Building a Layered Framework for the Table Abstraction

H. Conrad Cunningham

Dept. of Computer & Information Science

University of Mississippi

Jingyi Wang

Acxiom Corporation

Project

Context: development of an instructional data and file structures library

- artifacts for study of good design techniques
- system for use, extension, and modification

Motivation: study techniques for

- presenting important methods to students (frameworks, software design patterns, design by contract, etc.)
- unifying related file and data structures in framework

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

Key1	Data1
Key2	Data2
Key3	Data3
Key4	Data4

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

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 software design patterns and design contracts

Software Design Contracts

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

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

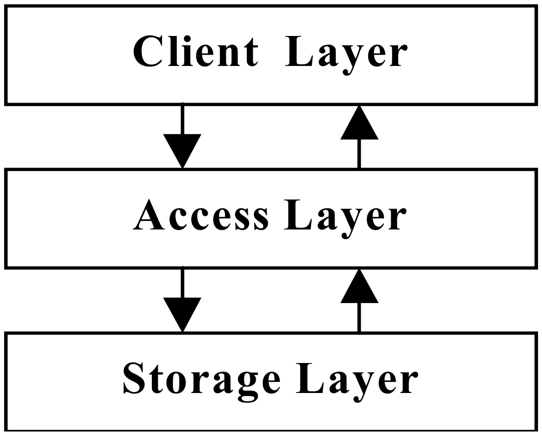
Invariant for table
all records are valid, no duplicate keys

Software 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 implemented with services of layer below
- Enables independent implementation of layers



Applying Layered Architecture Pattern

Client Layer

- client programs
- uses layer below to store and retrieve records

Access Layer

- table implementations
- provides key-based access to records for layer above
- uses physical storage in layer below

Storage Layer

- storage managers
- provides physical storage for records

Access Layer Design

Challenges:

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

Pattern:

- Interface

Access Layer Interfaces

`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

Table Interface

`void insert(Keyed r)` inserts `r` into table

`void delete(Comparable key)` removes record with `key`

`void update(Keyed r)` changes record with same key

`Keyed retrieve(Comparable key)` returns record with `key`

`int getSize()` returns size of table

`boolean containsKey(Comparable key)` searches for `key`

`boolean isEmpty()` checks whether table is empty

`boolean isFull()` checks whether table is full

- for unbounded, always returns false

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:

```
(∀ k, r : r = table(k) :
   isValidKey(k) && isValidRec(r)  &&
   isStorable(r) && k = r.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())`

Table Design Contract (3 of 4)

`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)

Access Layer Challenges

Support client-defined keys and records

– callbacks to `Comparable` and `Keyed` abstractions which hide the implementation details

Enable diverse implementations of the table

– careful design of table interface semantics using design by contract

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:

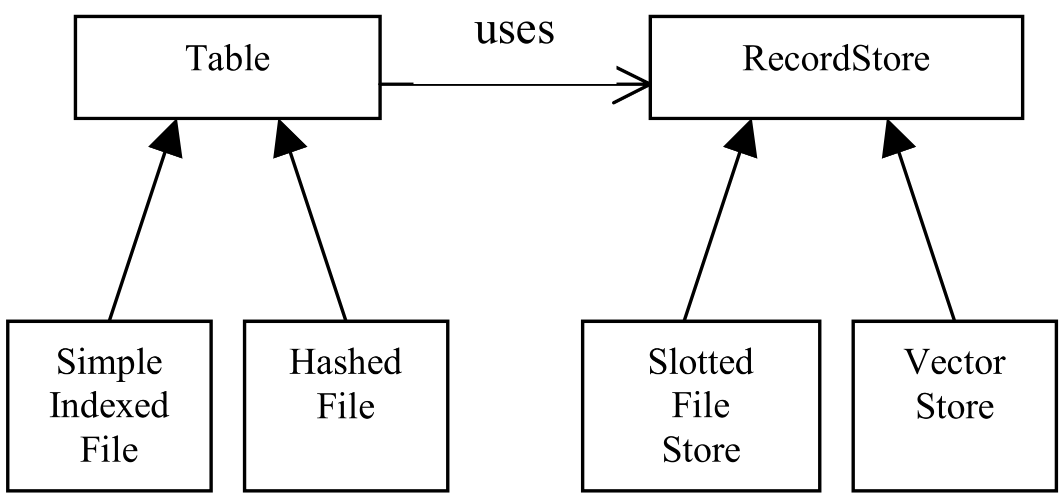
- support diverse table implementations in Access Layer (simple indexes, hashing, balanced trees, etc.)
- allow diverse physical media (in-memory, on-disk, etc.)
- decouple implementations as much as possible
- support client-defined records
- enable persistence of table, including access layer

Patterns:

- Bridge
- Proxy

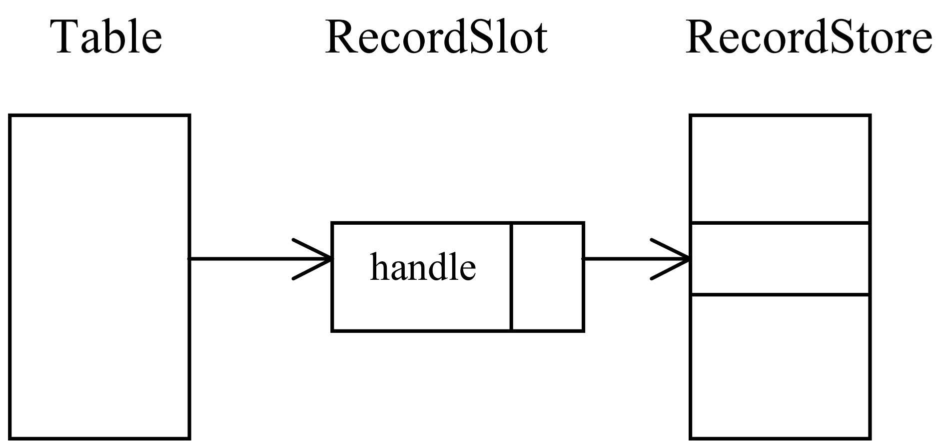
Bridge Pattern

- Decouple "interface" 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 nature/location of record slots in `RecordStore` implementation
- Introduce proxy object as surrogate for target



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 `store :: int → Object`

- represents abstract RecordStore state

Set `Handles ⊂ int`, `NULLHANDLE ∉ Handles`

Set `alloc ⊆ Handles`

- represents set of allocated slot handles

Set `unalloc = Handles - alloc`

- represents set of unallocated slot handles

Abstract predicate `isStorable(Object)`

- depends on storage mechanism (differs from Access Layer)

Invariant:

```
(∀ h, r : r = store(h) : isStorable(r)) &&
(∀ h :: h ∈ alloc ≡ defined(store(h)))
```

RecordStore Interface

`RecordSlot getSlot()`
allocates a new record slot

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

`void releaseSlot(RecordSlot slot)`
deallocates record `slot`

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() ∉ #alloc`
`&& result.getHandle() ∈ alloc ∪ {NULLHANDLE}`

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

Pre: `handle ∈ alloc`

Post: `result.getContainer() = this_RecordStore`
`&& result.getRecord() = #store(handle)`
`&& result.getHandle() = handle`

RecordStore Design Contract (2 of 2)

`void releaseSlot(RecordSlot slot)` deallocates record `slot`

Pre: `slot.getHandle() ∈ alloc && slot.getContainer() = this_RecordStore`

Post: `alloc = #alloc - {slot.getHandle()} && store = #store - {(slot.getHandle(),slot.getRecord())}`

RecordSlot Interface

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

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

`int getHandle()` returns handle of this slot

`RecordStore getContainer()` returns reference to `RecordStore` holding this slot

`boolean isEmpty()` determines whether this slot empty

RecordSlot Model

- Reference to `RecordStore` to which this `RecordSlot` belongs

- `handle` for the associated physical storage slot in the `RecordStore`

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 = getHandle():`

(`h` ∈ `#alloc` ⇒ `result = #store(h)`) &&

(`h = NULLHANDLE` ⇒ `result = NULLRECORD`)

`int getHandle()` returns handle of this slot

Pre: `true`

Post: `result =` 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`

Record Interface Note

- `Record` used by Storage Layer may be defined by either layer above
  - might be one Client Layer `Keyed` record
  - might contain more than one (or perhaps a portion of one) Client Layer record (e.g, multiway tree nodes)
- Storage Layer calls back to `Record` implementation in a layer above
  - implementation in Access Layer might call back to implementations in Client Layer

Storage Layer Challenges

Support diverse table implementations in Access Layer

- careful design of `RecordStore` and `RecordSlot` abstractions to have sufficient functionality

Allow diverse physical media (in-memory, on-disk, etc.)

- careful design of `RecordStore` abstraction to hide media details, but be implementable in many ways

Decouple implementations as much as possible

- use of `RecordSlot`, `handle`, and `Record`

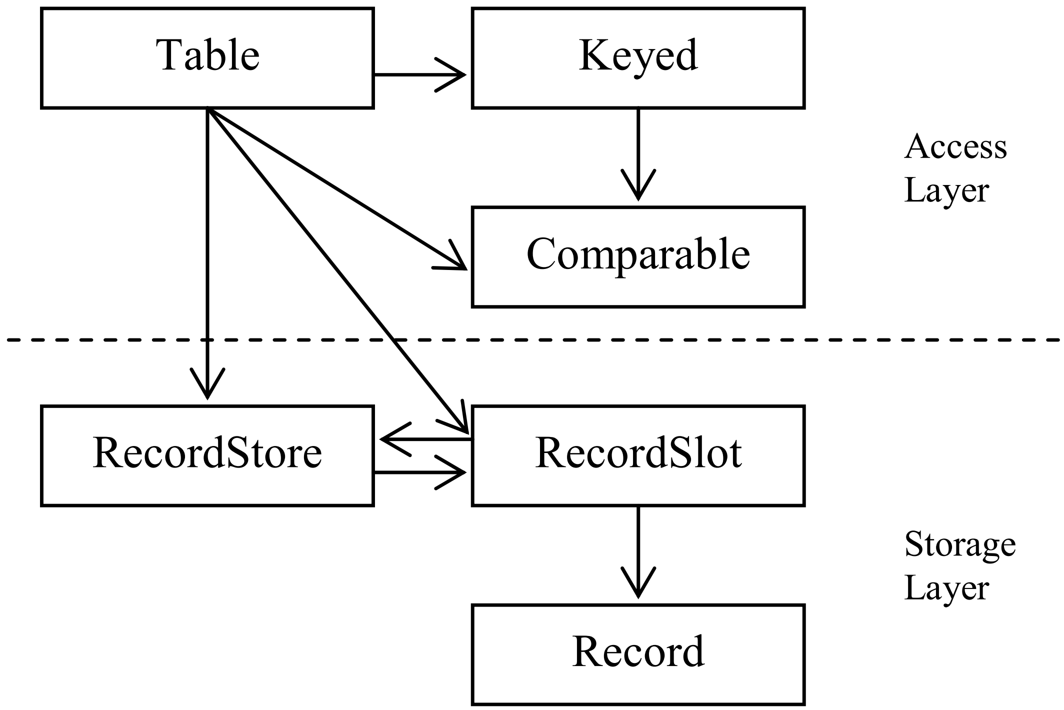
Support client-defined records

- callbacks to `Record` implementations

Enable persistence of table, including access layer

- store `RecordStore` identifier and `handles`

Abstraction Usage Relationships



Other Design Patterns Used

- 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