned and made the new value of the `number_index_files` attribute. This does not affect the value of the `current_index_file` setting. If the limit on the number of index files has been reached, or the database is closed or opened without write permission, an `odb.error` exception is raised. If the GDBM library is not able to create a new file, an `IOError` exception is raised.

`add_object_file()`

[Requires that the database be open with write permission.] This method corresponds to the `add_index_file()` method, but operates with the object file group. If successful, a new file is added for the storage of object data, and the `number_object_files` member variable is updated. On failure, an exception is raised; refer to the description of the `add_index_file()` method for details on the exceptions which may be raised. The new file is not made the default for storing new object data; the setting of the `current_object_file` must be adjusted to make that change.

`close()`

The database can be closed at any point by using this method. For databases opened with write permission, this can be used to flush any changes to disk; for read-only database objects, this can release the database to a process waiting to open it with read/write permissions. This release only actually occurs if there are no other readers of the database at the time. Once the database is closed, almost no operations can be performed on the database object. This method is made available in addition to closing the database on object deletion to allow the application a way to ensure the physical database is closed immediately rather than relying on the garbage collector to finalize the object. This is the only way to control closing of the database.

Data Dictionary Manipulation

`add_class`(*id, name, alt_id, other_info,* (*name, type, min, max*) ...)

[Requires that the database be open with write permission.] Creates a new class in the database. All arguments are required, and must include at least one field definition. The *id* and *alt_id* parameters must be unique within the database; if either is not, an `odb.error` exception is raised with an associated value string stating that one or the other is already in use.

`add_field`(*class_id, name, type, min, max*)

[Requires that the database be open with write permission.] Adds a field to an existing class. If the database is read-only, an `odb.error` exception is raised. Any other error relates to the value of one or more parameters, and a `ValueError` exception is raised with a string associated value. The class affected, identified by *class_id*, must already exist, and the new field name must be unique within the class. If any error occurs, a `ValueError` or `IOError` exception is raised with an appropriate message string as an associated value. An `odb.error` exception indicates an internal failure.

`classes()`

Returns a list of class IDs defined for this database. Each ID is a non-empty string. These ID strings, or strings that are equal to these, can be passed to the `get_class_info()` method to obtain more information on any named class. As an example, the expression '`"`*cid*`" in` *db*`.classes()`' should be used to determine if the class `"`*cid*`"` exists in the database *db*.

`delete_field`(*class_id, name*)

[Requires that the database be open with write permission.] Fields may be removed from class definitions using this method. The class must be identified by the string *class_id*, and the field by *name*. If the class is not present in the database, or does not contain a field named *name*, a `ValueError` exception is raised. An `IOError` is raised in the event of a failure while storing the updated class definition, and an `odb.error` exception is raised if the named field is the last field in the class. Note that classes cannot be deleted from the database, and all classes must contain at least one field.

These are limitations of the base ODB library.

`field_types()`

The `field_types()` method is actually identical to the module function of the same name, but is provided in the hope that ODB may someday allow additional data types to be defined for each database, with needed support or descriptors stored in the database. In the meanwhile, this method provides a "free" forward compatibility path for applications. Using this will allow an augmented ODB and *odb*/Tools with the ability to define additional field types per database to replace the existing libraries without requiring application changes.

`get_class_info(`*class_id*`)`

This method allows the application to get information on a named entity class defined in the database. The *class_id* specifies the ID of the class of interest. If the class exists in the database, the method will return a structure of the form:

```
('id', 'name', alternative_id, 'other info', next_field_id,
        (field_id, 'fieldname', 'fieldtype', min_occur, max_occur),
        ...
        )
```

If the specified class doesn't exist, an `odb.error` exception is raised.

`update_class(`*id, name,* [*alt_id,* [*other_info*]] `)`

[Requires that the database be open with write permission.] Modifies the attributes of an database entity class definition. The *id* string identifies the class to update. Any other parameter may be `None` to indicate that it should not be changed, or take on a string (*name, other_info*) or integer (*alt_id*) value to update that attribute of the class definition. A true value will be returned on success. An `odb.error` exception is raised when a new alternative ID duplicates one in use for a different class, and a `ValueError` is raised if the specified class does not exist in the database.

`update_field(`*class_id, oldname, newname, type, min, max*`)`

[Requires that the database be open with write permission.] In a manner similar to that defined for `update_class()`, this method provides a mechanism to change

the definition of a field within an already-defined class. The field named *oldname* in *class_id* is updated to have the new attributes specified in the parameters. The *newname* and *type* parameters may take the value `None` to indicate that they should not be changed. If any of the parameter values are illegal, or if the class is not found, a `ValueError` exception is raised.

Entity Handling Methods

`[`*object_id*`]`

The use of the indexing operator is provided to allow retrieval of an object from the database. This operator is the only way to extract an object. The *object_id* index must be a string representing the object's ID, and will be an element of the list returned by the `keys()` method. The return value will be an OdbInstance object as described below. If *object_id* does not correspond to an object in the database, a `KeyError` exception will be raised. Note that database entities may not be defined or changed using this mechanism: this provides read-only access. To change an object, retrieve it using this, then use that object's `store_data()` method, described below.

`delete_object(`*obj_specifier*`)`

[Requires that the database be open with write permission.] Removal of individual objects from the database is performed by this database method. It is called with either an object ID string or an OdbInstance object. If deleting the object does not result in an integrity failure for a linked object, the deletion proceeds and the object is removed from the database, causing this method to return a true value. Such an integrity failure could only be caused by the need to reduce the number of links in the linked object's fields below the minimum number of values threshold for that object's class. Failure to delete the object may cause an `odb.error` or `KeyError` to be raised. The associated value of the exception is either a 3-tuple giving a database error number, `odb` module error, and any error message from the underlying ODB or GDBM libraries, or it may be a simple string. Note that the ODB library currently

provides no way to tell if a storage failure (`IOError` exception) occurred for the object for which deletion was requested or for a linked object. This may change in the future; the second element of the 3-tuple exception value will indicate if the failure may have been due to a linked object at the time the module was compiled.

`has_key(`*object_id*`)`

Determines if an object ID string is valid for this database. If so, a true value is returned, otherwise, false. The *object_id* parameter must be a string.

`keys()`

Returns a sequence object which yields each key of the database. The object provides lazy evaluations whenever possible to increase interactive efficiency. This object supports most non-destructive objects on lists, with the exceptions of concatenation and replication. These operations may be emulated by requesting a slice of the object without specifying the endpoints explicitly. This works because requesting the length of the object will force evaluation to be completed, allowing the length and slice operation to be meaningful. To retrieve a list object with all the keys for the database *db*, evaluate the expression '*db*`.keys[:]`'.

There are two special member variables for this object, named `blocking_factor` and `length`. The `blocking_factor` member defines the minimum increase in the size of the internal memory buffer allocated to store evaluation results. This value must be a positive integer and defaults to 1024. Each key which has been evaluated requires one slot in the internal structure. Decreasing the value of this variable may lead to memory fragmentation. The `length` member variable provides access to the number of keys which have already been evaluated. This is not equivalent to the length of the list of keys: Calling the built-in `len()` function on a keys object forces evaluation of the list, while accessing the `length` member does not.

`new_object(`*class_id*`)`

[Requires that the database be open with write permission.] Creates a new object in the database with the given class ID. The *class_id* must be a string, and must already

be defined in the database. If the class is not defined, a `ValueError` exception will be raised. The new object will be assigned an unused object ID string automatically, but will not be stored immediately since there are no data fields for the object. The return value is an OdbInstance object to which data should be stored immediately using the *obj*`.store_data()` method. An `odb.error` exception will be raised in the event of an internal failure.

Index Handling Methods

`get_index_entry(`*string*`)`

Retrieves an index entry from the database. The single parameter is a text string to search for. If the string is found in the indexes, a tuple is returned with the first element being the string sought, and the second being a list of objects identified in the index:

```
('indexed string',
        ['object_id_0', 'object_id_1', 'object_id_2' ...]
        )
```

Be aware that this list may be empty and may contain objects which have been removed from the database. If the string is not found in the index, the value `None` is returned.

`store_index_entry(`*index_entry*`)`

[Requires that the database be open with write permission.] Adds or updates an index entry in the database. The entry must have the form described for the return value of `get_index_entry()` method, above. No validation of the object IDs specified is performed. The only exceptions raised are the `IOError` and `TypeError` exceptions.

3.4.2 `OdbDatabase` Member Data

The `OdbDatabase` objects offer a few data members as well as the methods defined above. Most of these elements are read-only, and are used to access information regarding

the current state of the database as defined when the object was created. Attributes which provide write access are noted, along with the effects of assigning new values.

`closed`

This value is false if the database is open for any form of access, otherwise it is true.

`current_index_file`

[Requires that the database be open with write permission.] The ODB allows each database to have multiple files for the storage of indexes and data objects (the two uses of files are distinct, and the data does not share files with the indexes). Within each group of files, one file is considered *current*, and is the location in which all new records of that form are stored. This data attribute records the setting of the current file in the index file group, and allows the application to adjust this value as well. The integer in this attribute for the database *db* is in the range `0` ... *db*`.number_index_files - 1`. Attempting to set this to a value outside that range or to a non-integer raises a `ValueError` exception. If an error occurs recording this information in the database, an `IOError` exception is raised.

`current_object_file`

[Requires that the database be open with write permission.] This attribute performs in a manner identical to that defined for `current_index_file`, but for the object file group. The legal range of integer values is `0` ... *db*`.number_object_files` for database *db*.

`filemode`

[Requires that the database be open with write permission.] This member is set when the database is opened, and is used to determine the access control permissions for files added to the database. When database creation is supported directly, this value will be used for that as well. The default value is `0666` (octal), representing global read and write permissions under POSIX compliant operating systems.

`name`

This attribute holds the base value of the current database, excluding any filename

extensions. If the database was opened using an absolute or relative path name, that path information is included.

`number_index_files`

[Requires that the database be open with write permission.] As discussed under the data attribute `current_index_file` above, ODB maintains its data and indexes in two groups of files. This value specifies the number of files in the index group. It can only be affected by calling the `add_index_file()` method. The maximum value is `10`.

`number_object_files`

[Requires that the database be open with write permission.] This attribute specifies the number of object storage files in the database. The maximum value is `10`, and the actual value is affected by the `add_object_file()` method.

`open`

This value is true if the database is open for any access.

`rwmode`

[Requires that the database be open.] This member specifies the permissions string specified when the database was opened. The specification `'rw'` will be used when `'w'` was used in the call to `odb.open()` since both reading and writing are permitted in that mode.

3.5 `OdbInstance` Objects

`OdbInstance` objects represent individual objects from an ODB database. These objects cannot be created directly, but must be provided by a member function of an `OdbDatabase` object. The indexing operator and the `new_object()` method provide objects of this type as result values. Several methods are provided to allow interrogation and update of the object's internal state, which corresponds directly to the contents of the physical database. Update is only allowed if the parent database was opened in read/write mode and remains

open at the time of update.

3.5.1 `OdbInstance` Methods

`field_data()`

Return the current values of all fields as a list of lists. The top-level list contains a single entry for each field ID in the class of the object. If the field has been deleted, the entry for that field is `None`, otherwise it is a list of all values in the field. If a field has no values but is not deleted from the class definition, the entry is an empty list, `[]`, not `None`.

`has_field(`*name*`)`

Determines if this object has a data field named by the given string value. Returns `1` if such a field exists, or `0` if it doesn't.

`store_data(`*data*`)`

This method replaces all field values for an object except for the LINK fields. The parameter *data* should receive a list of the same structure as that returned by the `field_data()` method. A `field_data()` return value can be destructively modified to provide the input to this method if desired: Since the result of the `field_data()` method is returned as a list of lists, rather than tuples, replacing a single value in the result structure is meaningful, and the modified structure may be passed to `store_data()` to be updated in the database. Updating the structure will not create any side effects, and will avoid having to re-build the entire structure.

3.5.2 `OdbInstance` Member Data

All data elements are read-only, and are provided to make available information regarding the source of the object and the type of data it contains.

`class_id`

Returns a string identifying the class to which the data object belongs. This string can be used in a call to the `get_class_info()` method of the parent database to get

additional information on the object's class.

`database`

This is the parent `OdbDatabase` object. It is used extensively internally, and is available to the application as a read-only attribute to ease the use of multiple ODB databases within a single application. Testing for identity of this attribute for different `OdbInstance` objects can determine if both were extracted from the same database.

`object_id`

Returns the object ID of the object as a string. This may be used to retrieve the object again in the future or to create an entry in the Change Dictionary defined in module `Delta`.

Chapter 4
The Data Dictionary

Any effective database management system must provide some method of defining and manipulating the structure of information stored in the database. The description of the database contents and the relationships among the various entities is commonly called the *data dictionary.* While the form of data in ODB databases is reasonably well defined by the structures and field types provided by the base package, a great deal of information about the structure is used by the applications using the system. A general tool for working with ODB databases must provide access to and support manipulation of as much of this information as possible.

The ODB package supports this by defining a mechanism by which database entities may belong to certain *classes* of objects; each class provides a set of *fields* which may take on values of restricted type and number. Every entity in an ODB database belongs to exactly one class. The definition of a class may change over time, with fields being removed or added to the class definition, or with the constraints placed on a field being changed. These operations will not take place often with a database that is in production use, but flexibility in evolving the database schema is very useful for any database application under development, or for a database management system designed for highly dynamic systems.

The *odb*/Tools package provides access to all the facilities offered by the base ODB package, and makes some additional operations normally available only through the `odbms` administrators' tool more readily available for use from within applications. No provision is offered to extend the database beyond what the base package is able to manipulate normally, so no new features are provided by *odb*/Tools but may be provided by applications as needed.

4.1 Overview of the Data Dictionary Facilities

The underlying ODB library provides access to the data dictionary through a number of functions which provide many useful operations on the dictionary but do not always present a consistent interface or coding style. These functions are highly procedural. The *odb*/Tools package provides an object oriented abstraction which operates at a slightly higher level, retaining the ability to perform all the available operations on the dictionary.

The data dictionary allows the application to define classes, or types, of database entities. Each entity can be constrained on the data fields it may contain and the type and number of values each field may store. All classes must define at least one field, but the number of fields which may be defined for each is constrained only by the word size of the host environment and the size of virtual memory.

Class IDs and field names are sequences of lower case alphabetic characters, digits, and the underscore. *odb*/Tools imposes the additional restriction that the first character in the name not be numeric for fields accessed using the shorthand form of field access, as described in Chapter 6, though the more general access method imposes no such restriction. There is no maximal restriction on the length of class IDs or field names, but each name must contain at least one character. Field names must be unique within the enclosing class, and class IDs must be unique within the database.

Each field may hold values of a single type. Type identifiers are given as strings in Python, and are not case sensitive. The defined type specifiers and descriptions of the corresponding data requirements are given below.

`'STRING'`

> String fields are the most general data values available in an ODB database. The content is restricted only in that byte values of zero are not allowed: there are no restrictions on the character set used beyond that, and the value may be arbitrary binary data. Since zero byte values are not allowed in string data, a character set which includes zero bytes, such as the 16-bit Unicode set [Uni91], may not be used.

However, multibyte character sets which do not include zero bytes are allowed. Popular multibyte sets which can be used with ODB include Shift-JIS [Wal94] and UTF-8 [Ted93].

`'DATE'`

These fields are handled as STRING fields within the ODB and the *odb*/Tools package, but are used to store dates from the Gregorian calendar in the format `yyyymmdd`, where `yyyy` is the year with the century specified, `mm` is the month, and `dd` is the day of the month. Data in a DATE field is presented to the client code as a conventional string in the host language (zero-terminated from C, or of type `StringType` in Python). It is the responsibility of the application to format date information correctly. *odb*/Tools will preserve application formatting of these fields, so an alternate format may be used if appropriate.

`'LINK'`

The values of these fields represent a bi-directional relationship between two objects in the database. Each object in the relation contains a LINK value referring to the other. Special operations are provided to manipulate these values; simple assignment semantics do not apply in the current implementation of *odb*/Tools. The values are represented as tuples with the type `(OBJECT ID, IntType)`. The string representation stored in the physical database contains an Object ID concatenated with an ASCII ETX character (byte value 3) followed by a field ID represented as a decimal number.

`'OBJ-ID'`

This type, representing an object ID, can be used to represent a uni-directional link. No special support is provided for this type. The values of these fields are strings representing the unique identifier of a database entity. No assumptions can be made regarding the content of this data, but these values can be tested for equality using the '`==`' operator. Identity tests using the '`is`' operator are not supported.

4.2 Introduction to the Classes

There are three classes which implement the data dictionary interface in *odb*/Tools. The primary interface is defined by `DataDictionary` and is supplemented by `ClassDefn` and `FieldDefn`. These classes are implemented in Python and use the `odb` module to query the state of the dictionary and modify it as needed by the application. Each class is defined in a separate module. In most cases, the application will not need to import any of these directly, or instantiate a `DataDictionary` object. The `Database` class defines the method `DataDictionary` which will supply a data dictionary with the permissions with which the database is opened. When not accessing the `odb` module directly, this is the best method to obtain a handle to the appropriate `DataDictionary` instance.

4.3 Common Methods

There are a number of methods which are shared by all three classes which form the data dictionary interface. These methods are described below, rather than being repeated for each class.

`Flexibility(`*flex*`)`

> This method sets the new value of the flexibility setting to *flex*. Only the boolean value `0` or `1` will be stored; the object passed in will not be held if not one of these two values. The new value is returned.

`Flexible()`

> This method returns whether the flexibility attribute is set for the object. Though only methods of the `FieldDefn` object use this to modify their behavior, the attribute is inherited from each object's parent in the data dictionary representation. The attribute is false by default in the `DataDictionary`, but may be set using the `Flexibility()` method, described above, to allow child objects to inherit this attribute.

`Mutable()`

> This method returns true if update of the data dictionary through this object is

possible. Attempting to modify an object for which this method does not return a true value causes a `DataDictionary.DDError` exception to be raised.

4.4 Odb module `DataDictionary`

The `DataDictionary` module defines operations on the entire data dictionary structure, and allows individual classes to be accessed, as well as providing a facility for adding new classes to an ODB database. The base level of access control on the data dictionary is defined by this class as well.

Two top-level entries are made available in this module. The first is an exception used by all modules which provide the data dictionary support, and the constructor for the `DataDictionary` class.

`DDError`

This exception is used for all failures on the data dictionary objects created by the constructor defined above, including errors in the related implementation classes `ClassDefn` and `FieldDefn`. The associated value for this exception is always a string. The exceptions `TypeError` and `ValueError` may be raised as well; refer to the documentation for each method for circumstances which raise each possible exception.

`DataDictionary(`*odb_database* [, *rwmode*] `)`

The data dictionary itself is constructed from an `OdbDatabase` object to allow both read- and write-access to the physical dictionary, as well as to improve efficiency and eliminate complexity in the high-level `Database` class. The data dictionary object is created as read-only unless both the database was opened with read/write permission and the *rwmode* parameter is specified with a true value. The data dictionary will never be write-enabled if the corresponding database is read-only. Methods are provided to query the permissions on the resulting `DataDictionary` object.

4.4.1 `DataDictionary` Methods

The data dictionary objects returned by the constructor `DataDictionary()` have several methods to query and manipulate the structure of an ODB database. For each of the methods below, attempting to modify the dictionary of a read-only database causes the `DataDictionary.DDError` exception to be raised with a string as an associated value.

`AddClass(`*class_id* [, *name* [, *other_info*]] `)`

New class definitions may be added to the database with this method. If `Mutable()` is true of this database and *class_id* represents a unique, unused class ID, a new `ClassDefn` object is created and returned to the user. The new class definition is different from other `ClassDefn` objects in that full flexibility to manipulate the definition is enabled. See section 4.5 for more information about full flexibility. The optional *name* and *other_info* parameters default to empty strings.

`Class(`*class_id*`)`

Definition objects for individual objects may be retrieved using this method. The *class_id* parameter should contain the abbreviated class ID; the class name is not usable since it is not required to be unique across a database. The class ID must be a string containing no upper case alphabetic characters. The definition object returned is an instance of the `ClassDefn` class defined below. The returned object will inherit the access permissions of the parent database. This method will raise a `TypeError` if *class_id* is not a string, and a `DDError` if the class ID requested does not represent some class in the database.

`Classes()`

This method returns a list of all classes defined in the database. Each entry in the list is a class ID string which may be used to return a definition object for the corresponding class using the `Class()` method of the dictionary. This method never raises an exception.

`HasClass(`*class_id*`)`

The `HasClass()` method determines if a particular class ID is used by the database, returning a true value if the ID has already been assigned. This method will only raise `TypeError` if the *class_id* is not a string.

4.5 Odb module `ClassDefn`

The `ClassDefn` module provides access to information concerning an entity class in an ODB database. There is only one new public object defined in this module. This is the constructor for `ClassDefn` objects, and should normally only be called by the `DataDictionary` object methods.

`ClassDefn(`*datadict, class_id*`)`

The `ClassDefn` constructor takes the parent `DataDictionary` object and the class ID string as parameters. The resulting class definition inherits the dictionary's access permissions and flexibility setting. Calling the constructor directly will fail with a `DataDictionary.DDError` exception.

The *flexibility* setting of a class definition is largely a topic dealt with internally to the classes which implement the data dictionary; the user should never care about it. The only classes for which the setting has any meaning are the `ClassDefn` and `FieldDefn` classes, which normally just inherit the setting from their parent. The setting is always false for `DataDictionary` instances as currently implemented. Newly created class and field definitions will be granted "flexibility" to make changes to themselves which would allow the potential invalidation of database objects were any in the database. While it is impossible to completely protect the database from hostile updates to the data dictionary, this makes such changes harder to bring about accidentally. The `Flexible()` and `Flexibility()` methods, described in Section 4.3, are used to query and manipulate the flexibility setting.

The class definition object returned by a call to `DataDictionary.AddClass()` will receive the flexibility attribute from the parent data dictionary.

4.5.1 `ClassDefn` Methods

`AddField(`*name* [, *ftype* [, *min, max*]] `)`

New field definitions may be added to the database with this method. The new field will be automatically assigned a field ID and given the name specified by *name*. The initial type of the field is STRING if another type is not specified in *ftype*, and there may be exactly one value in the field unless the *min* and *max* values are specified. The minimum number of values, if specified, must be a non-negative integer, and the maximum number of values must be a positive integer or `None`. The `None` value is used to indicate that there is no upper limit on the number of values assigned to the field. An instance of the `FieldDefn` class is returned to represent the field and allow update. The new field definition object is given the flexibility attribute described above; details of behavioral changes are given with the descriptions of the `FieldDefn` methods affected. Creating a LINK field may only be done by calling this method to create a field; a field may not become a LINK after creation, nor may it loose this status. Refer to the documentation for the `FieldDefn.Type()` method for more information on field type manipulations.

`AlternativeID(` [*new_id*] `)`

The alternative ID of an entity class provides a numeric alternative to the class ID string normally used to identify a class. The alternative ID must be a non-negative integer and is required to be unique within a database. Calling this method with an empty parameter list returns the current value of the alternative ID. Passing a single parameter of the required type sets the value if uniqueness is maintained and returns the new value. If the requested ID is already in use by another class in the database, a `DataDictionary.DDError` exception is raised.

`Field(`*field_name*`)`

Definition objects for individual fields may be retrieved using this method. The *field_name* parameter should contain the numeric field identifier or the field name

as a string. Both of these attributes are unique within a class definition. The field name is considered the "normal" value to be passed in, since it is an effective way to access the field and allows more readable code. For interactive applications, this is most likely the information the user is able to supply.

`Fields()`

This method returns a list of all field names defined within the class. Each entry in the list is a string which may be used to return a definition object for the corresponding class using the `Field()` method of the class definition. This method never raises an exception.

`HasField(`*field_name*`)`

The `HasField()` method determines if a particular field name is used by the class, returning a true value if the name has already been assigned. This method will only raise `TypeError` if the *field_name* is not a string.

`ID()`

This method returns the ID string of the class. There is no way to set this ID once the class has been created using `DataDictionary.AddClass()`.

`Name(` [*new_name*] `)`

Each entity class has a name, typically used to indicated in human-readable form the class of external objects represented by entities of that class. The names are not assured to be unique across the database, but are useful for associating database entities with real-world counterparts. Calling this method with no parameter returns the name of the class. Invoking it with a single string parameter sets the class name and returns the new value.

`OtherInfo(` [*new_info*] `)`

Each class defined in an ODB database allows a string value to be added to the class definition. This field is not used by the database library, but is maintained to allow information to be appended if needed. The value has the same restriction that all string values do under ODB: that no zero bytes be part of the string data. This

method allows *odb*/Tools applications to query and set the value of this field. Calling this with no parameters returns the current value of this field, and providing a string parameter sets the value and returns the new value.

4.6 Odb module `FieldDefn`

The `FieldDefn` module provides access to information concerning a field of an entity class in an ODB database. As with the `ClassDefn` module, here is only one new public object defined here. This is the constructor for `FieldDefn` objects, and should normally only be called by the `ClassDefn` object methods.

`FieldDefn(`*classdefn, field_id*`)`

The `FieldDefn` constructor takes the parent `ClassDefn` object and the field ID number as parameters. The resulting field definition inherits the dictionary's access permissions and flexibility setting. Calling the constructor directly will fail with a `DataDictionary.DDError` exception.

The flexibility setting of a field definition is used to allow the field to be modified in ways that increase the restrictions placed on the field. Normally, only operations which decrease the restrictions placed on a field are permitted. Each method below details the effect of the flexibility setting on their operation. The methods `Flexible()` and `Flexibility()`, described in Section 4.3, may be used to query and manipulate this setting if necessary.

4.6.1 `FieldDefn` Methods

`ID()`

This method returns the numeric ID of the field. There is no way to set this ID once the class has been created using `ClassDefn.AddField()`.

`Max(` [*new_max*] `)`

Returns the maximum number of values allowed in the field when called with no parameters. If *new_max* is provided, that value must be either a positive integer or

`None`. The return value is the value after any changes have been made.

The flexibility attribute assigned at the creation of the `FieldDefn` object is used to determine the range of possible new values. The default, without flexibility, is from the current setting through the "no limit" value symbolized by `None`. With flexibility enabled, any positive integer will be accepted, as well as the `None` value. This prevents the invalidation of existing database objects by reducing the number of allowed values in the field below the number already stored.

`Min(` [*new_min*] `)`

In a fashion similar to that of the `Max()` method described above, the `Min()` method provides access to the minimum number of field values allowed. Calling this method without any parameter returns the current value, which will always be a non-negative integer. Providing a positive integer as a parameter sets the value of the attribute. This value must be explicit; there is no semantic equivalent of the `None` value used with the `Max()` method. The new value is returned.

As with `Max()`, the flexibility attribute affects the range of permissible values. Without flexibility, the range extends from zero to the current setting. This is done to prevent increasing the minimum number of field values beyond that available in the database. With flexibility enabled, any integer value less than or equal to the maximum number of values is allowed.

`Name(` [*new_name*] `)`

Each class field has a name which is normally used to access it. Unlike class names, field names are unique within the enclosing object (the class). Calling this method with no parameter returns the name of the field. Invoking it with a single string parameter sets the field name and returns the new value if the new value is unique. If not, the `DataDictionary.DDError` exception is raised.

`Range(` [*new_min, new_max*] `)`

This method, when called with no parameters, returns the tuple `(Min(), Max())`.

With two parameters, each is interpreted as described for the parameters of the same names in the methods `Min()` and `Max()`. All rules for the allowable changes hold as specified for those methods, including the effects of the flexibility setting.

`Type(` [*typespec*] `)`

The data type of the field may be determined by calling this method with no parameters. The string value returned is one of the type identifiers described above. Calling this method with a single string field type identifier sets the type of the field for read/write databases. The only restriction is that fields may not be changed to or from the LINK type. Any attempt to do so will raise a `DataDictionary.DDError` exception. The flexibility setting does *not* affect this operation in any way. Of those type changes which are allowed, no attempt is made to alter the data of the field values in the database entities; this remains the responsibility of the application.

Chapter 5
Accessing the Database

The `odb` module, discussed above, provides only a minimalist's interface to the capabilities of the ODB database library. While the functions of the library are accessible, if indirectly, there is no real improvement over the base library. This module, in conjunction with the `DataDictionary` class discussed in an earlier chapter, offers a much more robust and effective interface for the object-oriented application.

5.1 Odb module `Database`

As with other high-level modules in the *odb*/Tools package, this module provides an object class and an exception which is raised on certain error conditions. The class is used to represent a complete database, but does not provide direct manipulation of the data dictionary.

`Database(`*name* [, *rwmode* [, *filemode*]] `)`

This is the primary point of access to an ODB database for application programmers. This class is responsible for providing a Python object representation of the database over the `odb` module, providing better object-oriented support for the database as a whole. The *name* parameter should be the base name for the database, excluding any file name extension and including any needed relative or absolute path specification. The optional *rwmode* parameter is used to specify whether the database should be opened in read-only mode or should allow update; the default value is read-only. Valid mode specifies include `'r'` for read-only mode and code'rw' and `'w'`, both indicating read-write mode. The specifier `'c'` indicates that the database should be created if

it doesn't exist and opened for update. The last specifier, `'n'`, indicates that the database should be reset to empty or created if it doesn't already exist, and then opened for reading and writing. The third parameter, *filemode*, is used to specify the access control permissions used to create new database files when needed. The default value is `0666` masked by the user's current setting for `umask(2)`.

`DatabaseError`

This exception is used to indicate failures specific to the database. The associated values for this exception are strings describing the cause or type of failure. Specific circumstances under which this exception is raised are detailed by the various methods of the `Database` class.

5.1.1 `Database` Methods

The `Database` class provides access to most of the general facilities of the ODB library. Since such a large portion of the functionality is made available through this class, the methods will be divided into functional groups in the list of method definitions which follows.

General Methods

`AddIndexFile()`

This method mirrors the `add_index_file()` method of the `OdbDatabase` object, allowing the addition of new index files to the database. If successful, the method returns the number of index files in the database, otherwise an exception is thrown. If the database was opened without write permission, the exception raised is `DatabaseError`, else the exception may be `odb.error`, indicating that the limit on index files has been reached, or `IOError`, indicating that the GDBM library was not able to create the new file. For all exceptions, the associated value is a string describing the failure. Successful completion does not install the new file as the default in which new index entries are stored; that can only be affected by setting the `current_index_file`

member variable.

`AddObjectFile()`

In a manner similar to the `AddIndexFile()` method described above, this method affects the number of files used to store object data in the database. When successful, this method adds a data file in which entity data is stored in the database. The exceptions raised match those described for `AddIndexFile()` exactly; refer to that description for details. On success, the return value is an integer giving the number of object files currently available for the database. The new file is not made the default for storing new object instances; the `current_object_file` member variable must be used to control that setting.

`Close()`

This method performs an implicit `Rollback()` and closes the database, preventing any further use of derived objects and discarding all changes made since the database was opened or the most recent `Commit()` was executed.

`Commit()`

As with all database management systems, there must be some method by which the end of a transaction may be designated. With *odb*/Tools, this takes the form of executing this `Commit()` method to terminate the current transaction and accept the changes stored in the change dictionary. Refer to the `Rollback()` method for information on not accepting changes.

`DataDictionary()`

The data dictionary of a database opened using this class cannot be created directly using the `DataDictionary()` constructor since the corresponding `OdbDatabase` object is not available. This method allows the data dictionary to be made available. The `DataDictionary` object returned has the same permissions as the database: if the database is opened with write permission, the data dictionary may be manipulated as well as interrogated. Data dictionary changes are not stored in the data dictionary, but take affect immediately. Multiple calls to this method will return the same instance of

the data dictionary, so it is safe to use in the face of changes made to the dictionary.

`Mutable()`

This method returns true if the database was opened in a mode allowing updates. The modes passed to the constructor which open a mutable database include `'c'`, `'n'`, `'rw'`, and `'w'`.

`Rollback()`

The `Rollback()` method performs the counterpart operation of the `Commit()` method, reseting the change dictionary and related state information regarding changes made to the set of database entities in the database. The database will hold the same data after calling this method as it did before either the most recent call to `Commit()` or the opening of the database, whichever came last. Changes to the data dictionary or the indexes are not affected by this method; only changes to entity data.

Entity Handling Methods

`[`*object_id*`]`

If the object specified by *object_id* exists in the database, this will retrieve it. The class of the object returned is determined by the access permissions on the database. If the database is opened read-only, an `ObjectInstance` is returned, otherwise an `ObjectIndirect` is returned. Both of these classes support the object protocol, so there should be no affect on application code, unless the ability to modify an object is assumed.

`has_key(`*object_id*`)`

The `Database` objects behave in some ways like a Python dictionary object in how data is extracted from the database. The `has_key()` method has exactly the same semantics as it does for the built-in dictionaries. Given a string argument, this method returns true if the string is a valid object ID in the database, whether the object identified by the string exists in the physical database or is a new object created by `NewObject()` but not yet stored by a `Commit()` operation. For all other values of

object_id, this method returns a false value.

`keys()`

This method also emulates the method of the same name on built-in dictionaries. A list of all valid object IDs for this database is returned. Note that the related methods of the built-in dictionaries, `items()` and `values()`, are not implemented for the `Database` class to avoid exhaustion of memory resources.

`NewObject(`*class_id*`)`

The `NewObject()` method creates a new object in the database with the object class specified by *class_id*. If *class_id* is not a valid class ID for the database, a `DataDictionary.DDError` exception is raised. If successful, a new object is returned of the specified class as an `ObjectIndirect` instance. The new object will have no data associated with it until the application adds it. This implies that the object cannot be saved during a `Commit()` operation until at least minimal data required by the entity class has been provided.

`RemoveObject(`*object_or_id*`)`

This method deletes an object from the dictionary. Links to related objects are removed as well. If no object exists in the database which corresponds to the object ID passed in, or if the object passed in does not belong to this database, a `DatabaseError` exception is raised.

5.1.2 `Database` Member Data

The `Database` class offers several data elements which may be used to gain information about the database being manipulated. These elements all mirror elements of the underlying `OdbDatabase` object, and provide the same forms of access with identical semantics. This list is provided here for completeness.

`closed`

This value is false if the database is open for any form of access, otherwise it is true.

`current_index_file`

The ODB allows each database to have multiple files for the storage of indexes and data objects (the two uses of files are distinct, and the data does not share files with the indexes). Within each group of files, one file is considered *current*, and is the location in which all new records of that form are stored. This data attribute records the setting of the current file in the index file group, and allows the application to adjust this value as well. The integer in this attribute for the database *db* is in the range `0` ... *db*`.number_index_files - 1`. Attempting to set this to a value outside that range or to a non-integer raises a `ValueError` exception. If an error occurs recording this information in the database, an `IOError` exception is raised. Since adjusting the current file for the index and object groups is not automatic under ODB, manipulation of this setting should be considered an administrative operation. The database must be open for a reference to this variable to be valid, and update is only permitted if the database is open with write permission.

`current_object_file`

This attribute performs in a manner identical to that defined for `current_index_file`, but for the object file group. The legal range of integer values is `0` ... *db*`.number_object_files` for database *db*. The database must be open for a reference to this variable to be valid, and update is only permitted if the database is open with write permission.

`filemode`

This member variable records the *filemode* requested when the database was opened. If none was specified, the default value (`0666`) is returned.

`name`

This attribute holds the base value of the current database, excluding any filename extensions. If the database was opened using an absolute or relative path name, that path information is included.

`number_index_files`

This attribute specifies the number of index storage files in the database. The maximum value is given by `odb.MAX_NUMBER_INDEX_FILES`, and the actual value is affected by the `AddIndexFile()` method. This variable is read-only and requires the database to be open.

`number_object_files`

This attribute specifies the number of object storage files in the database. The maximum value is given by `odb.MAX_NUMBER_OBJECT_FILES`, and the actual value is affected by the `AddObjectFile()` method. This variable is read-only and requires the database to be open.

`open`

This value is true if the database is open for any access.

`rwmode`

This member specifies the permissions string specified when the database was opened. The specification `'rw'` will be used when `'w'` was used in the call to `odb.open()` since both reading and writing are permitted in that mode. The only difference between `rwmode` on a `Database` instance and an `OdbDatabase` instance is that `Database.rwmode` supports and may evaluate to the `'c'` and `'n'` access specifiers. Refer to the documentation on the `Database()` constructor for information on these specifiers. This variable is read-only and requires the database to be open.

Chapter 6
The Object Protocol

The *odb*/Tools package includes several objects which provide an interface which makes them look and act like objects from the database. This interface is called the *Object Protocol*. Restricting general access to an object to the methods defined by this interface allows an application to operate effectively regardless of the underlying object layers. This protocol has been used to provide an interface to the actual database objects, to elements in the Change Dictionary, and to various proxy objects used at various points.

This protocol is ensured to operate on all program objects produced by the Python support interface to the ODB database library. When the specifications set forth below are consistently adhered to, each object supporting the protocol is ensured to be maintained correctly without introducing data integrity or program failure. Objects which are read-only will correctly maintain their state, and read/write objects will ensure that all program objects representing the same database entity will report the correct state.

A second protocol, named the *Field Protocol*, defines the public interface to the objects representing individual fields within objects. This is required in order to handle fields with the same transparency as the Object Protocol allows for objects.

There are two important points to keep in mind when working with objects supporting the protocols. Though all supporting objects support the same protocols and provide the same effective semantics for each method defined by the protocols, the implementations of these methods may vary significantly in operation, and can have side affects which affect future operation of the objects.

For example, if an object represents a change record from an object in the physical database, data which is not part of the delta may not be available. If a value was appended

to a field, the initial values are not defined for the delta object. Fields which have not changed may not be represented at all, but may be added dynamically when referenced. The protocol defines means by which availability of data can be determined to ensure that behaviors may be predicted or handled meaningfully given the application does not make assumptions but is willing to query the state of an object before performing operations.

The second point to remember is that the `OdbInstance` objects defined in the low-level support for the ODB library interface are not intended for general use and do not support this protocol.

6.1 Object Protocol Methods

The Object Protocol is defined by a set of methods provided by objects supporting the interface. These methods are defined in this section.

`ClassID()`

> This method returns a string naming the class ID for the database entity the object represents. It is ensured to be a valid ID. No additional information about the class should be considered valid if derived from the object providing this ID, with the sole exception that field names returned by the `Fields()` method, defined below, will be valid field names for the given class. Canonical information regarding a class can only be retrieved from `DataDictionary` objects.

`Field(`*name*`)`

> The `Field()` method is a generalization of the shorter access method described below for retrieving individual fields from the object. Though the short-hand method is expected to be the most-used mechanism due to the brevity and clarity of the notation, it does restrict field names to conform to the naming conventions of Python identifiers. This access method imposes no such restrictions, and allows any legal field name permitted with the ODB library, including field names that embed punctuation and whitespace. Only the `ObjectProtocol` class needs to implement this method;

derivative classes need only define the `__getattr__()` discipline, as described below.

`Fields()`

Returns a list of the names of all data fields present in the object. This does not indicate that data is available for all fields named in the list. Fields may exist in the class of the object without being present in the list. Inclusion signifies that the field name holds meaning for the object in its current state.

`HasField(`*field_name*`)`

The `HasField()` method returns a truth value indicating whether a field named by *field_name* exists in the object. The expression *ob*`.HasField(`*field_name*`)` is defined to be equivalent to the expression *field_name* `in` *ob*`.Fields()`. A false value does not indicate that *field_name* is not a valid field name in the object's class.

`ObjectID()`

This method returns the object ID of the object.

`Mutable()`

This method returns a true value if the object may be modified in any way. All objects are readable.

field_name

A reference to a field name will return a representation of the field, with operations appropriate to the semantics of the generating object. If the protocol-compliant object is read-only, modifying the field representation will either raise an exception or be ineffective at changing the state of the database entity. A read/write object will generate a representation which will propagate changes to the base entity. This method of accessing an individual field is considered a short-hand form of the `Field()` method. For any given object *obj*, the expressions *obj.field_name* and *obj*`.Field('`*field_name*`')` are identical when *field_name* is a valid Python identifier. When the field name is not a valid identifier, the `Field()` method must be used to access the field representation.

This aspect of Object Protocol behavior must be implemented through the standard `__getattr__()` and `__setattr__()` disciplines. Recall that field names of ODB en-

tities must be in lower case; the behavior of these methods is allowed to vary for any attribute name which is contains upper-case characters and, therefore, is not a potential field name. If the name is potentially valid, the object must define an appropriate behavior or throw a `DataDictionary.DDError` exception. The associated value of the exception is not defined or restricted, and should be meaningful to allow debugging of client code. Exception values must be documented by the protocol provider in the `__getattr__()` discipline's `__doc__` attribute as well as in other published documentation.

6.1.1 Supporting Objects

The objects listed here support the protocol. Each provides a definition of the interface which implements the appropriate meaning of the methods to provide conformance and data integrity in the context of their functional specifications.

`ObjectProtocol()`

This is the base class which provides the default Object Protocol implementation. Each subclass which supports the protocol should inherit this class or one of its descendents, and override the various methods as appropriate. Refer to the source code for details on which methods need to be overridden or supplemented, and which can be considered complete. This class should not be instantiated.

`ObjectDelta()`

This object represents the changes to an individual object in the database. The original data is not maintained and is not available in objects of this class.

`ObjectIndirect()`

The `Database` class provides an interface to an ODB database buffered by a change dictionary object when the database is opened in read/write mode. The `ObjectIndirect` class provides an interface which directs updates to the change dictionary to allow a delayed commit or rollback action to take place. This object provides a composition of a single pair of `ObjectInstance` and `ObjectDelta` objects.

`ObjectInstance()`

The base client-level interface to database entity objects from the ODB database is implemented in this object. This object provides an implementation directly bound to the database entity. These objects are always read-only as the data they contain never changes. Database update is handled by the `Database.Commit()` method and the Change Dictionary.

6.2 Field Protocol Methods

The Field Protocol is defined by a set of methods provided by objects supporting the interface. These methods are defined in this section, and correspond directly to standard Python operations on mapping objects, of which all ODB field objects are a form. A number of additional methods are provided as well. Note that the `__len__()` discipline is implemented by all supporting objects. This allows the standard Python `len()` function to operate meaningfully on all field objects.

`append(`*new_value*`)`

The `append` method acts as does the method of the same name for Python's built-in list objects. For read/write objects, the *new_value* is appended to the list of values with an index one higher than the highest existing index. If the field object is read-only, an exception of type `Database.DatabaseError` is raised.

`FieldID()`

This function returns the numeric ID of the field this object represents. Information about the field can then be retrieved from a `DataDictionary` object associated with the parent database. The information need not be provided by the field object.

`has_key(`*index*`)`

This method returns true if the non-negative integer corresponds to a value entry in the field object, otherwise, false is returned.

`items()`

The return value of this method consists of a list of `(`*key*`,` *value*`)` pairs, with a pair included in the list for each element of the return value of the `keys()` method. The keys are sorted in ascending order.

`keys()`

The list of valid field value indexes is returned by this method. The list may be empty. For instance, if an object *obj* in the database has three values in the `'name'` field, the expression *obj*`.name.keys()` evaluates to `[0, 1, 2]`. Similarly, if *obj* is a `FieldDelta` instance with one value, the same expression may evaluate to `[12]` if that value is to become the thirteenth entry in the database.

`LinkTo(`*target_field*`)`

Links between objects are created with this method. A link is formed between the parent object of this field and the object of the target field, using these fields as endpoints, thereby defining a relationship between the objects. Both fields are type checked to be of type LINK, and existing links which are identical to the new link are not replaced. This method raises a `Database.DatabaseError` exception in classes which are read-only or are not equipped with access to verify the integrity of the link. For fields not of type LINK, a `DataDictionary.DDError` exception is raised.

`Mutable()`

This method returns a true value if the field values may be modified in any way. All field objects are readable.

`ObjectID()`

This method returns the object ID of the parent object.

`RemoveLinkTo(`*target_field*`)`

The `RemoveLinkTo()` method is the inverse of the `LinkTo()` method: it removes a link from one field to another. Both the source and target fields are type checked, and the link they represent is verified to exist. If either field is not of type `'LINK'`, a `DataDictionary.DDError` exception is raised. If the link does not exist, a

`Database.DatabaseError` exception is raised.

`values()`

This method returns a list of just the values in the field, in the same order as the corresponding indices from the `keys()` method. Essentially, the `keys()` and `values()` methods provide the "sides" of the tuples returned by `items()`.

`[`*index*`]`

Indexing notation is used to retrieve individual elements from a field representation. The index must be a non-negative integer represented in the return value of `keys()`; if it is not, an `IndexError` exception is raised. The `__getitem__()` discipline must be used to implement this. For field objects with write permission, the `__setitem__()` discipline must be implemented to allow assignment to index expressions (i.e., '*obj*`.name[`*index*`] = 'new value'`'). For read-only objects, defined failure semantics must be provided and documented.

Using the generalized disciplines allows access to field values to be fairly transparent for both reading and writing values. The following example stores the old value of a field entry and stores a new value in its place.

```
#  'db' is an open database with write-permission.
#
obj = db['oid']
old_name = obj.last_name[0]
obj.last_name[0] = 'Drake'
```

6.2.1 Supporting Objects

`FieldProtocol(`*...*`)`

This is the definitive implementation, providing a common base which other supporting classes should inherit from even if all method implementations are being replaced. It holds the same relationship to the Field Protocol that the `ObjectProtocol` class does to the Object Protocol. Source comments indicate which methods must be overridden and which should never be overridden. Many methods may be overridden to

provide performance benefits.

`FieldDelta(...)`

The Change Dictionary must represent the fields of an object in a dynamic form while maintaining transparency. These objects may be created any time a database entity is referenced, including through the `ObjectIndirect` and `FieldIndirect` objects. These objects will allow assignments to non-existent field value positions. For example, if the `keys()` method evaluates to `[0, 1]`, assignment to *field*`[2]` is legal, and after which `keys()` evaluates to `[0, 1, 2]`.

`FieldIndirect(...)`

This object is produced by the `Database` class when an object from a read/write database is requested. It functions by composing two other Field Protocol providers and dispatching operations appropriately.

`FieldInstance(...)`

The `FieldInstance` class is used to define read-only objects directly from the OdbInstance objects generated from the physical database. These objects are read-only to protect the database from inadvertent change, causing updates to be directed to the Change Dictionary through the `FieldIndirect` class.

6.2.2 Iterating Over Field Values

There are two different ways to approach iterating of all the values of a field, each with multiple expressions which can be used to implement them. All of these use a '`for`' loop to handle dispatching each field value to the operation being performed, and have quite similar structure. The most general mechanism is to use the `items()`, `keys()`, or `values()` methods explicitly:

```
for key, value in obj.field.items():
    print 'db[' + obj.ObjectID() + '].field[' + `key` + '] =', value
```

In this example, the `items()` method returns a list of two-element tuples, each consisting of an index and the field value at that index. Using two variables to hold the loop value

forces the tuple to be split, with each variable receiving one value on each iteration through the loop. For each value with a valid entry, the loop prints a statement of the value. If the `key()` or `values()` methods were used, there could only be one loop variable, since the elements of the lists those methods returns are not tuples, and cannot be split.

The implicit approach to iteration over field values can also be effective, but is only recommended for objects which have not been modified, such as those extracted from a read-only database. This mechanism operated in much the same way, but imposes a restriction due to the use of Python's internal list iteration structures. Here's an example performing an equivalent operation as the previous code snippet, but using the alternate expression:

```
key = 0
for value in obj.field:
    print 'db[' + obj.ObjectID() + '].field[' + `key` + '] =', value
    key = key + 1
```

There are two aspects of this example which are immediately notable. There can only be one loop variable since the values pulled from the implied list are individual values, not tuples. Also, there is no explicit list expression following the field name; the "list" which is implied is the field object itself. A brief explanation of the implied list is in order. Each time the Python interpreter evaluates a `for` *var...* `in` *expr*`:` statement, it does so by subscripting the value of *expr* by an integer starting at zero and incremented by one each time through the loop. The loop terminates when an `IndexError` is raised. This error is masked by the interpreter, and the loop terminates. All other exceptions are passed on to the running application. For a list object, valid subscripts are always zero through `len(`*list*`) - 1`. For a Field Protocol object, however, this is not necessarily the case; indexes may be skipped. For instance, if `obj.field.keys()` yields `[0, 1, 3]`, indicating that a value with index 2 existed but has been deleted, the example above might yield the output:

```
db['oid'].field[0] = This is the first value.
db['oid'].field[1] = And this is the second.
```

Note that the value for `db['oid'].field[3]` is suspiciously absent; this is because the evaluation of `obj.field[2]` raises an `IndexError`, terminating the `for` loop. For unchanged

objects, we know this problem will not occur, since the set of keys any field will be contiguous and start at zero when extracted from the database. Remember that an empty string may be the actual value, so do not confuse that with the lack of a value.

Chapter 7

Iteration and Traversal Mechanisms

In building database applications, there are a number of functions which are used in manipulating the database above and beyond simply updating the data or creating or removing entities. In particular, we must be able to *iterate* over the data, or *traverse* the database. Since these are such common and general operations, we should be able to generalize them effectively. These chapter discusses the object classes provided to support these mechanisms and useful programming idioms which enable the programmer to use the efficiently.

There are two important aspects of traversing a database, whether it be a simple sequential iteration or a more elaborate mechanism. These basic components are the underlying traversal mechanism itself and the means to make decisions, either to determine the path to take through a database or which entities are selected as input to some operation. Within the context of the *odb*/Tools, these components are called *Iterators* and *Predicates*, and are implemented as a set of classes which maintain various elements of state and perform independent actions. This chapter will discuss the Iterators, and the next will discuss Predicates, which will be introduced here only enough to explain the interaction between these two components.

7.1 `Iterators` Module

odb/Tools provides two fundamental iteration mechanisms which may be combined if needed to form more complex traversals of the database. The two basic mechanisms provided are controlled by two classes, named `FlatIterator` and `DepthIterator`. The former

provides sequential access to all entities in the database, and the later provides a depth-wise traversal across all LINK fields in a connected component of the database starting at a single node. Combining the two mechanisms can allow a depth-first traversal of all components in the database.

With only these two mechanisms, only limited encapsulation is achieved for sub-setting the database at a high level. There remains a need to select a subset of the result of each traversal to allow operations to be readily mapped to the portions of the database to which they apply. This selection mechanism is provided by a set of predicates which are used to determine which nodes to include in the result of the traversal process. The predicates are described in detail in the following chapter, which also includes information on creating new predicates to supply application-specific functionality. For the remainder of this chapter, it is sufficient to know that selection predicates are used to determine which nodes are included in the result set, and that they can have side affects in the global context.

Depth-first traversals require additional treatment as well. Where selection predicates determine which nodes are selected for inclusion in an iteration, a method is needed to determine which links between entities are traversed. A similar mechanism, using objects called *traversal predicates*, has been established to satisfy the requirement for such a facility. These predicates receive information regarding the type of node at each endpoint, and which fields within each are used to establish the link. As with selection predicates, these predicates be create global side affects. Detailed information on the defined traversal predicates and on extending the set of predicates is available in the following chapter.

7.2 Sequential Iteration

A single iteration primitive is provided to pass over all entities in the database without regard for the structure of links between the various entities.

`FlatIterator(`*database* [, *selector*] `)`

Construct a sequential iterator on the database object *database*, using the selection

predicate specified by *selector*. If not predicate is specified, all entities will be selected. The object constructed responds to simple subscripting operations, though not slices, and should be used as the source object in a '`for`' loop. Explicit use of the subscript operator ('`iter[5]`'), but is unlikely to be useful. Indexing by a non-integer raises a `TypeError` exception, and a value which is out of bounds causes an `IndexError` exception.

`FlatIterator`s are simple objects with little to offer other than the act of iteration, but provide for this aspect of handling a database quite effectively. For example, many typical applications will use segments similar to the following:

```
from Database  import Database
from Iterators import FlatIterator

db   = Database('name', 'r')
pred = create_my_selector()
iter = FlatIterator(db, pred)

for obj in iter:
    perform_operation(obj)
```

Other constructs which work with general sequences typically may be used with iterators as well, and are especially intuitive with the `FlatIterator` objects. For instance, the following code fragment creates a list of all the class IDs used in a database, with one entry in the list for each entity in the database. Sorting the list allows a little more code to determine how many entities in the database are of each class.

```
list = map(lambda obj: obj.ClassID(), FlatIterator(db))
list.sort()

while list:
    cls = list[0]
    list.reverse()
    pos = list.index(cls)
    list.reverse()
    num = len(list) - pos
    print num, 'instances of class "' + cls + '"'
    del list[0:num]
```

Fortunately, there are better ways of doing this. See the `CountPredicate` example in Section 8.3.2 for a way to do something very similar in a more general way.

7.3 Depth-first Traversal

The second primitive iteration allows the programmer to perform a depth-first search starting from a particular node. This allows a subgraph of the database to be explored based on the relationships formed by the values of the link fields present in the search root and connected nodes.

`DepthIterator(`*database, start* [, *selector* = *None* [, *traversal* = *None*]] `)`

> Construct an iterator object which responds to subscripting operations to provide access to each object found matching the selector specified by *selector*. If *selector* is not specified, all entities are selected. The *traversal* parameter is used to control which links are followed. This parameter must be omitted or be a traversal predicate as described in the following chapter. If omitted, all links are traversed. The predicates determine which links are followed based on the type of relationship indicated by the link. The relationship is specified using the class and field definitions of each end of the link.

Iterators constructed using this class can visit all database entities in a connected component of the database graph. Since the root from which the traversal proceeds is specified, the root is not visited as the first node; links are simply traversed from the root. Any operations which must be applied to the root should be applied prior to traversal, after the traversal, or as a result of the root being reached during the traversal. To ensure that a node of the graph is only visited once, the selection predicate `AcceptOnce` is provided. This predicate is described in the following chapter.

It is important to note that the iterators created with this class reach only a single connected component of the database. To visit each node in all connected components, two iterators should be active: one to provide complete coverage of the database, and one to

descend into each subgraph. The following code fragment demonstrates this technique. A `FlatIterator` is used to ensure that all components are visited, and a `DepthIterator` is created to traverse each component.

```
    from Database   import Database
    from Iterators  import FlatIterator, DepthIterator
    from Predicates import AcceptOnce

    db   = Database('database', 'r')
    once = AcceptOnce()
    flat = FlatIterator(db, once)

    for obj in flat:
        do_root_activity(obj)
        for dep in DepthIterator(db, obj, once):
            do_other_activity(dep)
```

7.4 Using Iterators Efficiently

Often, a database operation requires that some statistic be collected on the portion of the database to be operated on in order to make decisions regarding the operation to be performed. The operation, in this case, can only be carried out after the statistic has been collected. *odb*/Tools iterators support this model of working very effectively. By storing the iterator in a variable, the entire sequence of selected database entities may be re-used without performing a new selection operation or requiring the programmer to record the sequence of object IDs during the traversal. Consider the following fragment.

```
import Database, Iterators, Predicates

db   = Database.Database('mydata', 'rw')
pred = Predicates.ObjectIsClass('dc')
iter = Iterators.FlatIterator(db, pred)

for obj in iter:
    collect_statistic()

if some_condition():
```

```
#  this does not do a new search:
for obj in iter:
    do_some_operation()
```

The function `collect_statistic()` can collect information and measure some attribute of the database. The function `some_condition()` can query some condition based on the information collected during the first iteration and return a boolean value. If that condition is true, the second iteration is executed without performing the original search a second time: the original results have been stored in the iterator for efficient re-use.

Chapter 8

Selection and Traversal Predicates

The iteration mechanisms described in the previous chapter require a means of determining membership in the set of objects to yield, as well as how to proceed in the depth-first iterator. As described, the use of *selection* and *traversal predicates* allows the application to describe the choices to be made independently of the iteration methods. This chapter describes the predicates provided by the *odb*/Tools package and how to define additional predicates.

Predicates operate by mapping their input parameters to truth values. A true value indicates that the predicate supports the inclusion of the input set into the result set, and a false value asserts the predicate's election to disregard the input. Combining simple predicates using logical composition allows complex decisions to be made using this mechanism.

Predicate implementation may take the form either of a function or an object class. The concept of a function predicate follows the commonly held idea of a function supported by most programming languages and used regularly by all programmers who intend to keep their jobs. The object approach involves defining classes which support an overloaded function call operator via the `__call__()` discipline. Using objects allows internal state to be readily maintained. This approach is taken for all predicates defined in the *odb*/Tools package. Section 8.3 discusses ways of creating new predicates in detail, using both the function and object approaches.

The *odb*/Tools package supports two types of predicates, *selection* predicates which select individual objects based on the objects themselves and any relevant global or private state, and *traversal* iterators which determine what links should be followed during a depth first search. These predicate types and pre-defined classes for each are defined in the

following two sections. Another section provides information on defining new predicates to perform application-specific work.

8.1 Odb module `Predicates`

The `Predicates` module provides a number of fundamental selection predicates which may be used directly or in composition, or as examples from which application-specific predicates may be created. Predicates supporting composition are also defined in this module. These two groups of predicates are sufficient for many purposes, and are likely to be used in any application.

For all predicates, a single exception is defined for error conditions.

`PredicateError`

This exception is used for all internal failures in the `Predicates` and `Traversals` modules. The `Traversals` module exports this exception as well.

8.1.1 Selection Predicate Classes

The `Predicates` module defines the following classes as selection predicates:

`Predicate()`

This class is a do-nothing predicate which always returns true. This is used as a base class for all other predicates. There is no reason to ever instantiate this class.

`AcceptOnce()`

While the `FlatIterator` class visits each object in the database at most once, this is not the case with the `DepthIterator`, which may visit an object any number of times. This predicate may be used to ensure that each object is visited at most once in that case, or to limit the number of visits caused by a portion of a composite predicate. This is achieved by returning true the first time the predicate is called for an object and false for subsequent calls for the same object. Internal changes to the object do not affect the operation of this predicate.

`ObjectHasField(`*field_name*`)`

This predicate is used to determine if an object has a field by a particular name. If it does, the predicate returns true, otherwise it returns false.

`ObjectIsClass(`*class_id*`)`

Predicates created with the `ObjectIsClass` generator test the class of the database objects which they receive. If the ODB class ID of the object matches *class_id*, the predicate evaluates to true.

`SoundexValueMatch(`*field_name, target*`)`

This class creates predicates which perform approximate string matching using the soundex algorithm. To use this, the `soundex` modules must be available to the Python interpreter; this module is not standard but is freely available on the Internet. The arguments to the constructor are the name of the field to check and the string to be matched. The predicate checks each value entry for the named field and returns true if any value matches *target*. If the field contains no values or none match, the predicate returns false. The soundex algorithm is not case-sensitive.

`StringValueContains(`*field_name, target* [*, case*] `)`

This class of predicates tests for the presence of a particular string within a field of the object being checked. If *target* is found within any value for the field named by *field_name*, true is returned; the target can occur at any point in the field value to be qualify as a match. If the object has no field named *field_name* or has not values for that field, false is returned. The optional argument *case* may be set to true to enable case sensitivity.

`StringValueStartsWith(`*field_name, target* [*, case*] `)`

This predicate generator is very similar to the `StringValueContains`, with the distinction that it tests for a match only at the beginning of the field being tested.

`StringValueMatch(`*field_name, target* [*, case*] `)`

This is the simplest of the string-matching predicates. The field values and target are matched exactly, with case sensitivity being the only point of flexibility. As before,

this is set by the *case* parameter, and is off by default.

8.1.2 Compositional Predicate Classes

The basic selection predicates described above are useful as they stand, but much more powerful selection mechanisms can be created using *logical composition* to build new predicates from these simple predicates and application specific predicates. This is achieved using predicates which combine the affects of others. Classes representing fundamental logical operations are also provided. Each of these represents a single logical operation which can be performed on the results of one or more "child" predicates passed in as arguments when the predicate object is created.

Unlike other selection predicates, these may also be used as traversal predicates, described below. The predicate type for these classes is determined by the child predicates passed in as parameters to the constructor: the type of the first parameter predicate is adopted, and all others are checked to be of the same type. If predicates of both types are passed in, a `Predicates.PredicateError` exception is raised.

`LogicalPredicate()`

This class provides a common base for the logical composition predicates described below. Unlike the `Predicate` and `Traversal` classes, it is an uncaught error to instantiate this class directly. It is not a do-nothing class like the other predicate bases.

`And(`*pred ...*`)`

Performs a logical $\wedge$ (and) operation on a number of predicates. Short circuiting evaluation is implemented.

`Not(`*pred*`)`

Performs a logical $\neg$ (negation) operation on a single predicate.

`Or(`*pred ...*`)`

Performs a logical $\vee$ (or) operation on a number of predicates. Short circuiting evaluation is implemented.

8.2 Odb module `Traversals`

Traversal predicates are similar to selection predicates in that they select the targets of an action, but the information they receive and the scope of their effect is more limited. These predicates exist to control how depth-first iterators walk the object graph rather than to control the selection of objects. Specifically, they allow the traversal to be truncated at links in which the application is not interested. For instance, an application may be interested in objects linked on a field named "`cites`" but not a field named "`cited_by`." A traversal predicate can be used to "prune" the traversal by returning a false value if the source field is named "`cited_by`."

`Traversal()`

This class is a do-nothing predicate which always returns true. This is used as a base class for all other traversal predicates. There is no reason to ever instantiate this class.

`TraverseClass(`*class_id*`)`

Predicates created as `TraverseClass` objects return true for any traversals from objects with a class ID of *class_id*. No other criteria is used for distinction. In particular, all source fields are accepted.

`TraverseField(`*field_name*`)`

Predicates of this class are similar to those of `TraverseField`, but return true only for traversals from fields identified by name as *field_name*. No other criteria is used to make the distinction; in particular, the class of the source object is not considered.

`TraverseClassField(`*class_id, field_name*`)`

This predicate, a single predicate which performs each of the tests the previous to predicates provide, is equivalent to `And(TraverseClass(`*class_id*`), TraverseField(`*field_name*`))`. It is provided as a means to improve performance in situations which require this combination.

8.2.1 Compositional Predicates and Traversal

As with the selection predicates, each simple predicate is useful alone, but additional clarity and precision can be achieved by combining application-specific predicates and the simple predicates using logical composition. To encourage readability, separate logical predicates are not provided for traversal operations; the logical operations are available through the predicates `And`, `Or`, and `Not` described above. Each of these tests that all child predicates are of the same classification (selection or traversal), and assume that same classification for themselves. Both modules, `Predicates` and `Traversals`, export these common classes.

8.3 Implementing New Predicates

Applications may often want to implement new predicates, either for use in selection or in traversal. This may be for performance reasons, or the existing predicates may not provide enough strength or information regarding program state, or perhaps the predicate needs to record information about the cause of rejection. In any case, there exists the need to provide extensibility in the set of predicates.

There are two ways to implement any predicate: Implementing the predicate as a function allows simplicity for predicates which require no information beyond the object being examined or global state. Using objects to create predicates allows more flexible definitions of encapsulated predicates which can affect their own operation by maintaining private state information. This object-oriented approach is used in the `Predicates` and `Traversals` modules to allow clean definition of predicate classes which make use of internal state. Both methods will be described here.

8.3.1 Functions as Predicates

A function used as a predicate should require only one parameter and allow one to be given. Any number of parameters with default values may be included, to allow the function additional uses if appropriate. A function which takes no parameters or requires

more than one causes a `TypeError` exception to be raised when it is called. The only significant restriction on function predicates is that they cannot be composed using the logical composition predicates. This limitation is caused a requirement imposed by the base class of the compositional predicates: each predicate made a part of the composition must have a callable attribute named `_PredicateType` which returns a string indicating the general type of the predicate. This requirement is made to allow limited type checking and to determine how the composed predicates must be called. Functions cannot take on programmer-defined attributes, preventing them from being composable.

When a predicate is invoked on an object, it is called with that object as the only parameter. The return value should be true if the object should be included and false if not. Normal Python truth determination rules apply. The actual return value will not be preserved or passed on, so should be a lightweight object to prevent performance penalties. If the predicate is implemented as a function, the way to provide these semantics is clear. To create a predicate which checks the first element of the `'name'` field of an object for a particular name, we might write this function:

```
def myFunctionPredicate(object):
    return object.HasField('name') and object.name[0] == 'Drake'
```

This function would then be passed to the `Iterator` constructor to prepare to iterate over the database *db*:

```
iter = db.Iterator(myFunctionPredicate)

for object in iter:
    ...
```

8.3.2 Classes as Predicate Generators

The Python object model allows user-defined classes to support a `__call__()` discipline which implements function call semantics. This allows the creation of objects with internal state which may be created dynamically and interpreted as callable functions.

The first advantage of using objects as predicates is the ability to create multiple predicates with the same code, but each having its own private state information. The additional encapsulation provided by an object can greatly improve the maintainability of application code.

A second major advantage is the ability to provide and manipulate state independently of performing a call on the predicate; alternate methods may be provided which allow query and mutation of the state, or perhaps allow storage to and restoration from a serialized representation.

An advantage used heavily with the predicates provided in the `Predicates` and `Traversals` modules is that the predicate is the object instance, not the class definition. The class can be described as a *predicate generator*, providing a mechanism by which new predicates are created as needed. All the predicates generated by a class will share some common attributes provided by the class implementation, with the behavior of the predicate dependent also on the internal state of the individual object. The predicates provided all use constructor arguments to initialize this internal state, but this is not a requirement. Object predicates may provide additional methods that affect state, or they may modify it directly during the course of normal operation.

The simplest meaningful predicate is created from a class defining at least two disciplines: `__init__()` and `__call__()`. These are used for fairly simple reasons. `__init__()` is required to initialize the object and handle construction parameters, and `__call__()` provides the function call semantics, accepting the invocation parameter (parameters for traversal predicates, as we shall see), and computing and returning the result. Any other methods are optional as far as using the objects as predicates is concerned, but will not interfere with that operation either. If a predicate is to support logical composition it is important that it be derived from the `Predicate` class either directly or indirectly, and not provide the method `_PredicateType()`, unless a completely distinct form of predicate is being created. This special method is used by the compositional predicates to allow an improved measure of runtime integrity checking. This method must be inherited from either

the `Predicate` or `Traversal` base class.

The class definition below can be used as a skeleton for building useful selection predicate generator classes. The predicate instances will support the composition mechanism since the base class `Predicate` is used and the `_PredicateType()` method is not overridden.

```
class SelectionTemplate(Predicate):
    def __init__(self, *args):
        #  process argument list, create initial state
        self.state = function_of(args)

    def __call__(self, object):
        #  use 'object' and the internal state to determine the result
        if test_on(self.state) and query_on(object):
            #  possibly update self.state as well
            return 1

        return 0
```

Modifying this template with the appropriate construction and function call methods allows meaningful predicates to be created fairly easily. But also consider that the class does not need to provide all of its facilities directly: it can inherit from multiple base classes or indirectly from `Predicate`.

As an example of this approach to predicate design, the `StatPredicate` class, defined in the source fragment below, calls two member functions to do work during the normal invocation of the predicate. The `Measure()` method takes a statistical measurement on the node being examined, and the `Select()` method determines if the node should be selected. Since this is used as a base class, these methods need not provide any high level of functionality in this class, but should be provided as minimal implementations. The `__call__()` method provides the required access to each of these methods independently.

```
from Predicates import Predicate

class StatPredicate(Predicate):
    def __call__(self, obj):
        self.Measure(obj)
        return self.Select(obj)

    def Measure(self, obj):
        pass

    def Select(self, obj):
        return 0
```

For the `StatPredicate` class to be useful, a subclass needs to be created which provides at least a more significant `Measure()` method, and, optionally, a different `Select()` implementation. These two methods could be provided by different base classes of the final predicate class, if appropriate.

One simple but potentially useful statistic which might be desired as a side effect of iterating over a database is a count of the number of nodes selected grouped by the entity class ID of the selected nodes. This could easily be achieved by using a predicate class which accepts all nodes checked and maintains a count of the nodes of each class ID internally. During or after the iteration, the state of the predicate could be tested to gain the needed information. Since the `StatPredicate` class rejects all nodes when defined as above, a new `Select()` method must be provided. A class may be defined which provides this capability. Since this may be useful independently of the measurement being taken, this class may be independent of the class providing the statistical measurement.

```
class AlwaysAccept:
    def Select(self, obj):
        return 1
```

Since the `Measure()` method required to implement the required capability is unlikely to be used in other predicates, the implementation in the `StatPredicate` base class can be overridden by a method in the final predicate class. The `__call__()` method may be

inherited from `StatPredicate` and `Select()` may be inherited from `AlwaysAccept`. The source for the `CountPredicate` class is given below. Note that the list of base classes includes `AlwaysAccept` before `StatPredicate`; this specifies that the implementation of the `Select()` method should be taken from `AlwaysAccept`.

Perhaps the most important aspect of this example is that the finished class used to create the statistical information acquires each aspect of its functionality from a different source in the class hierarchy, and the demonstration that multiple inheritance is immediately useful in the context of predicate creation. This example is contained in the *odb*/Tools package as the module `CountPred`.

```
class CountPredicate(AlwaysAccept, StatPredicate):
    def __init__(self):
        self._CNT = {}

    def Measure(self, obj):
        class_id = obj.ClassID()

        try:
            self._CNT[class_id] = self._CNT[class_id] + 1
        except KeyError:
            self._CNT[class_id] = 1

    def Count(self, class_id = None):
        '''Provide access to either the entire dictionary of results or
        to the counts for individual classes.'''

        if class_id is None:
            return self._CNT
        else:
            try:
                return self._CNT[class_id]
            except KeyError:
                return 0
```

8.3.3 Traversal Predicates

Traversal predicates operate in much the same way as selection predicates but are called at a different point during iteration and receive different parameters. Both the function and object implementation methods are available, with the same restriction on function predicates: they are not composable.

Each call to a general traversal predicate provides two parameters. Each parameter is an an *endpoint* which provides describing the class and field definitions for one end of the link. Endpoints have two data attributes, `classdefn` and `fielddefn`, corresponding to these two values. Where possible, endpoints support a lazy database lookup, allowing the target not to be retrieved if the predicate can be resolved to a false value without examining the terminating endpoint.

These two data regarding the endpoints are only information traversal predicates are allowed regarding the actual objects involved in the traversal. Since these predicates exist to control the selection of link semantics which are used rather than node selection, this is the only information pertinent to the decision. The class definitions are provided as `ClassDefn` objects, and field definitions take the form of `FieldDefn` instances.

8.3.4 Invoking Predicates Directly

While the use of predicates from within predicates should never be necessary, it can be useful to simplify the composition, and may provide an efficiency boost where a predicate may only be required in unusual circumstances, or where it may be replaced dynamically during an iteration. Other uses for predicates may arise as well, so it is useful to be able to invoke a predicate directly, without depending on the iteration mechanisms.

Since predicates are implemented either as functions or as objects, a variable bound to a predicate can be called just like a function. This example shows how to create a selection predicate and invoke it independently of the iteration mechanism.

```
from Predicates import *

pred = And(StringValueMatch('first_name', 'Fred'),
           StringValueMatch('last_name',  'Drake'))

if pred(myObject):
    print 'Object matches!'
```

While traversal predicates can be used this way as well, selection predicates will prove useful in more situations. Remember to pass the correct parameters to a traversal predicate. Classes for endpoints may be found in the module `Traversals`. If both the class and field definitions are known, the class `Endpoint` should be used. If a database lookup is required to determine the class and field definitions, use the class `DelayedEndpoint` to support lazy evaluation of the public data members.

Appendix A

Installing *odb*/Tools

This section describes the installation process for obtaining and installing the complete *odb*/Tools package. Information is included regarding where the base software distributions for Python and GDBM may be found, and what is required to install them with *odb*/Tools. For each package, be aware that changes may have been made in the distributions of the base software; please read all release notes included with these packages to allow adjustment to the installation process described here if needed.

A.1 Obtaining Python

The Python distribution is available free of charge to any individual or organization with access to the Internet, and is likely available on several CD-ROM distributions of the Linux operating system. It is often available on recent CD-ROMs providing interpreted language tools. There are no restrictions on using Python for any purpose, including commercial uses, so long as the copyright is maintained intact.

The primary means of acquiring Python is by FTP from the home site maintained by the Python Software Activity, or from one of the mirror sites listed below.

Address	Directory	Region
`ftp.python.org`	`/pub/python`	Eastern U.S.
`ftp.uu.net`	`/languages/python`	Eastern U.S.
`gatekeeper.dec.com`	`/pub/plan/python`	Western U.S.
`ftp.wustl.edu`	`/graphics/graphics/sgi-stuff/python`	North-west U.S.
`ftp.cwi.nl`	`/pub/python`	Europe

At each site, below the directory listed above, the directory `src` contains compressed archives containing the source code to the interpreter, the library, and LaTeX source for the

documentation. Formatted copies of the documentation are in the `doc` directory. Refer to the files named '`INDEX`' in each directory for information on the current version and which files are needed.

A.2 Obtaining and Installing GDBM

The GNU GDBM library is required for *odb*/Tools as well as for the DELTO system, so sites which have installed the Envision software will typically already have GDBM installed. Check with the system administrator to find out if this package is already available and how it has been installed.

If the library has not been installed previously, locate a copy of the distribution archive at one of the GNU distributions sites. Though a long list of these sites is well publicized, a short list is included below for convenience. The archive to retrieve will be named '`gdbm-?.?.?.tar.gz`', where `?.?.?` is the version number. Un-archive the distribution and build and install it according to the instructions in the file '`README`'; there are no special considerations in building this package for Envision software, including *odb*/Tools. If installing this for general availability on-site, the system administrator may need to lend assistance with the final stage of installation.

The GDBM library is built with debugging information by default. If this is not desired, optimization can be enabled from the `make` command line. If using the GCC compiler, the command line below is recommended.

```
make CFLAGS=-O2 LDFLAGS=-O2
```

Regardless of the manner of installation, the option used to link the library to an executable will be needed during the Python configuration. Make a note of the required option. Typically this will be `-lgdbm` if the library is installed for general availability or the entire path name of the library otherwise.

Address	Directory	Region
prep.ai.mit.edu	/pub/gnu	Eastern U.S.
sunsite.unc.edu	/pub/gnu	Eastern U.S.
gatekeeper.dec.com	/pub/gnu	Western U.S.

A.3 Building and Installing *odb*/Tools

The *odb*/Tools package is fairly easy to install, with most of the installation being handled by the standard Python installation mechanisms.

To begin the installation, the compressed archives containing the Python and *odb*/Tools distributions should be in the directory where the project directory should be created. Unpack the distribution using these commands:

```
gzip -dc python?.?.tar.gz | tar xf -
cd Python-?.?
gzip -dc ../odbTools.tar.gz | tar xf -
```

This will create a directory with a name of the form `Python-?.?`, where `?.?` is the version number of Python. The directory will contain a clean Python source tree and a number of files needed to add the *odb*/Tools functionality. To update the build system and enable modules required for the installation of *odb*/Tools, run the supplied shell script to patch various files and detect some site-specific information:

```
sh odbPatch.sh
```

The shell script run by this command will make a number of small patches to several of the files used in the build procedure delivered as part of the Python distribution. These changes add information to files which are used to build the input files for `make`. The modifications are handled this way to allow the Python distribution to be reconfigured without removing information required to build *odb*/Tools.

The next step in the installation is to configure the Python distribution. This typically is only required once, but there are several options which should be considered. Information about these options are detailed in the file '`README`' in the `Python-?.?` directory. The only

options discussed here are those likely to be important for a project making use of Python for access to *odb*/Tools.

A.3.1 Compiler Selection

As with much of the software available over the Internet, the Python configuration will use the GCC compiler from the Free Software Foundation by default. Though most of the development and testing of Python and *odb*/Tools used this compiler, if it was not used to compile other parts of the Envision software, including the GDBM library, the standard `cc` may be a better choice. This will be used automatically if GCC is not available, but if both compilers are available, use the `--without-gcc` option when running `./configure`, as described below.

A.3.2 Using the Readline Library

If *odb*/Tools will be used in a rapid-prototyping environment, adding the GNU `readline` library to the Python interpreter can be advantageous. The `readline` library allows the interpreter's interactive prompt to support a statement history of commands executed from the prompt. If the interpreter is used interactively for prototyping or debugging, this can be a great time saver. To enable this capability, use the `--with-readline=`*DIR* option, where *DIR* is the directory containing the compiled readline library. This library may need to be obtained if not already installed. It is available at the same sites as the GDBM library.

A.3.3 Setting the Installation Prefix

By default, Python will be configured for installation under '`/usr/local`', with each file in the appropriate subtree below that. If the interpreter being built will only be used with Envision, a different prefix may be required. For instance, if the Envision software tree is normally rooted at '`/projects/Envision`', this should be used as the prefix for configuring the Python interpreter as well. Use configuration option `--prefix=`*DIR* will set *DIR* to be the installation prefix. This is only required if the prefix should not be '`/usr/local`'.

A.3.4 Running `./configure`

The first stage of configuring the Python interpreter involves running the script `./configure` located in the `Python-?.?` directory. Any options selected based on the discussion above should be added to the command line. Since this script is a fairly typical configuration script as generated by the Free Software Foundation's `autoconf` program, there may be some additional options appropriate to particular sites. The system administrator should be able to provide assistance with this if needed. Run this script from the command line with any options selected.

A.3.5 Configuring the Built-in Modules

Many modules are available with the Python distribution. While most are written in Python and do not need to be configured, there are a number of modules which are written in C and must be compiled and linked with the interpreter. Some platforms allow these modules to be loaded dynamically if dynamic loading and linking is supported; refer to the instructions included with the Python distribution for information on building the interpreter to support this.

Many platforms do not support dynamic loading, and modules must be linked with the interpreter statically. The directory `Python-?.?/Modules` contains the file used to configure modules to be linked with the interpreter. If the file '`Setup`' does not exist in this directory, copy the file '`Setup.in`' to '`Setup`'. Only '`Setup`' should ever be edited. It may have been created by the '`odbPatch.sh`' script.

Examine the file '`Setup`' to determine which modules are available which have to be linked with the interpreter. Instructions for changing the file itself are located at the top of the file. If a desired module is commented out, simply un-comment the definition in this file. Be aware that some modules are specific to particular platforms; this will be indicated in the comments in the file. By default, the `gdbm` module is not built with a Python interpreter, but is required by *odb*/Tools. The '`odbPatch.sh`' script will have

made the necessary change to cause this to be built. It is important not to comment out this definition: this module is required in order for *odb*/Tools to operate. The *odb*/Tools package has also provided and enabled the `soundex` module to support approximate string matching in the `Predicates` module. If this isn't desired, this module may be commented out in '`Setup`'. The `Predicates` module will accommodate the lack of availability.

If using the `tkinter` module, refer to the comments in the '`README`' file in the `Python-?.?` directory. The `tkinter` module provides access to the Tk toolkit commonly associated with Tcl/Tk.

A.3.6 Building the Interpreter

Use the `make` command to build the interpreter. If desired, use the `OPT` variable to define the level of optimization used during compilation; the default is to include debugging information using the '`-g`' option. When using GCC to compile, the following command is recommended:

```
make OPT=-O2
```

Several levels of installation are available with the Python interpreter. The file '`README`' includes documentation on installation options and procedures; using *odb*/Tools requires at least `make install` and `make libinstall`. The executable program containing the augmented Python interpreter is named `odbtools`.

A.4 Testing the Installation

Once the interpreter has been built, there are two `make` targets which may be used to test the installation. The target `test` determines the success of the underlying Python build. The target `testodb` may be used to test the *odb*/Tools build; this target tests only the ability to import the various built-in and external modules which make up *odb*/Tools. These two `make` targets are independent of each other and may be used before issuing the '`make install`' and '`make libinstall`' commands as discussed above.

A.5 Version Information

This section provides information regarding the versions of software with which *odb*/Tools was developed and tested. Earlier versions of the Python interpreter may cause some difficulties with installation; versions before 1.1 may fail to build with *odb*/Tools.

Package	Linux 1.2	OSF/1 3.2	SunOS 4.1
GNU GCC compiler	2.6.2	2.6.3	2.5.8
GNU GDBM database library	1.7.3	1.7.1	1.7.1
GNU readline library	2.0.1		
Python	1.2	1.2	1.2

Appendix B

Running *odb*/Tools Scripts

This appendix provides instructions on making *odb*/Tools scripts executable from the command line of a UNIX-style command line environment. The installation procedures described in the previous appendix are assumed; some adjustments may be needed if significant variations occurred.

odb/Tools scripts are executed in the same way that most shell scripts are handled from the shell command line. The script must be saved in a text file with the execute permission set. The first line of the file must contain the '`#!`' characters at the start of the line, followed by the path to the interpreter. Using the installation instructions included with this document, this would be the installation prefix followed by '`/bin/odbtools`'. The remainder of the script should simply be the main module of the program, which may do no more than importing another module and calling a driver function. The beginning of an *odb*/Tools script might look like this using the installation defaults:

```
#!/usr/local/bin/odbtools
#
#  This is a script based on odb/Tools.  It is an example of ...
```

To set the executable permission for an *odb*/Tools script, use the `chmod`(1) command. Variations may be appropriate based on site administration policies.

```
chmod +x odb.script
```

Running a script created this way involves only typing the name of the script on the command line, optionally followed by any parameters used by the script. Command line parameters may be found in '`sys.argv[1:]`', and the name of the script will be in '`sys.argv[0]`' [vR95b].

REFERENCES

[Ave95] Guillermo A. Averboch. A system for document analysis, translation, and automatic hypertext linking. Master's thesis, Virginia Polytechnic Institute and State University, June 1995.

[Con95] Matthew J. Conway. Python: A GUI development tool. *interactions*, pages 23–28, April 1995.

[Gau94] Pierre Gaumond. *GNU GDBM Manual*. Free Software Foundation, 1.7.3 edition, 1994. Online documentation.

[HHN+95] Lenwood S. Heath, Deborah Hix, Lucy T. Nowell, William C. Wake, Guillermo A. Averboch, Eric Labow, Scott A. Guyer, Dennis J. Bruenu, Robert K. France, Kaushal Dalal, and Edward A. Fox. Envision: A user-centered database of computer science literature. *Communications of the ACM*, 38(4):52–53, April 1995.

[Lut96] Mark Lutz. *Using Python*. O'Reilly & Associates, 1996. Expected first quarter.

[Ted93] Lucy A. Tedd. *An Introduction to Computer-Based Library Systems*. John Wiley & Sons, 3 edition, 1993.

[Uni91] The Unicode Consortium. *The Unicode standard : worldwide character encoding*, volume 1. Addison-Wesley Publishing, 1 edition, 1991.

[vR95a] Guido van Rossum. *Extending and Embedding the Python Interpreter*. Stichting Mathematisch Centrum, April 1995. Release 1.2.

[vR95b] Guido van Rossum. *Python Library Manual*. Stichting Mathematisch Centrum, April 1995. Release 1.2.

[vR95c] Guido van Rossum. *Python Reference Manual*. Stichting Mathematisch Centrum, April 1995. Release 1.2.

[vR95d] Guido van Rossum. *Python Tutorial*. Stichting Mathematisch Centrum, April 1995. Release 1.2.

[vRdB91] Guido van Rossum and Jelke de Boer. Interactively testing remote servers using the python programming language. *CWI Quarterly*, 4(4):283–303, December 1991.

[Wal94] Norman Walsh. *Making TEX Work*. O'Reilly & Associates, 1994.

Index

AcceptOnce (in module Predicates), 72, 76
AddClass (DataDictionary method), 45
AddField (ClassDefn method), 47
AddIndexFile (Database method), 53
AddObjectFile (Database method), 54
add_class (OdbDatabase method), 30
add_field (OdbDatabase method), 31
add_index_file (OdbDatabase method), 29
add_object_file (OdbDatabase method), 30
alternative ID, 21, 32, 47
AlternativeID (ClassDefn method), 47
And (in module Predicates), 78
append (FieldProtocol method), 63
Averboch, Guillermo, 3

character sets, 41
  multibyte, 42
  Unicode, 41
Class (DataDictionary method), 45
ClassDefn (DataDictionary method), 46
ClassDefn module, 15, 46
Classes (DataDictionary method), 45
classes (OdbDatabase method), 31
ClassID (ObjectProtocol method), 60
class_id (OdbInstance attribute), 38
Close (Database method), 54
close (OdbDatabase method), 30
closed (Database attribute), 56
closed (OdbDatabase attribute), 36
Commit (Database method), 54
control files, 23
CountPred module, 16, 85
current_index_file (Database attribute), 57
current_index_file (OdbDatabase attribute), 36
current_object_file (Database attribute), 57
current_object_file (OdbDatabase attribute), 36

Database (in module Database), 52
Database module, 14, 52
database (OdbInstance attribute), 39
DatabaseError (in module Database), 53
DataDictionary (Database method), 54
DataDictionary (in module DataDictionary), 44
DataDictionary module, 15, 44
DDError (in module DataDictionary), 44
DDictReport module, 16
delete_field (OdbDatabase method), 31
delete_object (OdbDatabase method), 33
Delta module, 14
DELTO, 1, 20, 24, 89
DepthIterator (in module Iterators), 72

endpoints, 86, 87
Envision, 1, 89
error (in module odb), 27
error_msg (in module odb), 25

Field (ClassDefn method), 47
Field (ObjectProtocol method), 60
field type specifiers, 41
FieldDefn (ClassDefn method), 49
FieldDefn module, 15, 49
FieldDelta (in module FieldProtocol), 66
FieldID (FieldProtocol method), 63
FieldIndirect (in module Database), 66

FieldInstance (in module Object), 66
FieldProtocol (in module FieldProtocol), 65
Fields (ClassDefn method), 48
Fields (ObjectProtocol method), 61
field_data (OdbInstance method), 38
field_types (OdbDatabase method), 32
field_types (in module odb), 25
filemode (Database attribute), 57
filemode (OdbDatabase attribute), 36
FlatIterator (in module Iterators), 70
Flexibility (common data dictionary method), 43
flexibility, 43, 46, 49, 51
Flexible (common data dictionary method), 43
Free Software Foundation, 23, 91

GCC, 91, 93
get_class_info (OdbDatabase method), 32
get_index_entry (OdbDatabase method), 35

HasClass (DataDictionary method), 45
HasField (ClassDefn method), 48
HasField (ObjectProtocol method), 61
has_field (OdbInstance method), 38
has_key (Database method), 55
has_key (FieldProtocol method), 63
has_key (OdbDatabase method), 34

ID (ClassDefn method), 48
ID (FieldDefn method), 49
index file group, 23, 29, 36, 37, 53, 57, 58
index_extra_blocking (in module odb), 25
Internet, 13, 77, 88
items (FieldProtocol method), 64
Iterators module, 16, 69

keys (Database method), 56
keys (FieldProtocol method), 64
keys (OdbDatabase method), 34

LinkTo (FieldProtocol method), 64
Linux, 88
LogicalPredicate (in module Predicates), 78
Lutz, Mark, 14

Max (FieldDefn method), 49
MAX_CLASS_ID_LENGTH (in module odb), 28
MAX_CLASS_NAME_LENGTH (in module odb), 28
MAX_CLASS_OTHER_INFO_LENGTH (in module odb), 28
MAX_DATABASE_NAME_LENGTH (in module odb), 28
MAX_ERROR_MSG_LENGTH (in module odb), 28
MAX_FIELD_NAME_LENGTH (in module odb), 28
MAX_INDEX_STR_LENGTH (in module odb), 28
MAX_NUMBER_INDEX_FILES (in module odb), 28
MAX_NUMBER_OBJECT_FILES (in module odb), 29
MAX_OBJ_ID_LENGTH (in module odb), 29
Min (FieldDefn method), 50
modules
  ClassDefn, 15, 46
  CountPred, 16, 85
  DDictReport, 16
  DataDictionary, 15, 44
  Database, 14, 52
  Delta, 14
  FieldDefn, 15, 49
  Iterators, 16, 69
  Predicates, 16, 76
  Traversals, 16, 79
  odb, 14
  odb, 24

`Mutable` (Database method), 55
`Mutable` (FieldProtocol method), 64
`Mutable` (ObjectProtocol method), 61
`Mutable` (common data dictionary method), 43

`Name` (ClassDefn method), 48
`Name` (FieldDefn method), 50
`name` (Database attribute), 57
`name` (OdbDatabase attribute), 36
`NewObject` (Database method), 56
`new_object` (OdbDatabase method), 34
`Not` (in module Predicates), 78
`number_index_files` (Database attribute), 57
`number_index_files` (OdbDatabase attribute), 37
`number_object_files` (Database attribute), 58
`number_object_files` (OdbDatabase attribute), 37

object file group, 23, 30, 36, 37, 54, 57, 58
`ObjectDelta` (in module Delta), 62
`ObjectHasField` (in module Predicates), 77
`ObjectID` (FieldProtocol method), 64
`ObjectID` (ObjectProtocol method), 61
`ObjectIndirect` (in module Database), 62
`ObjectInstance` (in module Object), 63
`ObjectIsClass` (in module Predicates), 77
`ObjectProtocol` (in module ObjectProtocol), 62
`object_id` (OdbInstance attribute), 39
`odb` (built-in module), 24
`odb` module, 14
`OdbDatabaseType` (in module odb), 27
`OdbInstanceType` (in module odb), 27
`OdbKeysIteratorType` (in module odb), 27
`open` (Database attribute), 58
`open` (OdbDatabase attribute), 37
`open` (in module odb), 26
`Or` (in module Predicates), 78
O'Reilly & Associates, 14
`OtherInfo` (ClassDefn method), 48

Perl, 11
`Predicate` (in module Predicates), 76
`PredicateError` (in module Predicates), 76
`Predicates` module, 16, 76
Python Software Activity, 13, 88

`Range` (FieldDefn method), 50
`RemoveLinkTo` (FieldProtocol method), 64
`RemoveObject` (Database method), 56
`Rollback` (Database method), 55
`rwmode` (Database attribute), 58
`rwmode` (OdbDatabase attribute), 37

`SoundexValueMatch` (in module Predicates), 77
`store_data` (OdbInstance method), 38
`store_index_entry` (OdbDatabase method), 35
`StringValueContains` (in module Predicates), 77
`StringValueMatch` (in module Predicates), 77
`StringValueStartsWith` (in module Predicates), 77

Tcl/Tk, 11, 93
transaction processing
  committing, 54
  rolling back, 55
`Traversal` (in module Traversals), 79
`Traversals` module, 16, 79
`TraverseClass` (in module Traversals), 79

`TraverseClassField` (in module Traversals), 79
`TraverseField` (in module Traversals), 79
`Type` (FieldDefn method), 51
type specifiers, 41

Unicode, 41
`update_class` (OdbDatabase method), 32
`update_field` (OdbDatabase method), 32

`values` (FieldProtocol method), 65