OATSdb: v0

OATSdb: Getting Started

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: 2020

Today's Lecture: Overview

1. Introduction & "Nuts & Bolts"
2. **OATSdb** Base Classes & Interfaces
3. VO: Functional Requirements

Introduction & “Nuts & Bolts”

Slither Before We Run

- v0 will set up basic framework for **OATSdb** implementation!
- Nothing (certainly not software) is "cast in stone", so you can always go back and fix things in subsequent milestones
  - But worth putting in the effort to start things off on a good foundation ☺
- Today's lecture provides both a general introduction to the **OATSdb** project and concepts and a discussion that's specific to the v0 milestone

The requirements document for the **OATSdb** project is posted on Piazza: read it carefully!

- Consider this lecture an elaboration of some of the points discussed in the requirements document
  - Some students prefer a “slide format”: hence this lecture
  - More importantly, the lecture enables an interactive discussion!
- However: the requirements document provides motivation and context that I simply don’t have space for in this lecture
  - The document should be considered the “master version” relative to the lecture
  - I’ll update that document with useful feedback from you

Base Repository (I)

- I've created a github repository named **OATSdb** Base
- It contains a set of base classes & interfaces for this project

  - No excuses for misspelling the APIs
  - Read the Javadoc very, very carefully: this is the "real" requirements document

  - All project milestones will (re)implement the same set of interfaces
  - Each milestone will provide an implementation with its own unique characteristics
- Repository code defines the `edu.yu.oatsdb.base` package
  - v0: you'll provide a `edu.yu.oatsdb.v0` package that implements these interfaces with "v0" semantics (today's lecture)

Base Repository (II)

- VerifyLaptops open please: that you can download the repository
  - https://github.com/Yeshiva-University-CS/OATSdb_Base
- I'd *strongly prefer* that you don't check-in this code to your code base ...
- I **insist** that you don't modify the code in any way!
  - I will test your code only through the APIs defined in this package
  - Let's not take any chances breaking that ☺

See the requirements doc for the **required** directory layout and milestone **$DIR** specification.

Logging: I

- A Pragmatic Programmer doesn't start a serious software project without a logging library
  - Logging via `println` just doesn't scale
- As usual, you can ignore this advice ☺
- That said: if you do want to use a logging framework, you must use Apache Log4j 2
  - Otherwise: I won't be able to add your non-JDK libraries to my classpath as I test your code
  - Result: will "break the build" ☹
    - With the usual implications …

Logging: II

- If you do use Log4j 2 in your code ...
- When I execute your code, your output will be generated and logged as well
  - This will aid debugging and help us understand what went wrong
  - Assuming, of course, that anything does go wrong ☺
- I'll bundle a *log4j2.properties* file in my build of your code, and link in the required libraries at runtime
- I've included a usable *log4j2.properties* file in the **OATSdb** Base repository
- Tweak as desired, but note this line:

  `logger.file.name=edu.yu.oatsdb`
  - My code execution will only emit log output for packages nested under this package
- Warning: code that will be executed frequently, should be associated with a DEBUG level or below
  - You really, really, don't want to be emitting tons of output during "production"

Some miscellaneous observations

- I try to use JDK classes whenever possible, but find the `java.util.logging` APIs and output hard to use
- Every serious programmer writes at least two logging frameworks of their own before realizing that they should use a hardened, well-known, library instead ☺
  - You might as well "pick one, and stick with it"
  - *log4j2* is as good a choice as any
  - Let me know if you discover something better
- Configuration for all logging libraries can be a pain, ditto for understanding the framework's pieces
  - Suggestion: learn how to use at a basic level and don't try anything fancy
  - The Internet is your friend, but keep in mind that I'm using *log4j2*
  - Stackoverflow is your special friend

Log4j2: Features to Exploit

- All frameworks have multiple (severity) logging levels
  - Good frameworks make it easy to enable/disable logging at package, class, and severity granularities
- Separates logging API from logging implementation
  - You can plug-and-play different implementations (e.g., SL4J, *java.util.logging* etc)
- Even has support for different logging APIs
- *log4j2* (supposedly) has good performance, and low amount of "garbage collection" stress
- Advice: keep it simple
  - Pick anything that works and focus on your code ☺
- If you use maven, here is a sample "dependency" specification

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.8.2</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.8.2</version>
</dependency>
```

Dependency On Log4j2

- Here is the one exception to the "don't modify your version of the **OATSdb** Base code"
- Because it's so easy to make a mistake when using the factory methods of *OATSDBType* ...
  - (We'll discuss these methods soon ...)
- ... I've added some Log4j2 code in that class
  - Result: the **OATSdb** Base code has a *compile* and *runtime* dependency on Log4j2
- If you don't want to have a dependency on this framework, feel free to "comment out" the relevant lines of code
  - Be careful to not check this change into Git

Test Suites

- You don't have to write a single test ☺
- But: if you don't have a robust test suite, I can almost guarantee that you will fail ☹
- Key point: a Pragmatic Programmer doesn't start a serious software project without including a test suite
- For each milestone, I'll discuss some "test coverage" ideas that you may want to implement
  - That list will not be exhaustive!
  - You are responsible for translating requirements into tests

Test Suite Rules & Suggestions

- Because of my automation scripts, your code can depend on only: the JDK, log4j2, and JUnit 4
- You are more than welcome to (additionally) set up a separate code base that uses other dependencies
  - Meaning: you can develop in an environment customized to your taste & needs
  - The code that you check into the "**OATSdb** tree" however: cannot rely on these extra dependencies
- Also: if you include test code in the repository code base, you are responsible for not "breaking the build"
- I also suggest that your tests be in a different package from the "package under test"
  - This ensures that you're only testing public APIs
- Finally: keep your tests segregated by milestone: e.g., "v0" test package, "v1"
  - This will considerably simplify the complexity

You Have (Almost) Total Implementation Freedom!

This is all you have to provide ☺

```
public enum DBMSImpl implements DBMS {
  Instance;

  // Implement DBMS Interface
}

public enum TxMgrImpl implements TxMgr {
  Instance;

  // Implement TxMgr interface
}
```

- Everything else should be “private” or “package-private”
- I will use the `OATSDBType` factory methods to instantiate your implementation

```
public static DBMS dbmsFactory(OATSDBType oatsdbType) {}
public static TxMgr txMgrFactory(OATSDBType oatsdbType) {}
```

OATSdb Base Classes & Interfaces

(Trivial) Exception Classes

- The base interface classes reference these Exception classes
- The Javadoc is self-explanatory: your task is to throw the correct Exception at the appropriate time
- The list:
  - `ClientNotInTxException.java`
  - `ClientTxRolledBackException.java`
  - `NotSupportedException.java`
  - `RollbackException.java`
  - `SystemException.java`

OATSDBType: Our "Factory" Class

- OATSDBType declares an enum specifying values for supported **OATSdb** versions (milestones)
- Note the factory methods!
- You'll use these methods to instantiate the *DBMS* and *TxMgr* singleton Instance implementations
  - And I will use these methods as I test your code
- The purpose of these factory methods is to decouple the base class from linkage to implementation code (such as vo)
  - Instead: "instantiate by name" on demand ☺
  - We're using the magic of Reflection
  - Specifically: we're using the Bloch "enum as Singleton pattern idea)
  - Bloch's Effective Java worth reading!

v0: Ignore These Interfaces

- The following interfaces will be used in later milestones
- Ignore them for v0
  - `ConfigurableDBMS.java`
  - `ConfigurablePersistentDBMS.java`

At any moment, an **OATSdb** client is or is not in a transaction

```
// is this code in a transaction?
final String mapName = UUID.randomUUID().toString();
final String key = "the question";
final int value = 42;

// what about this code?
final Map<String, Integer> map1 =
  dbms.createMap(mapName, String.class, Integer.class);
map1.put(key, value);
```

- Vanilla Java code doesn't have to be (it's up to you) in a transaction
  - Example: the first code snippet
- Any code that invokes a *DBMS* API must be in a transaction
- And: all Map (container) methods must also be in a transaction
  - Example: the second code snippet

```
txMgr.begin();
createAccounts(nAccounts);  // application specific code
txMgr.commit();
```

Transaction model:

- Client writes application code
- Wraps application code in a transaction
- How?
  - `txMgr.begin` demarcates the start of the transaction boundary
  - Either: `txMgr.commit` demarcates the end of the transaction boundary
  - Or: `txMgr.rollback` demarcates the end of the transaction boundary
- You had better "close that transaction" one way or the other
  - Or subsequent code will continue to execute in that transaction context

Commit versus Rollback

- Both methods terminate the transaction
  - Meaning: you'll have to start a new transaction afterwards if you want to wrap subsequent code in a transaction
- `commit` makes the work done by the application visible to other clients and transactions
  - Meaning: persistent beyond this transaction's work
- `rollback` "undoes" the work done by the application
  - Will not be visible to other clients and transactions
  - Not even visible to the client that rolled back the transaction!

- The visibility semantics are not for vo!
- vo only responsible for providing the appropriate transaction boundary semantics

TxCompletionStatus *Versus* TxStatus (I)

- The **OATSdb** representation of a "transaction" is the *Tx* interface
  - This interface is minimal: only two methods, one returns the *TxStatus* and the other returns the *TxCompletionStatus*
- *TxStatus*: represents the possible "life-cycle states" that a transaction may be assigned
  - Very much a point in time statement: the next nano-second may move a transaction from e.g., ACTIVE to COMMITTING
- *TxCompletionStatus*: represents a more "stable" view of a transaction
  - Either it's COMMITTED, ROLLEDBACK, or NOT_COMPLETED
  - Under normal conditions, you'd expect that last state to change to one of the first two pretty quickly
- *TxCompletionStatus.java* includes static utility methods that map a *TxCompletionStatus* to a *TxStatus*

TxCompletionStatus *Versus* TxStatus (II)

- Clients can only access the current transaction (if any) through the `TxMgr.getTx` method
- This fact, combined with the "minimality" of the *Tx* interface, implies that you have (almost) complete freedom in how you choose to implement a "transaction" ☺
  - Suggestion: start out "bare bones", add internal function as needed
- Your *TxMgr* is responsible for driving a client's *Tx* through the appropriate *TxStatus* and *TxCompletionStatus* states
  - Based on the transaction "demarcation" methods we discussed previously
- As I test your code, I'll be verifying that your *Tx* implementation correctly reflects the state that's required to be associated with a transaction

Test Failures & Cascading Errors

- Consider the implications of a test failure ...
  - This is relevant to the "other guy", not you ☺
- The transaction associated with your client thread is likely in an indeterminate state
  - At the least: your internal data-structures may be corrupted
- My test suite will therefore make an attempt to `rollback` your transaction
- This enables the next test to begin with a clean state
- Key point: your implementation of `rollback` (and `commit`) must do all the necessary data-structure cleanup!
  - Failure to do this properly (even for vo) will indeed result in cascading errors ☹
- Since I use *TxMgr.getTx* and *Tx.getStatus* to determine whether it's safe to issue a `rollback`, make sure that you provide a good implementation!

Association Between Client Thread & Transaction

The Rules:

- A client thread can be associated with at most one transaction
- A client thread can only be associated with a transaction if it invoked `TxMgr.begin` and has not yet invoked `txMgr.commit` or `txMgr.rollback`
- Your server implementation must allow for multiple client threads to access its methods concurrently
- When a client invokes `TxMgr.begin`, your server is therefore responsible for creating a unique association between a given client thread and a *Tx* instance
- Your server is responsible for removing that association upon invocation of *TxMgr.commit* or *TxMgr.rollback*
- The implementation details of this process are up to you: but you must provide these semantics!

- Provides methods to “create” and “retrieve” named Maps
- Remember: (Generic) Maps are the **OATSdb** “table”
- MapEntries are the **OATSdb** “rows”
- Q: why do the Maps have to be named?
  - Vanilla Java maps are not named …
- A: In vanilla Java code, you use references to distinguish between two Map instances
  - The purpose of **OATSdb** named Maps is to allow retrieval of previously persisted Maps
  - Where the original Map reference is no longer available
- Naming is even more important since your server must be able to distinguish between two Maps of the same type

OATSdb Maps Are Generic!

- In your cs career, so far, you may have only used generics as a consumer
- Now you'll be writing your own generic classes!
- I urge you to get up to speed on this important topic
  - You need to understand the semantics, not just the syntax
  - This is a good tutorial
  - If it doesn't work for you, get a good textbook, use the Internet etc

```
Map<Long, Account> savings =
      dbms.createMap(savingAccounts, Long.class, Account
          .class);
Map<String, Account> namedSavings =
    dbms.getMap(namedSavingAccounts, String.class,
        Account.class);
```

DBMS: Note the API's Exceptions

- Conceptually: these Maps have an existence that is independent of your program's (short-lived) execution
- Therefore
  - `createMap` will throw *IllegalArgumentException* if "name is already in use"
    - No subtlety here: exception thrown even if name is bound to a Map of a different type
  - `getMap` will throw *NoSuchElementException* if (named) Map does not exist

Database versus Transactions

- These are conceptually orthogonal ideas
- Database: store the data and provide APIs to access that data
- Transactions: protect the DBMS data from corruption created by concurrent multiple threads
- The interaction between the *DBMS* and *TxMgr* occurs because **OATSdb** requires that all database data must be accessed inside a transaction's scope
  - This is how subsequent milestones will enable concurrency and isolation properties
- Hence the `ClientNotInTxException` declared on the DBMS methods
- Also: clients may only perform an operation on Map data when inside a transaction
  - The DBMS is responsible for throwing `ClientNotInTxException` if this requirement is violated

Transactions Are Good: Nested Transactions Are Bad

- There are well-defined nested transaction models
- **OATSdb** does not support any nested transaction model
  - Too much work ☺
  - Value (can be) dubious
- So: `begin` must throw a *NotSupportedException* if client begins a tx while already in a tx
- `commit` and `rollback` must throw *IllegalStateException* if client is not already in a tx

Back To The Programming Model

- The Maps returned by DBMS are `java.util.Map` instances
  - With a (possibly arbitrary) subsetting of the API
  - See below
- Use `put` to insert items
- Use `get` to retrieve items
- Use `remove` to remove items
- Conceptually: you'll be providing a "pass-through" implementation to a vanilla `java.util.Map`
  - Not saying that you must provide this implementation!

- The set create, read, update and delete database operations are affectionally known as CRUD
- These methods are the “Hello World” of database programming
- Many well-known “application builders” built their fame on making CRUD simple (especially for the web)
  - Ruby On Rails
  - Node.js + Express + MongoDB
  - No time now for snark ☺
- Q: where is the update method in the **OATSdb** API?

Updating a Map.Entry: Single-Level Store Semantics

- In **OATSdb**, an "update" is done by changing a Map.Entry's value field
  1. Acquire a reference to a Map.Entry through `get` or `put`
  2. Update the Map.Entry value field by changing its state
  3. Commit the transaction
- Key point: your main-memory object reference and the corresponding database reference are (conceptually) pointing to the same object!
  - Sometimes referred to as Single-level-store semantics
- Makes your life as an **OATSdb** provider trickier ☹
  - Because you must ensure that state changes made to any Map.Entry are visible to all clients that have acquired a reference to that Map.Entry

Associate Clients With Transactions

- You have to provide some magic ☺
- When a client invokes a Map method, **OATSdb** must know:
  - Whether the client is associated with a transaction?
  - What transaction the client is associated with?
  - What objects is the transaction associated with?
- Investigate: Thread.current Thread
- Investigate: ThreadLocal
- The former enables you to get your client's execution thread (so doesn't have to be supplied by the client explicitly)
- The latter enables you to group (arbitrary) data on a per-Thread basis

v0: Functional Requirements

vo: Main-Memory Database + Transaction Interface

vo provides:

- A functional "main-memory" database with supporting CRUD operations using `java.util.Map` API
  - Client can create named generic Maps
  - Client retrieve generic Maps by name
- No persistence: all data disappears when your process finishes!
- A transactional interface
  - You're not providing atomicity or isolation
  - In other words: vo TxMgr methods that demarcate transaction boundaries methods are "no-ops" from the perspective of ACID semantics
  - They don't change the state of the database in any way
- vo does implement transactional boundaries
- **OATSdb** vo does enforce the semantics that all DBMS methods must be accessed in a transaction
- **OATSdb** vo does enforce the semantics that all Map methods must be accessed in a transaction

vo: What's the Point?

You'll start implementing:

- DBMS: a "container of containers"
  - That is: must maintain all Maps created by the client
  - Identifying them by name
  - And: in a generic way
- TxMgr: map client threads to Tx instances
- You will become (more) familiar with Java generics
- You will become (more) familiar with Java Threads
- You will start building your test suites
- Note: container concept
- Client codes to "POJO" model, but container (**OATSdb**) is transparently mapping to more powerful/complicated implementation
  - Example: Map interface knows nothing about transactions, but your vo Map implementation requires that all clients invoking methods on it be associated with a transaction

You'll build on vo implementation for subsequent milestones

Container of Containers

- You must make provision for client asking you to create (and store) an arbitrary generic Map
- You need a Map that associates a Map name with arbitrary generic – not raw – Map instances
- You need to store the “key class” and “value class” information
  - So you can validate that the client’s request can be met by the Map you’re storing for her
- Advice
  - Investigate generic Wildcards
  - Investigate “heterogeneous containers”
- Some useful pointers
  - Bloch, Effective Java (2nd edition) #29 ”consider typesafe heterogeneous containers”
  - Bloch, Effective Java (2nd edition) #3: “Enforce the singleton property with a private constructor or an enum type”

Proxy Design Pattern

- `java.util.Map` is an interface
- You may choose to write the **OATSdb** implementation class from scratch
  - Remember: you're returning a Map instance to your client
  - Must behave (at least for CRUD interface) like a Map
- But must be more than a Map!
  - At minimum: must be "transaction aware"
- Suggestion: investigate the Proxy design pattern
  - Java support for this pattern
- Lots of articles: e.g., this one

Implementation Rules

- You can & should use ideas in textbooks, Internet, videos
  - Be professional: document where you got your idea(s) from & how/whether you modified/extended
- You absolutely may not share ideas with fellow students
  - Code *a fortiori*
- There is no one true way!
  - I've done a "existence proof" that **OATSdb** can be implemented
  - You may well devise a better implementation

Things To Test For: I

DBMS Happy Path

- Create a Map
- Retrieve a Map
- Suggest your test includes at least two Maps to make sure you're not cheating ☺
- `put`, then `get` Map.Entry instances

DBMS Edge Cases

- Create a Map not in a transaction
- Retrieve a Map not in a transaction
- CRUD not in a transaction
- What happens if you specify different key & value class for `getMap` than what you supplied for `createMap`?
- What does `createMap` do if that Map already exists?
- What does `getMap` do if that Map does not already exist?
- What if Map name is empty?

Things To Test For: III

TxMgr

- Does **OATSdb** enforce “client must be in a transaction” for relevant TxMgr methods?
- When a client does a commit or rollback: is that thread removed from the transaction context?
- Does **OATSdb** enforce “no nested transactions”?

Note: Not much else to test for vO transactions ...

For the Adventurous

- VO does not provide ACID semantics
- Can you write code that proves that?
  - Easy: conceptually ☺
  - Hard: Computers are so fast that may not be trivial to write (efficient) test case that demonstrates “no transactions”

Suggestions For Getting Started (I)

- Even vO may seem initially daunting, so here are some suggestions (feel free to ignore ☺) that may help you get started
- Spend some time writing "throw-away" ("experimental") code
  - Meaning: small chunks of code that are not integrated into an official package or class structure
- Some examples
  - Create a "generic container of containers"
  - Create a new generic Map and store it in that container
  - Retrieve a generic Map instance after it's been stored it in that container of containers
- Other examples
  - How do I access a client's thread?
  - Can I associate state with that thread for subsequent use?
  - Can I create a Map implementation class that behaves like a vanilla main-memory Map and yet allows me to do something different than a vanilla Map
    - Trivial function proves the point: print "hi mom" when `get` is invoked, print "hi dad" when `put` is invoked

Suggestions For Getting Started (II)

- In parallel with the previous activities
  - As you experiment with "getting a client's thread", create two client threads ...
    - Have each thread access the same piece of code in another class
    - Can the code in that other class distinguish between those two threads?
- Investigate the use of *java.util.concurrent.Future* interface (and related implementations) to create multiple threads
  - These "multiple threads" are your "multiple clients"
  - Have those threads access some other class
- Use "throw-away" code to quickly determine whether you're on the right track!
- Read the Javadoc that I wrote for the **OATSdb** Base repository carefully!
- Read the references included in the lecture before you write any serious code: why reinvent the wheel?
  - Or at least figure out why you're reinventing the wheel ☺

Suggestions For Getting Started (II)

- Create a set of test method names (no implementation at first)
  - This enables you to express "what am I planning to build?"
  - We call this test driven development
- Once you have these "micro pieces" in place, you'll have reasonable confidence that you can supply the vo **OATSdb** functional requirements
- Only afterward should you set up the **OATSdb** packages and start integrating the "micro pieces" into an implementation of the **OATSdb** interface

Today's Lecture: Wrapping it Up

Introduction & "Nuts & Bolts"

OATSdb Base Classes & Interfaces

vO: Functional Requirements