OATSdb Project: Requirements Document

Avraham Leff

COM3563: Fall 2020

1 Milestones For Fall 2020

1. **V0**: Sep 25, 10am
2. **V1**: Nov 30, 10am
3. **V2**: Dec 28, 10am

2 Introduction & Motivation

OATSdb is a pedagogical, **non-**relational database. This point immediately raises the question of its appropriateness for our course!

> This course looks at both the use and implementation of relational databases. It starts with a rigorous but fast-paced introduction to the relational model, schemas, indexes, views, SQL, ACID properties, and transactions. The course continues with its primary focus - database implementation ...

The answer is that a programming project ("learn by doing") for our course has to deal with the fundamental fact that relational databases are too complex to implement in a single semester!

One approach would be to have your project on *one feature* of a relational database. Possibilities include:

- Query parser
- Query optimizer
- Concurrency control

- Transaction management
- Buffer management
- Failure recovery

For this semester at least, I haven't gone with this approach because it's almost impossible to separate these features from one another (from a system implementation perspective). Because I don't want to experiment with a large-scale "team project", I've rejected the "one feature" approach.

Another approach is to have the programming project leverage an existing open-source database such as SQLite, Firebird, or Apache Derby. I haven't gone with this approach because dissecting an existing code-base takes a long time, especially if you need to understand it well enough to extend it!

2.1 The OATSdb Approach

Instead, **OATSdb** will extend your existing knowledge of the Java language and Java object model into the transactional and persistence domains. It's scope is sufficiently small to accomplish in one semester. You'll own every bit of that code, and understand every design decision that went into its implementation ☺. Database techniques won't be "abstractions": you'll use them only because you'll decide that you need those techniques.

As mentioned above, **OATSdb** does have a disadvantage: the Java object model is non-relational! Yes, this means that there is a large gap between the **OATSdb** programming model (more below) and a "real" database model. On the other hand, you'll still be using (simplified) versions of the algorithms and techniques from the "real" relational word. Another benefit: you'll learn to appreciate the real advantages of the relational database model.

3 OATSdb Programming Model

3.1 Terminology

- A *programming language* allows you to implement a software feature
- An *application programming interface* (API) specifies the way that clients interact with specific parts of your software

- A *programming model* is a higher level of abstraction than an API. It specifies the "basic concepts" that are common to all your system's software

Some programming model examples:

- The *relational data model* is a programming model for databases.
- The *Von Neumann* model is a programming model used to define how "traditional" sequential computers operate.
- The *shared memory* model defines how multiple threads can compute & communicate concurrently.

The advantage of having a programming model is that it provides a consistent "look & feel" for your software. Adding new features is facilitated because it's easier to see where the feature fits in to the broader picture. Perhaps most importantly: because the programming model has already given them the high-level picture, clients understand how to use your system. They only have to read documentation to get the "details" of a specific API call! For example: given the relational data model, regardless of the details of a given relation (e.g., "payroll" versus "enrollment"), all relational data "behave" the same way.

3.2 OATSdb: Generalize `java.util.Map`

You are already familiar with three types of object-oriented software entities: *classes* (a "type"), *instances* of those types, and *collections* (of instances). Collections can be thought of as "containers" that hold aggregations of these instances.

Any modern programming language has built-in or library support for "array", "list", "set" and "map" collections. From the relational data model perspective: we have "schema" (the classes), "tables" (the containers) and "rows" (the instances).

OATSdb would use `java.util.Collection` as its "container", but it's hard to "query" a Collection. **OATSdb** there uses `java.util.Map` as its container API. The generic key and value template specifies the "class(es)" associated with a given Map container. The Map `put`, `get`, and `remove` API serve as the CRUD equivalent.

The main advantage of this programming model is that you're already familiar with java.util.Map (and its a concept that's common to many other

programming languages). Note: we're going to use exactly that interface, "as is"! Another advantage: we're not going to invent a query language. Instead, we'll use a "query by primary key" analog: the `get` API.

3.3 OATSdb "Value Add"

If the **OATSdb** programming model is simply "Map", what benefits does it provide? Use a vanilla `java.util.Map` implementation instead!

In a nutshell: **OATSdb** provides a *transactional implementation* of its programming model. As will be discussed in lecture (and see "Part Seven" in our textbook), transactions provide the classic ACID benefits.

Quick summary:

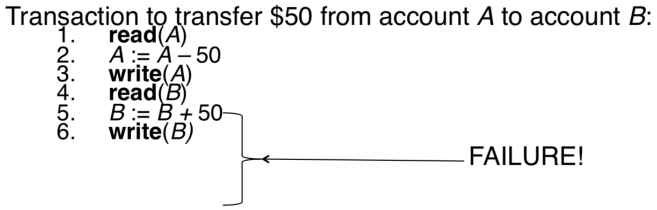
- **A**tomicity: either all operations in a transaction take effect, or none take effect
- **C**onsistency: operations, taken together, preserve db consistency
- **I**solation: intermediate, and possibly inconsistent states, are not visible to other transactions
- **D**urability: if a transaction successfully completes ("commits"), the changes made in the transaction to the db must persist – even if the system crashes.

3.4 Persistence

Consider a Map in which you've inserted three values, after which your program crashes. If you restart your program and access that Map, *durability* implies that your Map (and its objects) will be blessed with persistence. We won't be provide "persistence" beyond state that can be captured by your laptop's disks. In other words: we'll supply persistence that survives *main-memory* failures.

However, while "persistence" is a good feature, its benefits really shine when your application can exploit *transactional boundaries* and, within those boundaries acquire *atomicity* and *isolation* properties. Consider familiar programming language granularities such as "statement", "loop", "method": what if you want to express the notion of *"at a given granularity, I want the semantics of its execution to be 'all or nothing"'*? That is: either the code's effect will succeed in its entirety or no effect of that code will be visible. Vanilla programming language do not provide this capability, and its traditionally left for databases to support via transactions. We can

think of a transaction as providing a "code granularity" with this valuable "all or nothing" feature.



In this example, the transaction could be a transfer from my checking account to your checking account (both accounts in the same Map). Or: the transaction could be a transfer from my *savings* account to my *checking* account (i.e., the two accounts are in two Maps).

3.5 Atomicity

Atomicity means that transaction code is never left in a "partially completed" state. Regardless of what caused the failure (data consistency problems, power failures etc), if the transaction fails after *statement 3* but before *statement 6*, the write operation performed in *statement 3* will not be visible in the database. That is: balances in the two accounts will be unchanged when compared to the beginning of the transaction's execution.

3.6 Consistency

Consistency is a "fuzzy" notion because it's application-specific. For example: you may want the database to enforce an integrity constraint stating that *"balances can not go below zero"*. Or, you may want the database to enforce an integrity constraint that *"total value of account balances $A + B$ will be unchanged by the transaction"*. Pardon the circularity, but the consistency property states that transactions will leave the database in a consistent state with respect to these constraints ☺A less circular way of defining consistency is: *"all data will meet all constraints that have been placed on the data"*.

Let's make life easier for you: **OATSdb** will place the *consistency* responsibility on the application programmer ☺(not you, the systems programmer). Note the advantages of relational database technology where these features can be supplied by the DBMS.

3.7 Isolation

Referring back to Figure 3.4, the *isolation* property means that this transaction executes in isolation from other transactions. Specifically: other transactions will be unable to *write* ("change the state") of these accounts while statements $1 \rightarrow 6$ are executing. Later, we'll decide whether other transactions can *even read* the changing account balances in either A or B. This property is tied very closely to the topic of *concurrency control* (Chapter 18 in the textbook).

3.8 Durability

This property means that, after a transaction completes successfully, the changes it has made to the database persist – even if there are subsequent system failures. Clients are guaranteed: the transaction will not be *rolled back*.

3.9 What You'll Be Implementing

- Atomicity: "yes"
- Consistency: "no" (as discussed above)
- Isolation: "yes", we'll begin with a badly performing implementation, and see if we can improve this later.
- Durability: "yes" (in later project milestones) for main-memory failures.

3.10 Code Sample

To make the **OATSdb** programming model more concrete, consider the following code:

```
txMgr.begin();

Map<Long, Account> savingsMap = dbms.
  getMap(savingAccounts, Long.class, Account.class);
Map<Integer, Account> checkingMap = dbms.
  getMap(checkingAccounts, Integer.class, Account.
     class);
```

```
final Account savings = AccountFactory.Instance.
  create(userName1, 120, SAVINGS);
final Account checking = AccountFactory.Instance.
  create(userName1, 20, CHECKING);

savingsMap.put(savings.getAccountId(), savings);
checkingMap.put((int) checking.getAccountId(),
   checking);
txMgr.commit();
```

Note: Do you see how the **OATSdb** programming model is identical to the java.util.Map programming model?

Yes, you must begin and commit a transaction (to define the boundaries of your "unit of work"). Yes, you must access the database manager to get a Map ("container") reference. But, aside from these "non-vanilla-Java" activities, the container API is identical to the (selected subset) `java.util.Map` API. Even more importantly, the "object instance API" is identical to the "vanilla Java instance" API.

This point is so important that it deserves its own discussion.

4 POJOs: Another OATSdb Feature

> An acronym for: Plain Old Java Object.
>
> The term was coined while Rebecca Parsons, Josh MacKenzie and I were preparing for a talk at a conference in September 2000. In the talk we were pointing out the many benefits of encoding business logic into regular java objects rather than using Entity Beans. We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely.
>
> Martin Fowler

OATSdb allows you to code your regular Java objects, store into, and retrieve from your Map

Let's say you've developed an *Account* object, and now want to enhance it with wonderful transactional properties that we've been discussing? That is to say: when you use an *Account* instance in code that's inside transaction boundaries, you want changes made to this object to have ACID behavior. You don't want to have to add those properties: you want the DBMS (or some other software) to supply these properties.

In the early 2000s, the way to achieve this goal required that

1. The *Account* object "extend" or "implement" special server-side base classes

2. This modified class be "deployed" in a special step that would generate the code needed to provide the new behaviors.

Examples of such middleware include Enterprise JavaBeans (v1), Microsoft DCOM, and Java RMI.

Application programmers disliked this complexity: their job, after all, was to provide *business logic*, not to get involved with *server-side* function. However: application programmers had no choice but to go along in order to get benefits such as transactions, security, and naming services. The "contract" stated: if follow the server-side *component model* rules, you'll be rewarded with these services "auto-magically". But, ultimately, application programmers rebelled against this too complex, and too slow (topic for several lectures, not this course) process. The result was the POJO revolution ☺

4.1 OATSdb pojo Semantics

Per Wikipedia, a POJO **should not have to** extend pre-specified classes, as in:

```
public class Foo extends javax.servlet.http.
   HttpServlet {}
```

Nor should the POJO have to implement prespecified interfaces, as in:

```
public class Bar implements javax.ejb.EntityBean {}
```

Nor should the POJO have to use prespecified annotations, as in:

```
@javax.persistence.Entity public class Baz {}
```

Obligatory snark: many software frameworks still cheat a little in this regard ☺

We're going to make the attempt at a "pure POJO" approach. That means that a client of your DBMS must be allowed to deploy ("run") **any legal Java class, "as is", with no modifications on her part** into the DBMS that you'll be providing. After such deployment, her vanilla Java ("non-transactional") code that used those classes will continue to have the **same semantics**! The only difference is that, because she has incorporated this code in a transaction, that unit of work will now **also** have ACID semantics.

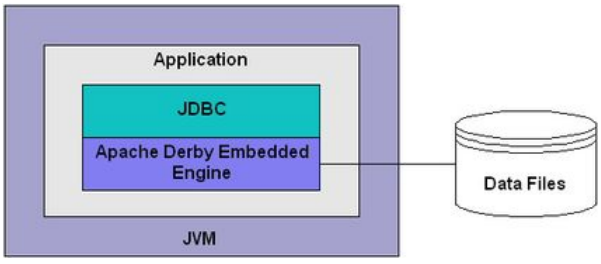
Your Map containers will have the sub-set API syntax and semantics of `java.util.Map` that we've already discussed. Clients will be able to use your Maps as containers for there vanilla Java objects.

> Our challenge: somehow endow these Maps and POJO objects with transactional capabilities!

4.2 OATSdb: An Embedded Database

An *embedded database* is a database that **runs in the same process as the application that's using it**. You may have already heard of some example, such as SQLite.

> SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files.
> About SQLite



Other examples include Apache Derby.

> When an application accesses a Derby database using the Embedded Derby JDBC driver, the Derby engine does not run in a separate process, and there are no separate database processes to start up and shut down. Instead, the Derby database engine runs inside the same Java Virtual Machine (JVM) as the application. So, Derby becomes part of the application just like any other jar file that the application use.

OATSdb is also an embedded database. Granted, the ability to have multiple clients connect to the same database instance is a useful feature. And, having the database run in a separate process from your application makes the database more robust. However: I feel that the added complexity (and work!) is not worth it (in this iteration). We'd have to (for example) add an HTTP based API on top of our in-process API. While this isn't that difficult conceptually (see e.g., Java's "servlet model"), there are more important issues from a "database implementation" perspective that we need to focus on.

5 Getting Started

- Download the base classes and interfaces from this repository.
- I'd prefer that you don't check-this code into your repo, but if you decide to check the code in, **take care to make absolutely no changes to this code**! I will be testing your implementation using only the original versions of these classes and interfaces.
- Read the Javadoc carefully!
- I will use the `OATSDBType` factory methods to create instances of your implementations of the *DBMS* and *TxMgr* interfaces. I urge you to do the same during development and test rather than using some internal method.

6 V0: Transactionally "Aware" Implementation

> I've allocated an entire lecture for a presentation of the *V0* milestone: be sure to have a look at those slides in addition to the material below.

For the *V0* milestone, you will provide implementations of the following interfaces:

- *DBMS*
- *Tx*
- *TxMgr*
- Your implementations of the *DBMS* and *TxMgr* classes **must be** in the `edu.yu.oatsdb.v0` package. If your implementation references code in other packages, their use must be "transparent" to my method invocations.
- Take a look at *OATSDBType.java*: I will instantiate *DBMS* and *TxMgr* singleton implementations from your code base. The approach I'm taking (to give you maximum flexibility) does require that your implementation classes have the following structure:

```
public enum DBMSImpl implements DBMS {
  Instance;

  // your code goes here
}

public enum TxMgrImpl implements TxMgr {
  Instance;

  // your code goes here
}
```

  Note: there are no additional requirements on the implementation structure of your classes!

- The *$DIR* is: `DatabaseImplementation/projects/oatsdb/V0`. Note that "V0" is capitalized, but the package name is not.

Your implementation must allow clients to create maps and then use them as containers for Java objects via the `put`, `get`, `remove` API. Such function is not necessarily trivial, because Map creation and retrieval must be done from the *DBMS* APIs: you can't just "new up a Map" as you do in regular Java programming.

Your *V0* implementation, must in addition, provide the following semantics.

6.1 Semantics: "Transactionally Aware"

Although this milestone doesn't provide transactional (ACID) semantics, your implementation does provide "transactional awareness". Specifically: your APIs allow me:

- To determine *whether or not* an **OATSdb** client is currently in a transaction
- To access the client's transaction object (*Tx*) and use that to determine the *state* of the client's transaction
- To drive the transaction life-cycle (see *TxStatus*) by invoking the appropriate *TxMgr* methods.

In addition: you must implement the required semantics of **OATSdb** operations only being valid within a transaction.

6.2 Semantics: "Map and Map.Entry References"

- Map references and their Map.Entry contents, that are created via the *DBMS* API persist between transactions.
- Single-level-store semantics: specifically, changes made to the *original object* are reflected in the object stored in the Map and *vice versa*.
  - Note: subsequent milestones will change this single-level-store requirement!
- The DBMS must throw the appropriate Exception if a client uses a Map reference outside a transaction.
- The DBMS is not responsible for throwing an exception if a client uses a *Map.Entry* reference outside a transaction.

6.3 Semantics of Null Keys or Values

In *V0*, keys and values are allowed to be NULL.

Important: in *V1* and *V2*, although values are allowed to be null, **keys may not be null**.

6.4 Transaction State Transitions

I suggest that you draw state-diagrams *TxStatus* and *TxCompletionStatus*, and think through the implications of the API invocations on the required state transitions.

6.5 Rubric

Assuming that your code builds successfully:

- 30%: does your DBMS responds correctly to a set of basic API calls ("does it run")?
- 70%: JUnit tests ("does it run correctly?")

7 V1: Atomicity & Isolation

> I've allocated an entire lecture for a presentation of the *V1* milestone: be sure to have a look at those slides in addition to the material below.

As with the *V0* milestone, for the *V1* milestone, you will provide implementations of the following interfaces:

- *DBMS*, *Tx*, *TxMgr*
- Your implementations of the *DBMS* and *TxMgr* classes **must be** in the `edu.yu.oatsdb.v1` package. If your implementation references code in other packages, their use must be "transparent" to my method invocations.
- The *$DIR* is: `DatabaseImplementation/projects/oatsdb/V1`. Note that "V1" is capitalized, but the package name is not.
- Refer back to the *V0* for descriptions of how your classes **must** package their implementations of the implementations and how I will instantiate your classes. The only difference is the use the "V1" enum value instead of the "V0" value.

This milestone builds on the previous milestone's semantics. Previously, your DBMS was "transactionally aware" but didn't provide any transactional capabilities. Now, you will provide clients with *atomicity* and *isolation* function. This important change will be made with (almost) **no changes to the interface** ☺

7.1 Atomicity

As with V0, all DBMS work must be performed in a transaction, and the transaction "state transition" requirements carry over to V1. Similarly, all programming model semantics are unchanged.

Now, consider two transactions, tx_1 and tx_2 that execute *serially* (concurrent execution semantics are described in the "Isolation" section below).

In V1, we require that:

- Work performed in tx_1 is only visible to tx_2 *after* tx_1 *commits*
- If tx_1 fails or is rolled back, its work *will not be visible to* tx_2

7.2 Semantics: "Map and Map.Entry References"

- Map references that are created via the *DBMS* API remain valid between transactions. In other words, a Map reference acquired in tx_1 does not have to be "re-acquired" in tx_2: the previously acquired reference can be used "as is".
- The DBMS must throw the appropriate Exception if a client uses a Map reference outside a transaction.
- Map.Entry references that are created via the *DBMS* API **do not** remain valid between transactions. In other words, a Map.Entry reference acquired in tx_1 must be "re-acquired" if tx_2 code wants to use that DBMS object.
- The DBMS is not responsible for throwing an exception if a client uses a Map.Entry reference outside a transaction.
- tx_1 work performed on a stale Map.Entry reference will not be visible to tx_2

> The lecture motivates this (uncomfortable) distinction between Map and Map.Entry references.

7.3 Semantics of Null Keys or Values

Important: in *V1* and *V2*, although values are allowed to be null, **keys may not be null**.

7.4 Isolation

Your V1 implementation will provide *isolation* for your DBMS clients. Specifically, your implementation will permit *concurrent access* for any number of clients. Conceptually, you must ensure that changes made by tx_1 to its data are not visible to concurrently executing tx_2 (and *vice versa*). Your implementation **must use** a *lock-based* protocol (specifically 2PL) to prevent concurrent transactional access to the same database resource.

Also: your implementation must prevent deadlocks from occurring by using a *timeout-based* strategy. From an API perspective, your V1 DBMS must implement the *ConfigurableDBMS* interface (below). This is the only API change that is visible to your clients.

```
package edu.yu.oatsdb.base;
public interface ConfigurableDBMS extends DBMS {
  /** Sets the duration of the "transaction timeout
     ".
   * A client whose transaction's duration exceeds
   * the DBMS's timeout will be automatically rolled
   * back by the DBMS.
   *
   * @param ms the timeout duration in ms, must be
   * greater than 0
   */
  void setTxTimeoutInMillis(int ms);

  /** Returns the current DBMS transaction timeout
   * duration.
   *
   * @return duration in milliseconds
   */
  int getTxTimeoutInMillis();
}
```

The specified timeout duration is used by your DBMS as a trigger to rollback a transaction that has blocked on a database resource for "too long". However, the DBMS must also aggressively "wake up" a blocked transaction as soon as another transaction releases the lock on the resource on which this transaction is waiting.

Note: your strategy for providing *isolation* may well interact with your strategy for providing *atomicity*. Think about these issues before you start coding!

7.5 Rubric

Assuming that your code builds successfully:

- 10%: does your DBMS responds correctly to a set of basic API calls ("does it run")?
- 10%: does your DBMS run to completion on a *performance test* (this is independent of "how well" your code performs!). At the very least, performance should improve (up to a point) with a higher degree of concurrency!
- 15%: does your DBMS run to completion on a *stress test*
- 20%: does your DBMS correctly respond to *concurrent timing* tests?
- 45%: JUnit tests ("does it run correctly?")

8 V2: Persistence

I've allocated half of a lecture for a presentation of the *V2* milestone: be sure to have a look at those slides in addition to the material below.

As with the *V1* milestone, for the *V2* milestone, you will provide implementations of the following interfaces:

- *DBMS*, *Tx*, *TxMgr*
- Your implementations of the *DBMS* and *TxMgr* classes **must be** in the `edu.yu.oatsdb.v2` package. If your implementation references code in other packages, their use must be "transparent" to my method invocations.
- The *$DIR* is: `DatabaseImplementation/projects/oatsdb/V2`. Note that "V2" is capitalized, but the package name is not.

- Refer back to the *V0* for descriptions of how your classes **must** package their implementations of the implementations and how I will instantiate your classes. The only difference is the use the "V2" enum value instead of the "V2" value.

This milestone builds on the previous milestone's semantics. Previously, your DBMS provided *transactional atomicity* and *isolation* for a main-memory database. Once the **OATSdb** process exits, all state stored in the database vanishes! In this milestone, you will add *transactional persistence* to your database. This important change will be made with (almost) **no changes to the interface** ☺

Note: you will use the file system on your computer as the "persistent storage media".

8.1 Transactional Persistence

From an API perspective, your V1 DBMS must implement the *ConfigurablePersistentDBMS* interface (below). This is the only API change that is visible to your clients.

```
package edu.yu.oatsdb.base;

/** Extends the "Configurable DB Management System"
 * interface with an API that allows for
 * configuration of a persistent DBMS.
 *
 * Design note: In a production system these APIs
 * would be available only to administrators.

 * @author Avraham Leff
 */

public interface ConfigurablePersistentDBMS extends
    ConfigurableDBMS {

  /** Returns the disk usage in MB of this DBMS
   * instance.
   *
   * @return disk usage in MB for this DBMS
   */
  double getDiskUsageInMB();
```

```
    /** Delete all files and directories associated
     * with this DBMS instance from both disk and
     * from main-memory.  Effectively resets the
     * database.
     *
     * IMPORTANT: the effects of this API on existing
     * transactions is undefined.  This method should
     * be invoked only when the system is quiescent
     * (i.e., outside a transaction).
     */
    void clear();
}
```

The challenge for this milestone isn't "persistence" *per se*: it's providing *transactional* persistence. Work that is rolled back (whether by the client explicitly, or by the DBMS) must not persist. Conversely: work that is committed, must be successfully persisted such that no work is lost even if the system crashes in the next millisecond.

The approach discussed in lecture may not perform well, but is relatively straightforward to implement. You may choose to use other approaches (such as *log-based* recovery schemes: the API is very unconstrained and deliberately gives you complete implementation freedom.

8.2 Semantics of Null Keys or Values

Important: in *V1* and *V2*, although values are allowed to be null, **keys may not be null**.

8.3 Rubric

Assuming that your code builds successfully:

- 10%: does your DBMS responds correctly to a set of basic API calls ("does it run")?
- 10%: does your DBMS run to completion on a *performance test* (this is independent of "how well" your code performs!)
- 15%: does your DBMS run to completion on a *stress test*
- 20%: does your DBMS correctly respond to *concurrent timing* tests?

- 20%: does your DBMS correctly respond to *durability stress tests*: meaning, system crashes before and after transactional boundaries.
- 25%: JUnit tests ("does it run correctly?")

9 Tips, Suggestions, & Warnings

In the previous iteration of this project, I received some Piazza feedback to the effect that "we should be given an API" ...

Please note that you do have an API! For example:

- Implement (v0 version) of the OATSDBInterfaces project
- Implement *get, put, remove* Map APIs with v0 semantics

The issue being raised is really: "too much gray area" between the API and the implementation. I grant that the Javadoc doesn't fully specify what needs to be done. This is deliberate on my part: I want you to have to think through these issues so you can better appreciate what a DBMS is about.

I reached out for confirmation of this approach:

> "In the real world you don't get an API spec - you get verbal requirements that you, as a software engineer, need to translate into code."

I realize that some of you are not used to doing this extra work, but I strongly believe that you must get experience dealing with these ambiguities. Apparently COM 3563 has been elected to do the job ... This requirements document includes enough detail to guide your implementation to pass my test suite successfully.

> When in doubt, raise an issue on Piazza: I will clarify as appropriate.

Recommendation: Code Complete: A Practical Handbook of Software Construction, Second Edition