Building a Layered Framework for the Table Abstraction

H. Conrad Cunningham
Dept. of Computer & Information Science
University of Mississippi

Jingyi Wang
Acxiom Corporation
What is the Table Abstract Data Type?

- Collection of records
- One or more data fields per record
- Unique key value for each record
- Key-based access to record
- Many possible implementations

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key1</td>
<td>Data1</td>
</tr>
<tr>
<td>Key2</td>
<td>Data2</td>
</tr>
<tr>
<td>Key3</td>
<td>Data3</td>
</tr>
<tr>
<td>Key4</td>
<td>Data4</td>
</tr>
</tbody>
</table>
Table Operations

• Insert new record
• Delete existing record given key
• Update existing record
• Retrieve existing record given key
• Get number of records
• Query whether contains given key
• Query whether empty
• Query whether full
What is a Framework?

- Reusable object-oriented design
- Collection of abstract classes (and interfaces)
- Interactions among instances
- Skeleton that can be customized
- Inversion of control (upside-down library)
Requirements for Table Framework

- Provide Table operations
- Support many implementations
- Separate key-based access mechanism from storage mechanism
- Present coherent abstractions with well-defined interfaces
- Use design contracts and design patterns
Design Contracts

- Preconditions for correct use of operation
- Postconditions for correct result of operation
- Invariant conditions for correct implementation of class

Insert record operation
  pre: record is valid and not already in table
  post: record now in table

Invariant for table
  all records are valid, no duplicate keys
Design Patterns

• Describe recurring design problems arising in specific contexts
• Present well-proven generic solution schemes
• Describe solution’s components and their responsibilities and relationships
• To use:
  – select pattern that fits problem
  – structure solution to follow pattern
Layered Architecture Pattern

- Distinct groups of services
- Hierarchical arrangement of groups into layers
- Layer above implemented with services of layer below
- Enables independent implementation of layers
Applying the Layered Architecture Pattern

Client Layer
- client programs
- use layer below to store and retrieve records

Access Layer
- table implementations
- provide key-based access to records for layer above
- use physical storage in layer below

Storage Layer
- storage managers
- provide physical storage for records
Access Layer Design

Challenges:
- support client-defined keys and records
- enable diverse implementations of the table

Comparable interface for keys (in Java library)
- int compareTo(Object key) compares object with argument

Keyed interface for records
- Comparable getKey() extracts key from record

Table
- table operations
Access Layer Model

Partial function `table :: Comparable → Keyed`
- represents abstract table state
- #table in postcondition denotes table before operation

Abstract predicates (depend upon environment)
- `isValidKey(Comparable)` to identify valid keys
- `isValidRec(Keyed)` to identify valid records
- `isStorable(Keyed)` to identify records that can be stored

Invariant:

\[(\forall k, r : r = \text{table}(k) : \text{isValidKey}(k) \&\& \text{isValidRec}(r) \&\& \text{isStorable}(r) \&\& k = r.\text{getKey()} )\]
Table Design Contract (1 of 4)

void insert(Keyed r) inserts r into table
   Pre: isValidRec(r) && isStorable(r) &&
       !containsKey(r.getKey()) && !isFull()
   Post: table = #table ∪ {(r.getKey(),r)}

void delete(Comparable key) removes record with key from table
   Pre: isValidKey(key) && containsKey(key)
   Post: table = #table - {(key,#table(key))}
Table Design Contract (2 of 4)

void update(Keyed r) changes record with same key

Pre: isValidRec(r) && isStorable(r) && containsKey(r.getKey())

Post: table = (#table -
{(r.getKey()),#table(r.getKey())}) ) ∪
{(r.getKey()),r})

Keyed retrieve(Comparable key) returns record with key

Pre: isValidKey(key) && containsKey(key)

Post: result = #table(r.getKey())
int getSize() returns size of table
   Pre: true
   Post: result = cardinality(#table)

boolean containsKey(Comparable key) searches table for key
   Pre: isValidKey(key)
   Post: result = defined(#table(key))
Table Design Contract (4 of 4)

boolean isEmpty() checks whether table is empty
   Pre: true
   Post: result = (#table = ∅)

boolean isFull() checks whether table is full
   – for unbounded, always returns false
   Pre: true
   Post: result = (#table has no free space to store record)
Client/Access Layer Interactions

- Client calls Access Layer class implementing *Table* interface

- Access calls back to Client implementations of *Keyed* and *Comparable* interfaces
Storage Layer Design

Challenges:
- support client-defined records
- support diverse table implementations in Access Layer (simple indexes, hashing, balanced trees, etc.)
- allow diverse physical media (in-memory, on-disk storage, etc.)
- enable persistence of table
- decouple implementations as much as possible

Patterns:
- Bridge
- Proxy
Bridge Pattern

• Decouple “interface” of abstraction from “implementation”
  – table from storage in this case
• Allow them to vary independently
  – plug any storage mechanism into table
Proxy Pattern

- Transparently manage services of target object
  - isolate Table implementation from exact nature and location of record slots in RecordStore implementation
- Introduce proxy object as surrogate for target object
Storage Layer Interfaces

RecordStore
   – operations to allocate and deallocate storage slots

RecordSlot
   – operations to get and set records in slots
   – operations to get handle and containing RecordStore

Record
   – operations to read and write client records
Storage Layer Model

Partial function \texttt{store} :: int \to Object
- represents abstract \texttt{RecordStore} state

\texttt{Set Handles} \subseteq \texttt{int}, \texttt{NULLHANDLE} \notin \texttt{Handles}
\texttt{Set alloc} \subseteq \texttt{Handles}
- represents set of allocated slot handles
\texttt{Set unalloc} = \texttt{Handles} - \texttt{alloc}
- represents set of unallocated slot handles

Invariant:
\[(\forall h, r : r = \texttt{store}(r) : \texttt{isStorable}(r)) \land \land \]
\[(\forall h : h \in \texttt{alloc} \equiv \texttt{defined}(\texttt{store}(h)))\]
RecordStore Design Contract (1 of 2)

RecordSlot getSlot() allocates a new record slot

Pre:  true
Post:  result.getContainer() = this_RecordStore
       && result.getRecord() = NULLRECORD
       && result.getHandle() \notin #alloc
       && result.getHandle() \in alloc \cup \{NULLHANDLE\}

RecordSlot getSlot(int handle) rebuilds record slot using given handle

Pre:  handle \in alloc
Post:  result.getContainer() = this_RecordStore
       && result.getRecord() = #store(handle)
       && result.getHandle() = handle
void releaseSlot(RecordSlot slot) deallocates record slot

Pre: slot.getHandle() ∈ alloc
Post: alloc = #alloc - {slot.getHandle()} &&
     store = #store -
     {(slot.getHandle(), slot.getRecord())}
RecordSlot Design Contract (1 of 3)

void setRecord(Object rec) stores rec in this slot
   -- allocation of handle done here or already done by getSlot()

Pre:  isStorable(rec)
Post:
   Let h = getHandle() && g ∈ #unalloc:
   (h ∈ #alloc ⇒ store = (#store -
    {(h,#store(h))}) ∪ {(h,rec)}) &&
   (h = NULLHANDLE ⇒ alloc = #alloc ∪ {g} &&
    store = #store ∪ {(g,rec)})
RecordSlot Design Contract (2 of 3)

Object `getRecord()` returns record stored in this slot

Pre: true
Post: Let \( h = \text{getHandle}() \):
\[
(h \in \text{alloc} \Rightarrow \text{result} = \text{store}(h)) \land \land
(h = \text{NULLHANDLE} \Rightarrow \text{result} = \text{NULLRECORD})
\]

`int getHandle()` returns handle of this slot

Pre: true
Post: \text{result} = \text{handle associated with this slot}
RecordSlot Design Contract (3 of 3)

RecordStore getContainer() returns reference to RecordStore holding this slot
  Pre: true
  Post: result = RecordStore associated with this slot

boolean isEmpty() determines whether this slot empty
  Pre: true
  Post: result = (getHandle() = NULLHANDLE || record associated with slot is NULLRECORD)
Record Interface

Problem: how to write client’s record in generic way

Solution: call back to client’s record implementation

void writeRecord(DataOutput) writes the client’s record to stream
void readRecord(DataInput) reads the client’s record from stream
int getLength() returns number of bytes written by writeRecord
Abstraction Usage Relationships

Table \rightarrow Keyed

Comparable

RecordStore \rightarrow RecordSlot

Record

Access Layer

Storage Layer
Other Design Patterns Used

• Interface
• Null Object
• Iterator
  – extended Table operations
  – query mechanism
  – utility classes
• Template Method
• Decorator
• Strategy
Evolving Frameworks Patterns

- Generalizing from three examples

- Whitebox and blackbox frameworks

- Component library
  - Wang prototype two Tables and three RecordStores

- Hot spots
Conclusions

• Novel design achieved by separating access and storage mechanisms

• Design patterns offered systematic way to discover reliable designs

• Design contracts helped make specifications precise

• Case study potentially useful for educational purposes
Future Work

- Modify prototypes to match revised design
- Adapt earlier work of students on AVL and B-Tree class libraries
- Integrate into SoftwareInterfaces library
- Study hot spots and build finer-grained component library
- Study use of Schmid’s systematic generalization methodology for this problem
- Develop instructional materials
Acknowledgements

- Jingyi Wang for her work on the prototype framework
- Wei Feng, Jian Hu, and Deep Sharma for their work on earlier table-related libraries
- Bob Cook and Jennifer Jie Xu for reading the paper and making useful suggestions
- Sudharshan Vazhkudai, Jennifer Jie Xu, Vandana Thomas, Cuihua Zhang, Xiaobin Pang, and Ming Wei for work on other frameworks
- Todd Stevens, the Ole Miss patterns discussion group, and students in my Software Architecture and Distributed Objects classes for their suggestions
- Acxiom Corporation for its encouragement
- Diana Cunningham for her patience