Applying Software Patterns in the Design of a Table Framework

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

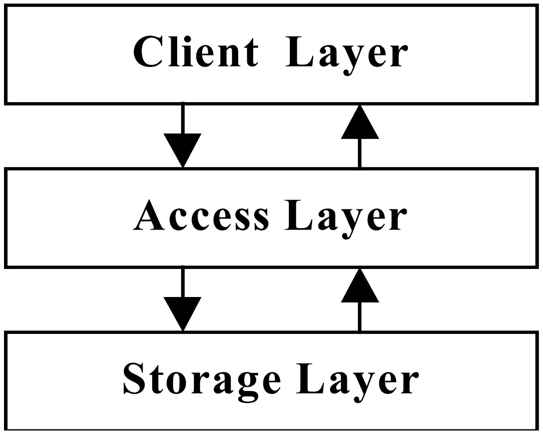
- Provide Table operations
- Support many implementations
- Separate key-based access mechanism from storage mechanism
- Present coherent abstractions with well-defined interfaces
- Use software design patterns

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

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:

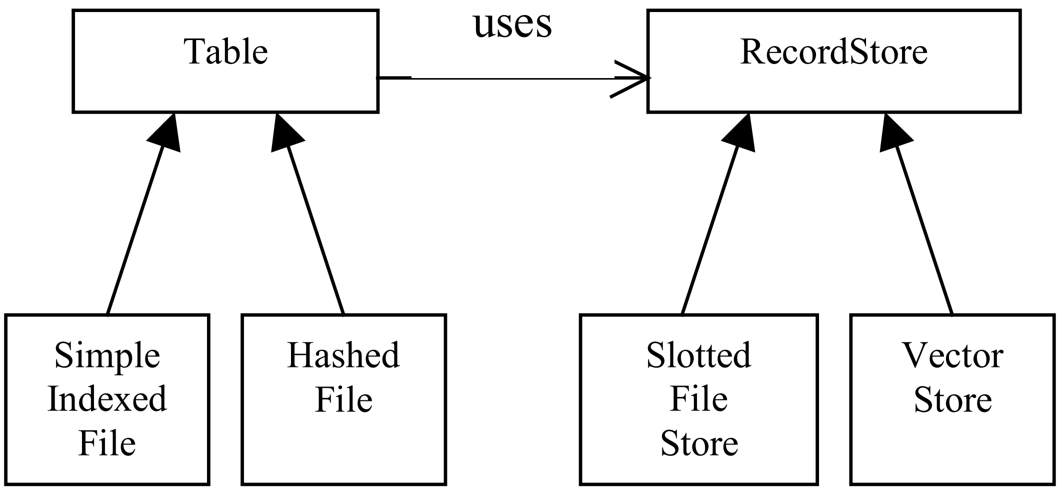
- support diverse table implementations in Access Layer (simple indexes, hashing, balanced trees, etc.)
- allow diverse physical media (in-memory, on-disk, etc.)
- enable persistence of table
- decouple implementations as much as possible
- support client-defined records

Patterns:

- Bridge
- Proxy

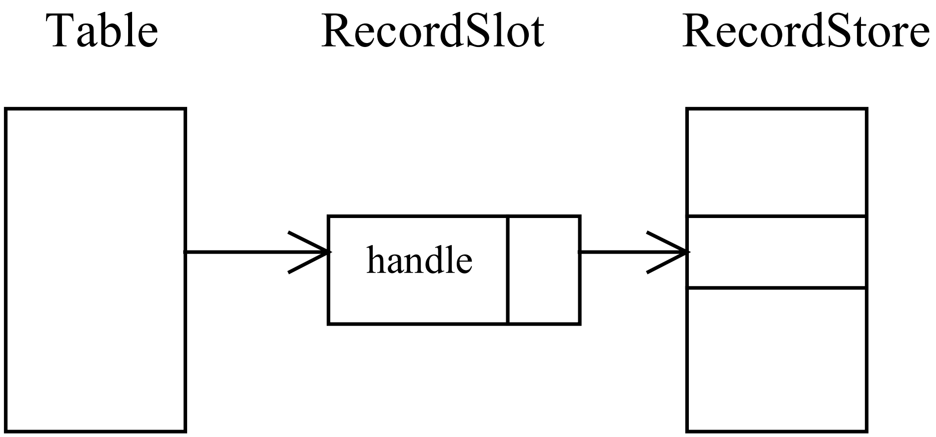
Bridge Pattern

- 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

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`

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

Record Interface

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

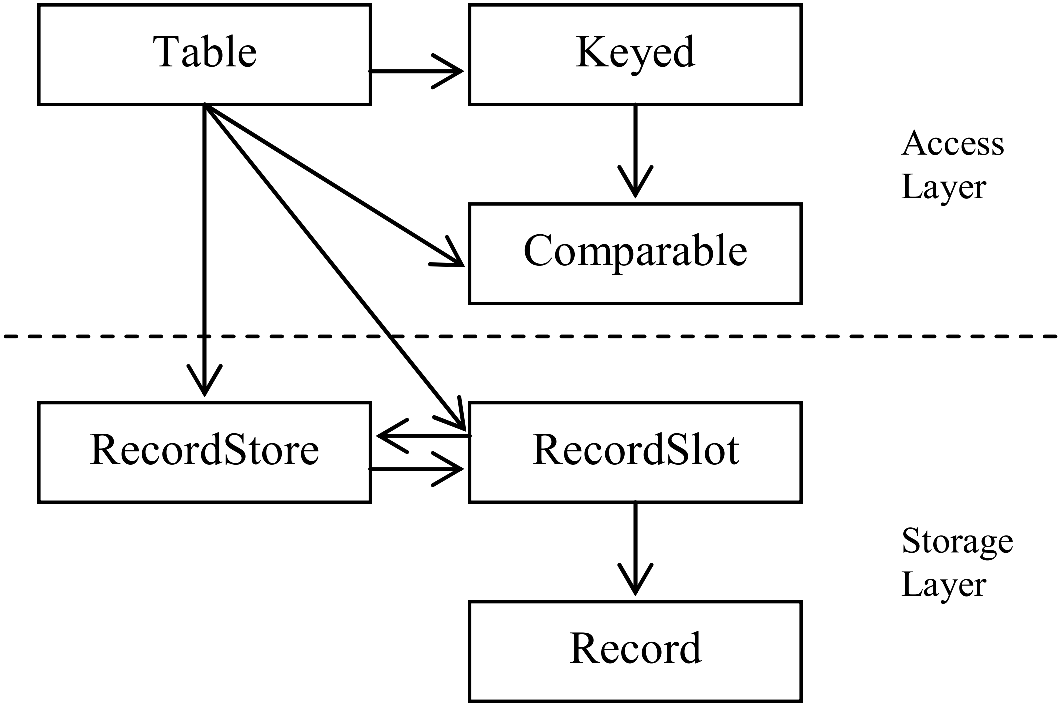
Solution: call back to client's record implementation

`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



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

Future Work

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