Transaction Patterns

A Collection of Four Transaction Related Patterns

By Mark Grand
mgrand@mindspring.com

This paper contains some transaction related patterns from my forthcoming book, Patterns in Java, Volume 3: Design Patterns for Enterprise and Distributed Applications.

A transaction is a sequence of operations that change the state of an object or collection of objects in a well defined way. Transactions are useful because they satisfy constraints about what the state of an object must be before, after or during a transaction. For example, a particular type of transaction may satisfy a constraint that an attribute of an object must be greater after the transaction than it was before the transaction. Sometimes, the constraints are unrelated to the objects that the transactions operate on. For example, a transaction may be required to take place in less than a certain amount of time.

The patterns in this chapter provide guidance in selecting and combining constraints for common types of transactions. Figure 1 shows how the patterns in this chapter build on each other.

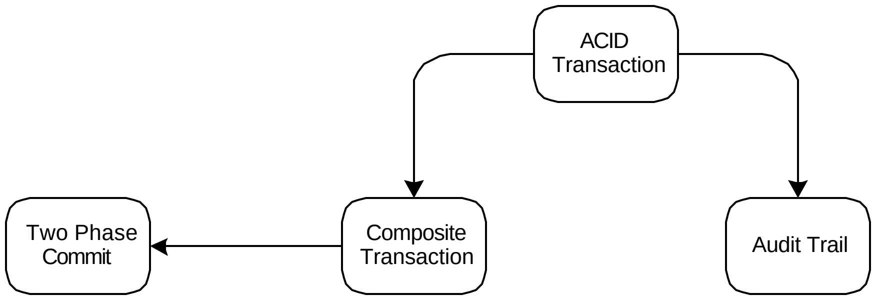


Figure 1: Pattern Map

The first and most fundamental pattern to read is the ACID Transaction pattern. It describes how to design transactions that never have inconsistent or unexpected outcomes. The Composite pattern describes how to compose a complex transaction from simpler transactions. The Two Phase commit pattern describes how to ensure that a composite transaction is atomic. The Audit Trail pattern describes how to maintain an historical of ACID transactions.

You may notice the lack of code examples in this paper. It is the author's opinion that the patterns in this paper are too high level for concrete code examples to be useful. The application of these transaction related patterns can be readily understood at the design level. However, there is generally no clear relationship between individual pieces of code and the fact that they are part of a transaction.

The topics discussed in this chapter are discussed in greater detail in [Gray-Reuter93] and [Date94].

ACID Transaction

Synopsis

Ensure that a transaction will never have any unexpected or inconsistent outcome. You do that by ensuring that the transaction has the ACID properties: atomicity, consistency, isolation and durability.

Context

Suppose that you are developing software for a barter exchange business. It works as a clearing house for indirect barter. For example, a hotel chain offers vouchers for a stay in one of their rooms in exchange for new mattresses. The clearing house matches up the hotel with a mattress manufacturer that spends lot of money on business trips. The clearing house also facilitates the exchange of the vouchers for the mattresses.

Every transaction that the system handles consists of an exchange of goods or services. Each transaction must update the system's model of who is supposed to exchange what goods and services with whom in exactly the expected way. The system must not lose any transactions. The system must ensure that two different clients don’t make a deal for the same goods or service.

Forces

- You want the result of a transaction to be well defined and predictable. Given only the initial state of the objects in a transaction and the operations to perform on the objects, you should be able to determine the final state of the objects. Even if the nature of the transaction is non-deterministic (i.e. a simulated roll of dice), given the initial state of the object involved, you should be able to enumerate the possible outcomes.

- Once started, a transaction may succeed. It is also possible for a transaction to fail or abort. There are many possible causes for a transaction to fail. Many causes may be unanticipated. Even if a transaction fails, you don’t want it to leave objects in an unpredictable state that compromises their integrity.

- A transaction can fail at any point. If there is a failure at a random point in the transaction, additional operations may be required to bring the objects involved into a predictable state. Determining what operations to perform may be difficult or impossible if a failed transaction can put objects in a state that violates their integrity.

- You want the outcome of transactions to depend only on the initial state of the objects a transaction acts on and the operations in the transaction. The possibility of concurrent transactions should not affect the result of the transactions.

- If the results of a transaction are stored in volatile memory, then the observed results of the transaction are less predictable. The contents of volatile memory can change unpredictably. More generally, the observed results of a transaction are unpredictable if they do not persist as long as objects that are interested in the results or until another transaction changes the state of the effected objects.

- Satisfying all of the forces listed in this section may add complexity and overhead to the implementation of a transaction. The nature of some transactions ensures a satisfactory outcome without having to address all of the forces. For example, if there will be no concurrent transactions, then there is no need to address the possibility that concurrent transaction could interfere with each other. Also, if the outcome is the same whether a transaction is performed once or more than once, then recovery from catastrophic failure can be simplified.

  It is not possible take such shortcuts in the design of most transactions. If a transaction does not lend itself to the use of these shortcuts, then there is a good justification for the complexity and overhead associated with satisfying all of the forces listed in this section.

Solution

You can ensure that a transaction has a predictable results by ensuring that it has the ACID properties:

Atomic — The changes a transaction makes to the state of objects it operates on are atomic: Either all of the changes happen exactly once or none of the changes happen. These changes include both internal state changes and database changes, transmission of messages and visible side effects on other programs.

Consistent — A transaction is a correct transformation of an object’s state. At the beginning of a transaction, the objects a transaction operates on are consistent with their integrity constraints. At the end of the transaction, whether it succeeds or fails, the objects are again in a state consistent with their integrity constraints. If all or the transaction’s preconditions are met before the transaction begins, all of its postconditions are met after the transaction successfully completes.

Isolated — Even though transactions execute concurrently, it appears to each transaction, T, that other transactions execute either before T or after T but not both. That means that if an object that is involved in a transaction fetches an attribute of an object, it does not matter when it does so. Throughout the lifetime of a transaction, the objects that the transaction operates on will not notice any changes in the state of other objects as the result of concurrent transactions.

Durable — Once a transaction completes successfully (commits), the changes it has made to the state of the object(s) become relatively persistent. They will persist at least as long as any object that can observe the changes.

Because most database managers ensure that transactions performed under their control have these properties, many people only think of ACID properties in connection with database transaction. It is important to understand that the ACID properties are valuable for all kinds of transactions.

Consequences

- Use of the ACID transaction pattern makes the outcome of transactions predicable.

- Use of the ACID transaction pattern can substantially increase that amount of storage required by a transaction. The additional storage requirement arises from a need to store the initial state of every object involved in a transaction, so that if the transaction fails it is possible to restore the initial state of the objects it acted on. Maintaining the isolation of a transaction may require the copying of objects. The purpose of the copies is to allow the original object to be modified while an unchanging copy of the original is visible to other transactions.

- Often, ACID transactions are implemented by having transaction logic manipulate objects indirectly through a mechanism that automatically enforces the ACID properties for transactions, such as a database manager. In those cases where the logic for a transaction is also responsible for maintaining the transaction's ACID properties, the complexity of the transaction logic may be greatly increased. The use of an intermediary that takes responsibility for enforcing the ACID properties makes it much easier to correctly implement the logic that drives a transaction.

Implementation

The simplest way to ensure the ACID properties of a transaction is for the transaction logic to manipulate the state of objects through a tool such as a database manager that automatically enforces the ACID properties. If a program works with transactions that will not involve persistent data or objects, the performance penalty introduced by a database manager that persists data may be undesirable. Such applications may be able to take advantage of in-memory databases.

Sometimes it is not possible to use any tool to enforce ACID properties. The common reasons for that are performance requirements and a need to keep the size of an embedded application small. Adding logic to an application to enforce the ACID properties for its transaction can introduce a lot of complexity into a design.

Here are strategies for explicitly supporting each of the ACID properties:

Atomicity

The primary issues to address when providing support for atomic transactions is that there must be a way to restore objects to their initial state if a transaction that ends in failure.

The simplest way to be able to restore an object's state after a failed transaction is to save the object's initial state in a way that it can easily be restored. The Snapshot pattern (discussed in [Grand98A]) provides guidance for this strategy. Figure 2 is a class diagram that shows this general approach.

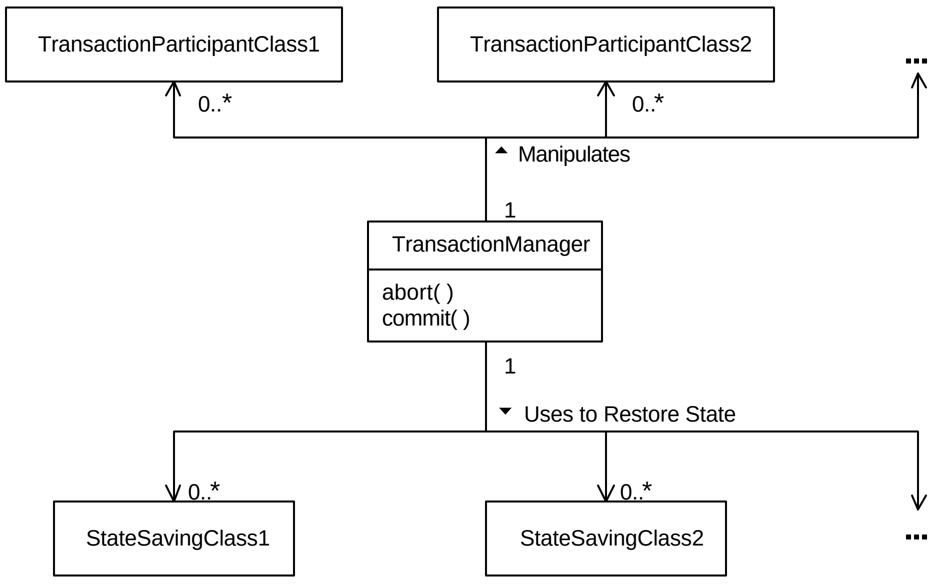


Figure 2: Saving State for Future Recovery

An object in the role of transaction manager manipulates instances of other classes that participate in a transaction. Before doing something that will change the state of an object that it manipulates, the transaction manager will use an instance of another class to save the initial state of the object. If the transaction manager's `commit` method is called to signal the successful completion of a transaction, then the objects that encapsulate the saved states are discarded. However, if the transaction manager detects a transaction failure, either from a call to its `abort` method or the abnormal termination of the transaction logic, then it restores the objects that participate in to their initial state.

If it is not necessary to save an object's state beyond the end of the current program execution, the simplest way to save the objects state is to clone it. You can make a shallow copy[1] of an object by calling its `clone` method.

All classes inherit a `clone` method from the `Object` class. The `clone` method returns a shallow copy of an object if its class gives permission for its instances to be cloned by implementing the `Cloneable` interface. The `Cloneable` interface is a marker interface (See the Marker Interface pattern in [Grans98A]). It does not declare and methods or variables. Its only purpose is to indicate that a class's instances may be cloned.

In order to restore the state of an object from an old copy of itself, the object must have a method for that purpose. The following listing shows an example of a class whose instances can be cloned and then restored from the copy.[2]

```
class Line implements Cloneable {
    private double startX, startY;
    private double endX, endY;
    private Color myColor;
    ...
    public void restore(Line ln) {
        startX = ln.startX;
        startY = ln.startY;
        endX = ln.endX;
        endY = ln.endY;
        myColor = ln.myColor;
    } // restore(Line)
```

[1] A shallow copy of a object is another object whose instance variables have the same values as the original object. It refers to the same objects as the original object. The other objects that it refers to are not copied.
[2] At the beginning of this chapter, I stated that there would be no code examples. The code examples that appear here are examples are implementation examples and not examples of the pattern itself.

```
} // class Line
```

For saving the state of an object whose state is needed indefinitely, a simple technique is to use Java’s serialization facility. Here is how to use it:

Java’s serialization facility can save and restore the entire state of an object if its class gives permission for its instances to be serialized. Classes give permission for their instances to be serialized by implementing the interface `java.io.Serializable` like this:

```
Import Java.io.Serializable;
...
class Foo implements Serializable {
```

The `Serializable` interface is a marker interface. It does not declare any variables or methods. Declaring that a class implements the `Serializable` interface simply indicates that instances of the class may be serialized.

To save the state objects by serialization, you need an `ObjectOutputStream` object. You can use an `ObjectOutputStream` object to write a stream of bytes that contains an object’s current state to a file or a byte array.

To create an `ObjectOutputStream` object that serializes that state of objects and writes the stream of bytes to a file, you would write code that looks like this:

```
FileOutputStream fout = new FileOutputStream("filename.ser");
ObjectOutputStream obOut = new ObjectOutputStream(fout);
```

The code creates an `OutputStream` to write a stream of bytes to a file. It then creates an `ObjectOutputStream` that will use the `OutputStream` to write a stream of bytes.

Once you have created an `OutputStream` object, you can serialize objects by passing them to the `OutputStream` object’s `writeObject` method, like this:

```
ObOut.writeObject(foo);
```

The `writeObject` method discovers the instance variables of an object passed to it and accesses them. It writes the values of instance variables declared with a primitive type such as `int` or `double` directly to the byte stream. If the value of an instance variable is an object reference, the `writeObject` method recursively serializes the referenced object.

Creating an object from the contents of a serialized byte stream is called *deserialization*. To deserialize a byte stream, you need an `ObjectInputStream` object. You can use an `ObjectInputStream` object to reconstruct an object or restore an object’s state from the state information stored in a serialized byte stream.

To create an `ObjectInputStream` object, you can write some code that looks like this:

```
FileInputStream fin = new FileInputSteam("filename.ser");
ObjectInputStream obIn = new ObjectInputStream(fin);
```

This code creates an `InputStream` to read a stream of bytes from a file. It then creates an `ObjectInputStream` object. You can use the `ObjectInputStream` object to create objects with instance information from the stream of bytes or restore an existing object to contain the instance information. You can get an `ObjectInputStream` object to do these things by calling its `readObject` method like this:

```
Foo myFoo = (Foo)obIn.readObject();
```

The `readObject` method returns a new object whose state comes from the instance information in the byte stream. That is not quite what you need when restoring an object to its initial state. What you need is a way to use the instance information to set the state of an existing variable. You can arrange for that as you would for allowing objects’ state to be restored from a clone. You ensure that the class of the objects to be restored has a method that allows instances of the class to copy their state from another instance of the class.

Saving the initial state of object that a transaction manipulates is not always the best technique for allowing the initial state of a objects manipulated by a transaction to be restored to their initial stated. If transactions perform multiple operations on objects that contain a lot of state information and the transaction only modifies some of the state information in the objects, saving the object's entire state information is wasteful.

An implementation approach that produces more efficient results in this situation is based on the Decorator pattern described in [Grand98A]. The technique is to leave the original object's state unmodified until the end of the transaction and use wrapper objects to contain the new values. If the transaction is successful, the new values are copied into the original object and the wrapper objects are then discarded. If the transaction ends in failure, then the wrapper objects are simply discarded. Figure 3 is a class diagram that shows this sort of design.

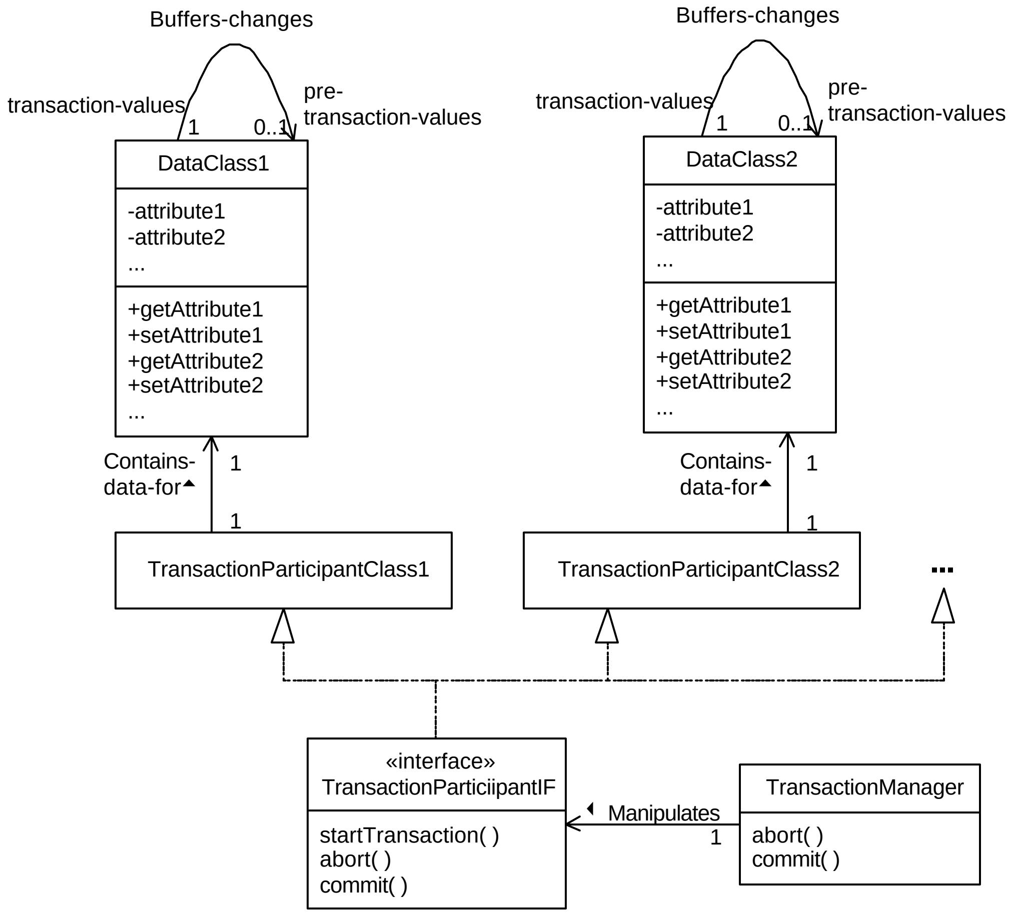


Figure 3 Atomicity through Wrapper Objects

In this design, the objects that the transaction manipulates do not contain their own instance data. Instead, a separate object contains their instance data. To ensure strong encapsulation, the class of the objects that contain instance data should be an inner class of the class of the manipulated objects.

When a transaction manager becomes aware than an object will be involved in a transaction, it calls the object's `startTransaction` method. The `startTransaction` method causes the object to create and use a new data object. When the manipulated object calls one of the new data object's methods to fetch the value of an attribute, if the data object does not yet have a value for that attribute, it calls the corresponding method of the original data object to get the value.

If a transaction ends in failure, then the transaction manager object calls the `abort` method of each of the manipulated object's. Each object's abort method causes it to discard the new data object and any values that it may contain.

If a transaction ends in success, then the transaction manager object calls the `commit` method of each of the manipulated object's. Each object's `commit` method causes the new data object to merge its data values into the original data object. It then discards the data object.

This design only requires data values to be copied if they are altered by a transaction. It may be more efficient than saving an object's entire state if the object contains a lot of state information that is not involved in the transaction. The disadvantage of this design is that it is more complex.

Consistency

There are no implementation techniques specifically related to consistency. All implementation techniques that help to ensure the correctness of programs also help to ensure consistency.

The most important thing that you should do ensure the consistency of transaction is testing. The Unit Testing and System Testing patterns described in [Grand98B] are useful in designing appropriate tests.

Isolation

Isolation is an issue when an object may be involved in concurrent transactions and some of the transactions will change the state of the object. There are a few different possible implementation techniques for enforcing isolation. The nature of the transactions determines the most appropriate implementation technique.

If all of the transactions will modify the state of an object, then you must ensure that the transactions do not access the object concurrently. The only way to guarantee isolation is to ensure that they access the object's state one at a time by making the methods that modify the object's state synchronized. This technique is described in more detail by the Single Threaded Execution pattern described in [Grand98A].

If some of the concurrent transactions modify an object's state and others use the object but do not modify its state, you can improve on the performance of single threaded execution. You can allow transactions that do not modify the object's state to access the object concurrently, while only allowing transactions that modify the object's state to access it in a single threaded manner. This technique is described in more detail by the Read/Write Lock pattern described in [Grand98A].

If transactions that are relatively long lived. It may be possible to further improve the performance of some transactions that use but not modify the state of the object if it is not necessary for the objects to have a distinct object identity. You can accomplish this by arranging for transactions that use an object but do not modify the object's state to use a copy of the object. Some patterns that can be helpful in doing this are

- The Return New Objects from Accessor Method pattern (described in [Grand98B]).
- The Copy Mutable Parameters pattern (described in [Grand98B])
- The Copy on Write Proxy pattern, which is used as an example in the description of the Proxy pattern in [Grand98A].

Durability

The basic consideration for ensuring the durability of a transaction is that its results must persist as long as there may be other objects that are concerned with the object's state. If the results of a transaction are not needed beyond a single execution of a program, it is usually sufficient to store the result of the transaction in the same memory as the objects that use those results.

If other objects may use the results of a transaction indefinitely, then the results should be stored on a non-volatile medium such as a magnetic disk. This can be trickier than it seems at first. The writing of transaction results to a disk file must appear atomic to other threads and programs. There are a few issues to deal with in ensuring this:

- A single write operation may be translated into multiple write operations by the object responsible for the write operation or the underlying operating system. That means that data written using a single write call may not appear in a file all at once.

- Operating systems may cache write operations for a variety of efficiency reasons. That means data written by multiple write operations may appear in a file at the same time or may be written in a different sequence than the original write operations.

- When accessing remote files, additional timing issues arise. When a program writes information to a local file, the modified portion of the file may reside in the operating system's cache for some time before it is actually written to the disk. If another program tries to read the modified portion of a file while the modifications are still cached, most operating systems will be smart enough to create the illusion that the file has already been modified. If read operations on a file reflect write operations as soon as the occur, the system is said to have *read/write consistency*.

  Read/Write consistency is more difficult to achieve when accessing a remote file. That is partially because there can be unbounded delays between the time that a program performs a write operation and the time that the write arrives at the remote disk. If you take no measures to ensure that access to a remote file has read/write consistency, the following sequence of event is possible:

  1. Program X performs a write operation.

  2. Program Y reads the unmodified but out of date file.

  3. Program X's write arrives at the file.

- An object that may read the same data from a file multiple times will pay a performance penalty if it does not cache the data to avoid unnecessary read operations. When reading from a remote file, caching becomes more important, because of the greater time required for read operations. However, caching introduces another problem.

  If the data in a file is modified, then any cache that contains data read from the file is no longer consistent with the file. This is called the *cache consistency* problem.

The following paragraphs contain some suggestions on how to deal with the problems related to the timing of actual writes to local files. The Read/Write Consistency pattern explains how to handle the read/write consistency problem. The Cache Consistency pattern explains how to handle cache consistency.

It is not generally possible to control exactly when the data from a write operation will actually be written to physical file. However, it is possible to force pending write operations to local file systems to complete. This guarantees that all pending write operations have completed at a known point in time. It is generally good enough for ensuring the durability of a transaction unless the transaction is subject to real time constraints.

There are two steps to forcing write operations to local file systems to complete. The first step is to tell objects your program is using to perform write operations to flush their internal buffers. For example, all subclasses of `OutputStream` inherit a method named `flush`. A call to the flush method forces the `OutputStream` object to flush any internal buffers that it might have.

The second step to forcing write operation to local file systems to complete is to get the `FileDescriptor` object for the file your are writing. `FileDescriptor` objects have a method named `sync`. A call to a `FileDescriptor` object's `sync` method tells the operating system to flush any cached write operations for the associated file.

All

There is another implementation issue that affects all four ACID properties. The issue is how to handle a commit operation that is unable to successfully complete. In all cases, the objects manipulated by the transaction must be left in a consistent state that either reflects that success or failure of the transaction.

There are two failure modes that we are concerned about. Ones is that the commit operation is unable to commit the changes made during the transaction, but the objects that are interested in the results of the transaction are alive and well. The other is a larger scale failure that causes the commit operation not to complete and also causes all of the objects that are interested in the results of the transaction to die.

The problem is simplest when the failure is limited to the commit operation and the objects interested in the results of the transaction are still alive and well. In this case, since the commit could not succeed, the transaction must fail. All that is required is to restore the objects manipulated by the transaction to their state at the beginning of the transaction.

The larger scale failure presents an additional challenge if the objects that were interested in the results of the transaction will persist after the failure. Before processes or threads are started that will allows objects to see an incomplete transaction, the incomplete transaction must be detected and its commit must be completed or backed out.

In summary, adding your own logic to an application to enforce ACID properties for transactions adds considerable complexity to the application. When possible, use an available tool than can manage the ACID properties for you.

If you must create your own support for the ACID properties of transactions, your design for each transaction will include some of the elements shown in the class diagram in Figure 4.

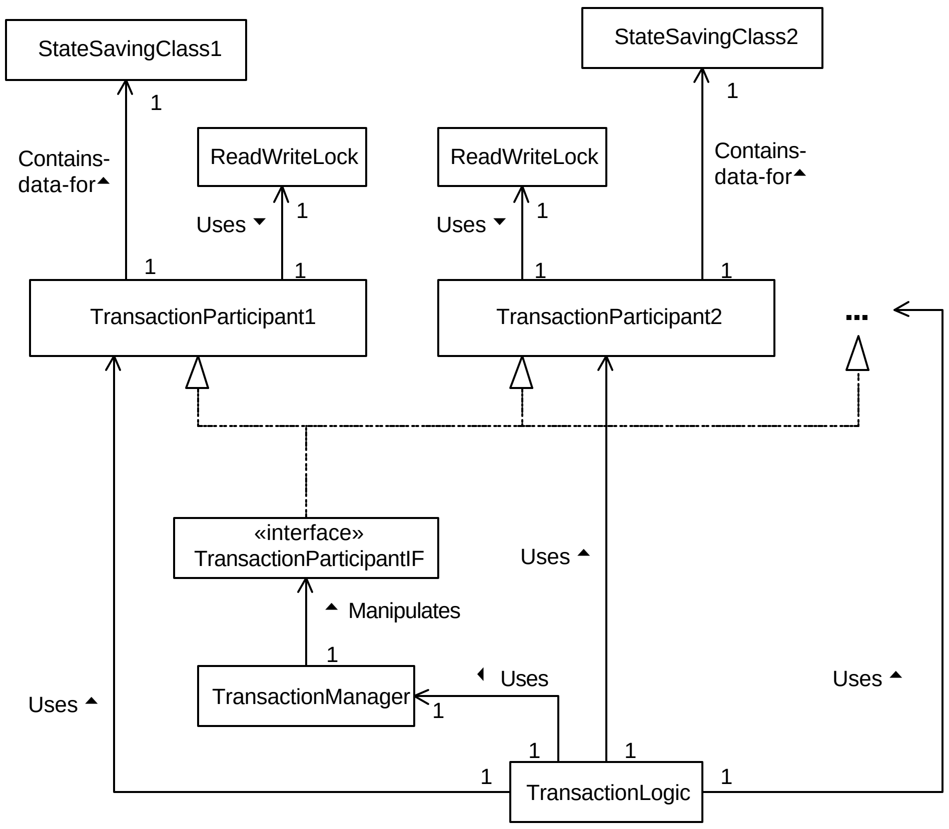


Figure 4: Generic Transaction Classes

Here are descriptions of the roles classes play in ACID transactions as indicated in Figure 4:

TransactionLogic

Though there are many ways to organize the logic of a transaction, the most common design is to have one class that encapsulates the core logic of a transaction. This class may encapsulate the core logic for multiple related transactions.

TransactionParticipant1, **TransactionParticipant2**, …

The logic encapsulated in a `TransactionLogic` class modifies the state of instances of these classes.

TransactionManager

This class encapsulates reusable common logic to support atomicity. For distributed transactions, it may also encapsulate the logic to support durability. `TransactionLogic` objects use an instance of this class to manage a transaction.

TransactionParticipantIF

Each `TransactionParticipant` class implements this interface. The purpose of this interface is to allow a `TransactionManager` object to manipulate `TransactionParticipant` objects without having a dependency on any specific `TransactionParticipant` class.

StateSavingClass1, **StateSavingClass2**, …

Classes in this role are responsible for saving and restoring the state of `TransactionParticipant` objects. These classes are usually specific to a single `TransactionParticipant` class or a small number of related `TransactionParticipant` classes.

ReadWriteLock

If there will be concurrent transactions accessing `TransactionParticipant` objects, with some transactions modifying an object and other transactions just requiring read access, an instance this class is used to coordinate shared read access and exclusive write access to the object. These classes are usually reusable.

Known Uses

Database management systems guarantee ACID properties for transactions. Some use an implementation of atomicity based on keeping a copy of the initial state of each item involved in a transaction. For example, Interbase keeps the original and the modified version of every record involved in a transaction until the transaction completes. When the transaction completes, it discards one or the other, depending on whether the transaction succeeds or fails.

Oracle uses an implementation of atomicity that is analogous to the implementation using wrapper objects.

Related Patterns

Snapshot

The Snapshot pattern (described in [Grand98A]) describes techniques for saving and restoring the state of objects. This is the better way to recover from a transaction failure when a transaction involves a long sequence of operations that modify the state of a small number of simple objects.

Command

The Command Pattern (described in [Grand98A]) describes techniques for remembering and undoing a sequence of operations. This is the better way to recover from a transaction failure when a transaction involves a short sequence of operations that modify the state of a large number of complex objects.

Audit Trail

Logging a sequence of operations to support the Command Pattern is structurally similar to maintaining an audit trail.

System Testing

The System Testing pattern (described in [Grand98B]) should be used to ensure the consistency of transactions.

Unit Testing

The Unit Testing pattern (described in [Grand98B]) may also help to ensure the consistency of transactions.

Single Threaded Execution

The Single Threaded Execution Pattern (described in [Grand98A]) can be used to keep transactions that modify the state of the same object isolated from each other.

Read/Write Lock

The Read/Write Lock pattern (described in [Grand98A]) can be used to keep transactions that use the same object isolated from each other while allowing transactions that do not modify the object’s state to execute concurrently.

Read/Write Consistency

If you directly manage the storage of persistent distributed objects, you may need the Read/Write Consistency pattern to ensure that data and objects that are read from files are consistent with the most recent write operation.

Cache Consistency

If you directly manage the storage of persistent distributed objects, you may need the Cache Consistency pattern to ensure that the result of a locally initiated read operation matches the current contents of a remote store.

Composite Transaction

Synopsis

You want to design and implement transactions correctly and with a minimum of effort. Simple transactions are easier to implement and make correct than complex transactions. Design and implement complex transactions from simpler ACID transactions.

Context

Sometimes, you want to design a complex ACID transaction using existing ACID transactions as building blocks. Using ACID existing transactions to build a more complex transaction does not automatically give it the ACID properties. Consider the following situation.

You work for the IT department of a supermarket chain. In addition to having a number of stores to sell food, the company has a central facility where it produces bread, cakes and other baked goods for the stores. The IT Department provides systems to support these activities:

- There is manufacturing software for the bakery. Every evening it is fed the quantities of each item that each store will need for the following day. It produces reports telling the bakers how much of each item to produce and what ingredients need to order for following days.

- There is transportation scheduling software. Every evening it is also fed the quantities of each item each store will need for the following day. It schedules trucks to transport baked goods to the stores. It produces reports telling the bakers how much of each item to put in each truck.

Currently, the amount of each product each store needs for the next day must be keyboarded into both software applications. This increases labor costs. It makes data entry errors more likely, since there are twice as many opportunities to make mistakes. The costs of data entry errors are higher because they can lead to baked goods being produced but not loaded onto a truck or too many trucks being scheduled.

You have the task of creating a mechanism that allows the data to be entered only once. You think of writing a data entry program that will put the data in the appropriate database table of each application. Though you know that you can make it work, you search for another way. Because the program would assume the internal structure of other applications, you are concerned about maintenance problems later on.

Reading each application's documentation, you find that they both have an API to programmatically present data to each application. Transactions initiated by the APIs have the ACID properties. This gives you a way to build the data entry mechanism using only supported features of the applications.

The fact that both APIs support the ACID properties greatly simplifies the task of building a composite data entry transaction with a predictable outcome. By creating a composite transaction that simply invokes each API, you get a transaction that is consistent and durable without doing anything else. However, you must carefully consider how to ensure that the composite transaction is atomic and isolated. They will generally not be atomic or isolated. You must either take additional steps to make them so or determine that a less stringent guarantee of their behavior is sufficient. Without proper attention to these details transactions can be lost, be applied multiple times or concurrent transactions may corrupt each other.

The composite transaction in the example is not automatically atomic. That is not a problem, for two reasons.

- Before the transaction runs, the quantity of all baked good scheduled to be produced for a store is zero. For that reason, there is no need to save the old values before the transaction. You can back out the transaction by setting all of the values to zero.

- Both the component transactions are *idempotent*. Idempotent means that a transaction can happen once or more than one and still have the same outcome. This simplifies the task of recovery from a crash because the only information that needs to be saved is the fact that the transaction was begun. It is not necessary to be certain that it completed.

The other area you will need to address is isolation. Though each component transaction has the isolation property, this sequence of events is possible:

Composite Transaction 1	Composite Transaction 2
Manufacturing Transaction 1	
	Manufacturing Transaction 2
	Transportation Transaction 2
Transportation Transaction 1	

If the composite transaction is isolated from other transactions, then neither transaction should be able to observe state changes made by the other. This is not the case in the above scenario. In this sequence of events, the first half of transaction 1 sees things as they were before transaction 2; the second half of transaction 1 sees things as they are after transaction2. If you only need to isolate these transactions from each other, you can solve the crash recovery problem and the isolation problem the same way:

Before the composite transaction invokes any of its component transactions, it can store the transaction data in a file. When the transaction is done, it deletes the file. If there is a crash that prevents the completion of the transaction, then when the program restarts it can detect the existence of the file and restart the transaction.

The existence of the file can also be used to isolate transactions. Using the Lock File pattern, if the file exists when a composite transaction starts, it waits until the file no longer exists before it continues.

Figure 4.5 is a class diagram that shows your design.

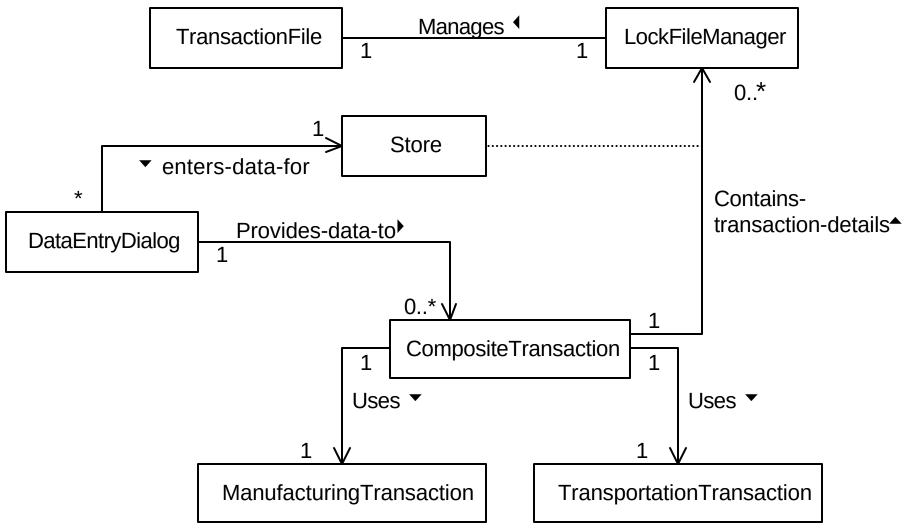


Figure 5: Composite Data Entry Transaction

Figure 5 adds a detail not previously discussed. Instead of using just one transaction file, it uses one transaction file per store. This is based on an assumption that each store only enters data for itself and no other stores. This means concurrent transactions from different stores are isolated from each other simply because they are from different stores. You only need the file to isolate concurrent transactions from the same store. Forcing transactions for one store to wait for transactions for another store to complete introduces an unnecessary delay.

In this example, it was possible to find a solution that did not require that the composite transaction was atomic and isolated. This is the exception rather than the rule. In most cases, it is necessary to take measures ensure that a composite transaction as all of the ACID properties.

Forces

- Building complex transactions with predictable outcomes from simpler transactions is greatly facilitated if the simpler transactions have the ACID properties.

- If the ACID properties of a set of transactions are implemented using a single mechanism that supports nested transactions, then implementing the ACID properties for a composite transaction composed of those transactions is very easy.

- If the ACID properties of a set of component transactions are implemented using a mechanism that does not support nested transactions, then implementing ACID properties for a composite transaction is more difficult. Implementing a composite transaction using component transactions whose ACID properties are implemented using incompatible mechanisms that do not work with each other is also difficult. In some cases, it is impossible.

- It is more difficult for a maintenance programmer having to maintain a composite transaction to understand the full inner workings of a composite transaction, especially if there are multiple levels of composition.

Solution

Design classes that implement complex transactions so that they delegate as much work as possible to classes that implement simpler transactions. When selecting classes that implement transactions for incorporation into more complex transactions, you should with use classes that already exist and are known to be correct or you should select classes that will have multiple uses.

The simpler transactions should have the ACID properties. That greatly simplifies the task of ensuring predicable properties for the composite transaction.

Carefully choose the granularity of the simpler transactions. When designing with existing transactions, you generally have to work with the transactions as the exist. If you are designing the simpler transactions along with the complex, the granularity of the simpler transaction should be a balance between the need to keep the simpler transactions simple and the need to keep the more complex transactions understandable.

Sometimes, circumstances make ensuring the ACID properties of a composite transaction complicated. Figure 6 shows the structure of a composite transaction design when there are no such circumstances that make it more complicated.

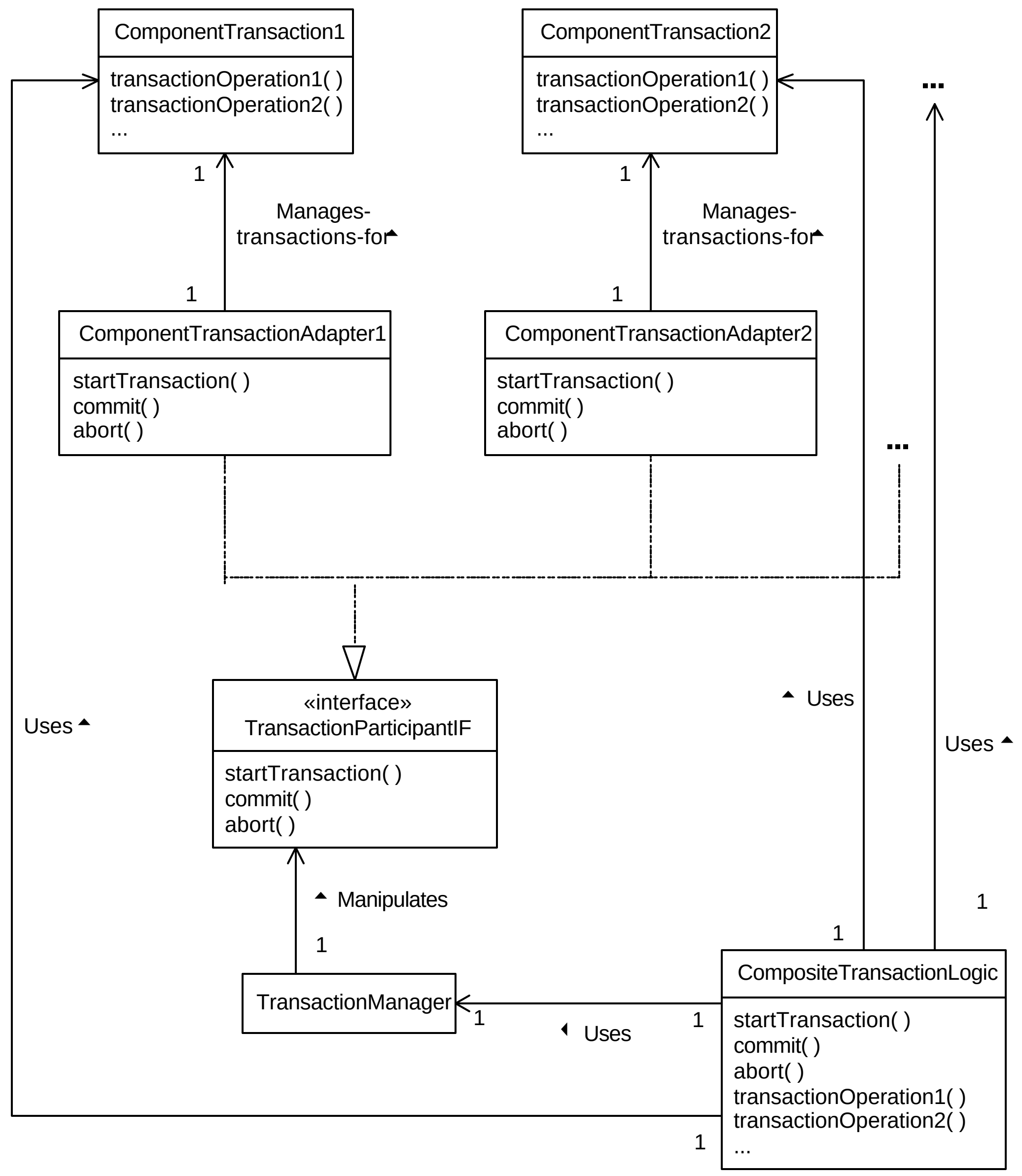


Figure 6: Composite Transaction Pattern

The classes shown in figure 6 play the following roles in the Composite Transaction pattern:

CompositeTransactionLogic Although there are many ways to organize the logic of a transaction, the most common design is to have one class that encapsulates the core logic of a transaction. This class can encapsulate the core logic for multiple related transactions.

ComponentTransaction1, ComponentTransaction2, … Classes in this role encapsulate a component transaction that is part of the composite transaction. `CompositeTransactionLogic` classes delegate transaction operations directly to `ComponentTransaction` objects. However, transaction management operations that begin or end a transaction are delegated indirectly through a `TransactionManager` class.

TransactionManager This class encapsulates reusable common logic to support atomicity and isolation. For distributed transactions, it may also encapsulate the logic to support durability. `CompositeTransactionLogic` objects use an instance of this class to manage a transaction.

In order to be independent of the classes that it manages within a transaction, it interacts with these classes through a `TransactionParticipant` interface.

TransactionParticipantIF `TransactionManager` classes interact with `ComponentTransaction` classes through an interface in this role.

ComponentTransactionAdapter Unless `ComponentTransaction` classes are specifically designed to work with the `TransactionManager` class being used, they don't implement the `TransactionParticipantIF` interface that the `TransactionManager` class requires. Classes in this role are adapters that implement the `TransactionParticipantIF` interface with logic that delegate to a `ComponentTransaction` class and supplement its logic in whatever way is necessary.

There are two areas in which applications of this pattern most often vary from the organization shown in Figure 5. Both areas of variability usually add complexity.

The first area of variability is that some portions of the composite transaction's logic may not already be encapsulated as a self-contained transaction. In many cases, such logic is too specialized for you to have an expectation of reusing it. It may not be possible to justify encapsulating such specialized logic in this way. In these situations, the design usually looks like a hybrid of figures 5 and 3, with some portions of the logic encapsulated in self-contained transactions and the unencapsulated portions having the additional details shown in figure 3.

The other area of variation is managing the predictability of the composite transaction's outcome. The preferred strategy for doing that is to ensure that the composite transaction has the ACID properties. Extensive experience has shown this is to be a successful strategy. Though using component transactions that have the ACID properties may simplify the task of ensuring that the composite transaction has the ACID properties, it is not sufficient.

The simplest situation for ensuring the ACID properties of the composite transaction is when there is a single mechanism for ensuring the ACID properties of all of the component transactions and the mechanism supports nested transactions. Such a mechanism does not only allow individual component transactions to abort themselves. It also allows the composite transaction to abort and restore all objects modified by committed component transactions to the state they had at the beginning of the composite transaction.

The simplest possibility is that you are using a tool to manage transactions and the tool supports nested transactions. Alternatively, if you control the implementation of all of the component transaction classes that you are using, then it is relatively easy to modify the techniques described by the ACID Transaction pattern to support nested transactions.

If the component transactions are managed by a mechanism that does not support nested transactions then you will need a different way to ensure the predictable outcomes of the composite transactions. If the component transactions are managed by different mechanisms, as is the case in the example under the "Context" heading, it is also necessary to find a different way to ensure the predictability of the outcome of the composite transaction.

The Two Phase Commit pattern describes a way to combine component transactions that have the ACID properties and are managed by different mechanisms into one composite transaction that has the ACID properties. However, you may not be able to use the Two Phase Commit pattern if all of the classes that encapsulate the component transactions have not been designed to participate in the Two Phase Commit pattern.

In some cases, it may be impractical or even impossible to ensure the ACID properties for the composite transaction. You will find descriptions of common alternatives and how to implement them under the "Implementation" heading.

Consequences

- Writing classes that perform complex transactions by having them delegate to classes that perform simpler transactions is a good form of reuse, especially when the classes that implement the simpler transactions already exist or will have multiple uses.

- The core logic of a transaction implemented as a composite transaction is less likely to contain bugs than a monolithic implementation of the same transaction. That is because the component transactions you build on are usually already debugged. Because implementing transactions in this way simplifies the core logic of the transaction, there are fewer opportunities to introduce bugs into it.

- If you are not able to use nested transactions or the Two Phase Commit pattern to manage the ACID properties of a composite transaction, it may be difficult to implement the ACID properties for the composite transaction. It may even be impossible to implement the ACID properties for the composite transaction. In such situations, you are forced to compromise on the guarantees you can make about the predictability of the transaction's outcomes.

- If there are no dependencies between component transactions, then it is possible for them to execute concurrently.

Implementation

There are a number of lesser guarantees that you may try to implement when it is not possible to enforce the ACID properties for a composite transaction. Some of the more common ones are discussed in this section

When it is not possible to ensure that a transaction is atomic, it may be possible to ensure that it is idempotent. If you are rely on idempotence rather than atomicity, then you must be able to ensure that a transaction will be completed at least once, after it is begun.

In some situations, it is possible to ignore the issue of isolation. If the nature of the transaction ensures that there will be no concurrent transactions that modify the same objects, then you do not need to anything to ensure that the transactions execute in isolation.

Known Uses

Sybase RDBMS and SQL Server support nested transactions and facilitate the construction of composite transactions.

JAVA API Usage

The Java Transaction API has facilities to aid in the construction of composite transactions.

Related Patterns

ACID Transaction

The Composite Transaction pattern is built on the ACID transaction pattern.

Adapter

The Composite pattern uses the Adapter pattern, which is described in Volume 1.

Command

The Command Pattern (described in [Grand98A]) can be the basis for an undo mechanism used to undo operations and restore objects to the state they were in at the beginning of a transaction.

Composed Method

The Composed Method pattern (described in [Grand98B]) is a coding pattern that describes a way of composing methods from other methods that is structurally similar to the way the Composite Transaction pattern composes transactions.

Lock File

The Lock File pattern can be used to enforce the isolation property for a composite transaction.

Two Phase Commit

The Two Phase Commit pattern can be used to ensure the ACID properties of a composite transaction composed from simpler ACID transactions.

Mailbox

When there is a need to ensure the reliability a composite transaction, you will want to take steps to ensure the reliability of the component transactions that constitute it. If the composite transaction is distributed, you will also want to ensure the reliable transmission of messages between the objects that participate in the transaction by such means as the Mailbox pattern.

Two Phase Commit

This pattern is based on material that appears in [Gray-Reuter93].

Synopsis

If a transaction is composed of simpler transactions, you want them to either all complete successfully or to all abort, leaving all objects as they were before the transactions. You achieve this by making an object responsible for coordinating the transactions so that they all complete successfully or all abort.

Context

Suppose that you have developed software for a barter exchange business. The software is responsible for managing barter exchanges. It records offers of exchange, acceptances and the consummation of each exchange.

The business has grown to the point where it has offices in a number of cities, each office facilitating barter exchanges between people local to its city. The business's management has decided that it is time to move take the business to the next level and allow barter between people in different parts of the country. They want someone in one city to be able to swap theater tickets for balloon rides near a different city. Currently that is not possible.

Each office runs its own computer that manages transactions for its clients. The offices run independently of each other. In order to support exchanges between clients of different offices, if must be possible to execute ACID transactions that are distributed between multiple offices.

To make that happen, there must be a mechanism that coordinates the portion of each transaction that executes in each office. It must be the case that every portion of each transaction successfully commits or every portion of each transaction aborts. It must never happen that one office thinks that a transaction completed successfully and another thinks that it aborted.

Forces

- Otherwise independent atomic transactions must participate in a composite atomic transaction.

- If any one of the component transactions participating in a composite atomic transaction fail, they must all fail. This implies that the component transactions are coordinated in some way.

- Though it is possible to distribute the responsibility for coordinating transactions over multiple objects, it is an unusual design decision. Distributing coordination of self-contained transactions adds complexity. It is an area that is not as well understood as designs that make a single object responsible for the coordination. Distributed coordination of transactions is still a valid research topic.

- There is very extensive industry experience with designs that make a single object responsible for coordinating transactions.

- The results of a transaction should persist as long as any objects may be interested in the results or until another transactions changes the state of the effected objects. If the transactions being coordinated have the ACID properties, then their durability attribute implies that this will be true for the results of each of the coordinated transactions individually.

- The responsibility for coordinating component transactions persists until the composite transaction has completed. However, the object(s) responsible for coordinating a transaction may experience a catastrophic failure during a transaction.

Solution

Make a single object responsible for coordinate otherwise independent ACID transactions participating in a composite transaction so that the composite transaction has the ACID properties. The object responsible for the coordination is called the coordinator. The coordinator coordinates the completion of the composite transaction in two phases. First, it determines if each component transaction has completed it work successfully or not. If any of the component transactions complete unsuccessfully, then the coordinator causes all of the component transactions to abort. If all of the component transactions complete successfully, the coordinator causes all of the component transactions to commit their results.

Figure 7 is a class diagram that shows the roles in which objects participate in the Two Phase Commit pattern.

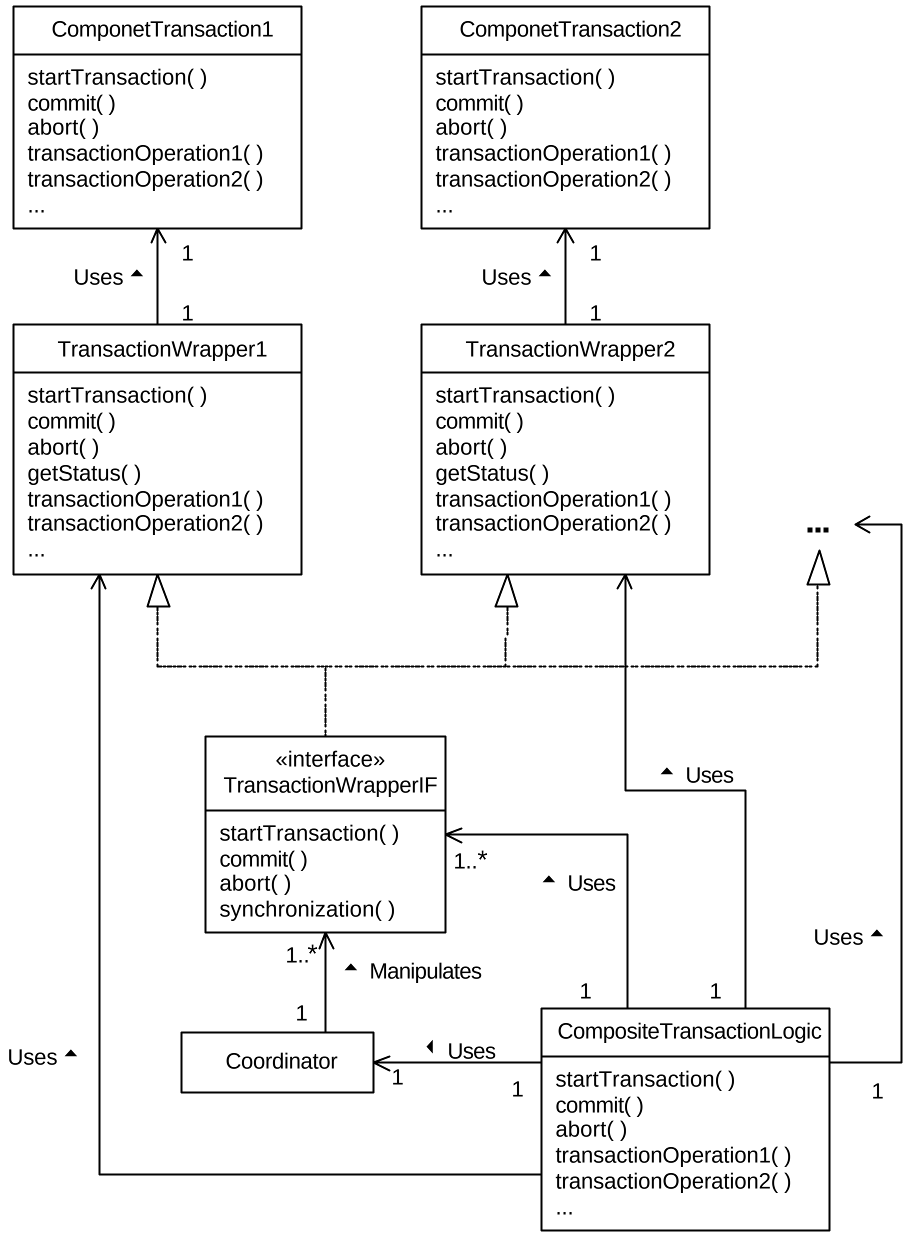


Figure 7: Two Phase Commit Pattern

Here are descriptions of the roles in which classes participate in the Two Phase Commit pattern:

CompositeTransactionLogic

A class in this role is responsible for the top level logic of a composite transaction.

Coordinator

An instance of a class in this role is responsible for coordinating the component transactions of a composite transaction. It determines if they are all successful and then either tells them all to commit or to all abort. Classes in this role are usually reusable and contain no application specific code.

ComponentTransaction1, ComponentTransaction2, …

Classes in this role encapsulate the component transactions that comprise the composite transaction.

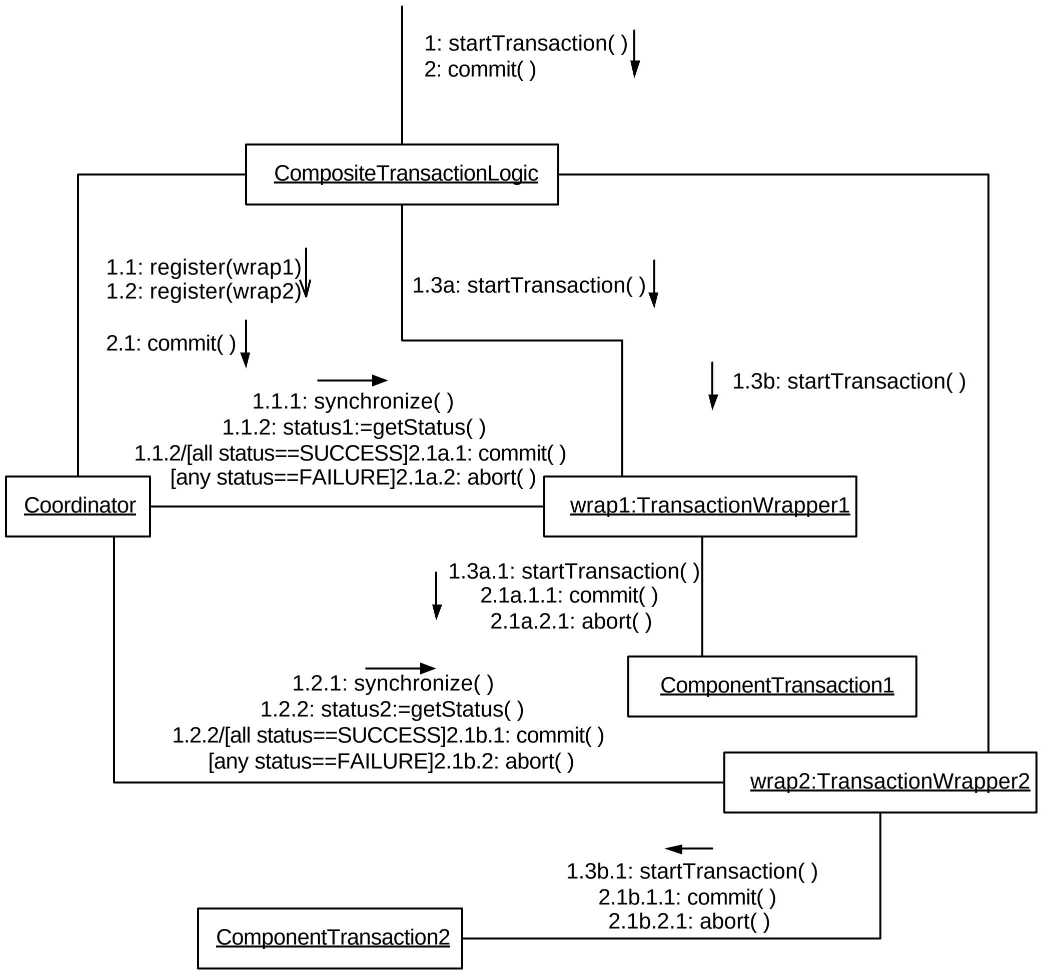
TransactionWrapper1, TransactionWrapper2, …

`ComponentTransaction` objects can only participate directly in the Two Phase Commit pattern if they are designed to do so. That is usually not the case. `ComponentTransaction` objects that are not designed to directly participate in the Two Phase Commit pattern can do so through a wrapper object that provides the logic necessary for it to do so. Classes in this role are those wrapper objects. The details of the logic they need to provide are discussed later in this section.

TransactionWrapperIF

Classes in the `TransactionWrapper` role must implement this interface, which is required by the `Coordinator` class.

The collaboration diagram in Figure 8 illustrates the way that these classes work together.



1: startTransaction( )
2: commit( )
CompositeTransactionLogic
1.1: register(wrap1)
1.2: register(wrap2)
2.1: commit( )
1.3a: startTransaction( )
1.3b: startTransaction( )
1.1.1: synchronize( )
1.1.2: status1:=getStatus( )
1.1.2/[all status==SUCCESS]2.1a.1: commit( )
[any status==FAILURE]2.1a.2: abort( )
Coordinator
wrap1:TransactionWrapper1
1.3a.1: startTransaction( )
2.1a.1.1: commit( )
2.1a.2.1: abort( )
ComponentTransaction1
1.2.1: synchronize( )
1.2.2: status2:=getStatus( )
1.2.2/[all status==SUCCESS]2.1b.1: commit( )
[any status==FAILURE]2.1b.2: abort( )
wrap2:TransactionWrapper2
1.3b.1: startTransaction( )
2.1b.1.1: commit( )
2.1b.2.1: abort( )
ComponentTransaction2

Figure 8: Two Phase Commit Collaboration

Here is a step-by-step description of the interactions shown in Figure 8:

1

The composite transaction is started.

1.1, 1.2

The composite transaction registers the objects that wrap the component transactions with the `Coordinator` object. This simplifies the logic of the composite transaction by allowing it to commit or abort the transaction with a single call to the `Coordinator` object. It also allows the `Coordinator` object to provide better handling of component transactions that fail. This is discussed in more detail under the “Implementation” heading.

These calls are asynchronous. They start another thread to do their work and then return immediately.

1.1.1, 1.2.1

The `Coordinator` object calls the `synchronize` method of the objects that wrap the component transactions. The purpose of these calls is so that the `Coordinator` object knows when each of the component transactions has completed. Calls to the `synchronize` methods do not return until the component transaction they are associated with completes.

The call that precedes these calls is asynchronous. These calls made by the `Coordinator` object are made in a different thread than the calls to the `Coordinator` object. The objects that call the `Coordinator` object's `register` method are able to go about their business while the `Coordinator` object is waiting for its calls to `synchronize` methods to return.

1.1.2, 1.2.2

The `Coordinator` object calls the `getStatus` method of the objects that wrap the component transactions. The purpose of these calls is to determine if each component transaction was successful or failed.

1.3a, 1.3b

The logic of the composite transactions starts each of the component transactions by calling the `startTransaction` method of the wrapper object for each component transaction. When possible and advantageous, the invocations of the component transactions are concurrent. In many cases, that is not possible. Sequential invocation of component transactions is more common than concurrent invocation.

1.3a.1, 1.3b.1

The wrapper objects start their corresponding component transaction.

2

The `CompositeTransactionLogic` object's `commit` method is called. A call to the `commit` method requests the object to commit the results of the composite transaction.

2.1

The `CompositeTransactionLogic` object delegates the work of committing the composite transaction to the `Coordinator` object by calling its `commit` method.

2.1a.1, 2.1b.1

For each component transaction, the `Coordinator` object waits for the transaction to complete. If the status of all of the component transactions indicate that they completed successfully and will be able to commit their results successfully, then the `Coordinator` object calls the `commit` method of each component transaction's wrapper object.

2.1a.1.1, 2.1b.1.1

If their `commit` method is called, the wrapper objects for the component transactions commit those transactions.

2.1a.2, 2.1b.2

If any component transactions did not complete successfully or their status indicates that some of them will not be able to successfully commit their results, then the `Coordinator` object calls the `abort` method of each component transaction's wrapper object.

2.1a.2.1, 2.1b.2.1

If their `abort` method is called, the wrapper objects for the component transactions abort those transactions.

Based on this, the additional logic that wrapper objects for component transactions may be required to provide are:

- The ability to tell if a component transaction has completed its work successfully and will be able to successfully commit its results.

- The ability to determine if a component transaction will be unable to complete its work.

When a Component Transaction Fails

A `Coordinator` object learns that a component transaction has failed when its wrapper's `getStatus` method returns failure. If any of the component transactions fail, the `Coordinator` object aborts them all.

When a Component Transaction Object Crashes

When a `Coordinator` object learns that a component transaction object has crashed, it also aborts all of the component transactions. `Coordinator` objects are generally not able to learn directly that a component transaction object has crashed. Instead they generally learn indirectly of a crash. A `Coordinator` object may infer that a component transaction object by the amount of time that it takes for the call to the component object's `getStatus` method to return. If the call takes too long to return, the `Coordinator` object may consider the call to have timed out and infer that the call will never return.

If the length of time that a component transaction takes is highly variable, then the amount of time that the must elapse before the `Coordinator` object may consider the call to have timed out may be unreasonable long. In such cases, the `Coordinator` object can use the Heartbeat pattern to detect the crash of a `Coordinator` object.

When The Coordinator Crashes

One other aspect of this pattern to look at more closely is what happens when the `Coordinator` object crashes.

When a `Coordinator` object has a transaction pending, it records in a file the fact that there is a pending transaction and the identities of the transaction's participants. If the `Coordinator` object crashes, it is automatically restarted.

After the `Coordinator` object restarts, it checks the file for pending transactions. When the `Coordinator` object finds a transaction in the file, the transaction will be in one of three states:

- The transaction may be open. In this case, the `Coordinator` object does not need to take any immediate action.
- The transaction may be aborted. This will be the case if the `Coordinator` object received a request to abort the transaction, but had not informed all of the transaction's participants to abort before the crash. In this case, the `Coordinator` object calls the `abort` method of all the transaction's participant's.
- The transaction may be committed. This will be the case if the `Coordinator` object received a request to commit the transaction but had not informed all of the participants to commit before the crash. This is the most interesting of the three cases.

If the `Coordinator` object had been asked to commit the transaction before it crashed, it has to find out the status of the participants before it can proceed. The `Coordinator` object proceeds by calling the `getStatus` method of all the transaction's participants. Each call to a participant's `getStatus` method tells the `Coordinator` object one of three things:

- The participant has committed the transaction.
- The participant is ready to commit the transaction
- The participant is unable to commit the transaction.

Because of the way that two phase commit works, it should never be the case that some participants have been committed and other participants are unable to commit. Once the `Coordinator` object knows the status of all the participants, it either asks all of the participants to commit or all of the participants to abort the transaction.

Consequences

- The Two Phase Commit pattern ensures that all component transactions in a composite transaction either commit their results or abort.

- In most situations, the Two Phase Commit pattern adds only a modest amount of overhead.

- There is a situation in which a transaction implemented using the Two Phase Commit pattern can take an indefinite amount of time to complete.

  The lifetime of a composite transaction is greater than the lifetime of its component transactions. In a distributed environment, it is possible for some components of the transaction to crash or have some other sort of catastrophic failure while the others are alive and well. If the object(s) coordinating a transaction experience a catastrophic failure, the `Coordinator` object can generally detect the failure within a bounded and predetermined amount of time. In such cases, the `Coordinator` object tells the rest of the component transactions to abort.

  If the `Coordinator` object experiences a catastrophic failure, it is not generally possible to guarantee a maximum amount of time it will take for it to be restarted and complete the transaction. That means there is no definite guarantee on how long it can take for coordinated transactions to complete.

- Some transactions cannot participated in the Two Phase Commit pattern because there is no way for a wrapper object to get the information it needs about the transaction.

Implementation

It is possible to guarantee an upper bound on the amount of time that it takes for coordinated transactions to complete by forcing them to abort after a predetermined amount of time has elapsed. This has the unfortunate consequence of creating a period of time when it is possible for the outcome of the coordinated transactions to become inconsistent. The problem arises from the fact that when the object(s) that coordinates a transaction decides that the coordinated transactions should commit their results, the message does not reach the objects responsible for each transaction at the same time. This creates a window of vulnerability between the time that the message to commit reaches the first transaction and the time it reaches the last transaction. During the window, some of the transactions may time out, causing those transactions to abort while those that the message reaches in time commit their results.

In distributed environments, you should ensure that the `Coordinator` object becomes aware of the catastrophic failure of a component transaction within a bounded amount of time. The simplest solution is available if the component transaction takes about the same amount of time to complete every time it runs. In this situation, the `Coordinator` object can detect a catastrophic failure of a component transaction by placing a limit on how much time it will wait for it to complete before it decides that the component transaction has failed.

If the amount of time that a component transaction requires is not predictable, you can use the Heartbeat pattern to ensure that the `Coordinator` object detects a catastrophic failure within a set amount of time.

JAVA API Usage

The Java Transaction API defines interfaces that are suitable for some of the roles of the Two Phase Commit pattern.

Known Uses

The CORBA transaction service supports the Two Phase Commit pattern.

Databases such as Oracle and Sybase that support distributed transactions use the Two Phase Commit pattern.

Related Patterns

ACID Transaction

The Two Phase Commit pattern is used to build composite transactions having the ACID properties from component transactions that have the ACID properties.

Composite Transaction

The Two Phase Commit pattern is used with the Composite Transaction pattern

Decorator

The Decorator pattern (described in [Grand98A]) provides the basis for the organization of the wrapper objects used in the Two Phase Commit pattern.

Heartbeat

The Heartbeat pattern may be used with the Two Phase Commit pattern to ensure that the `Coordinator` object is able to detect catastrophic failures of component transactions in a bounded amount of time.

Process Pair

The Process Pair pattern may be used to restart a `Coordinator` object after it crashes.

Audit Trail

Synopsis

You need to verify that transactions are being processed correctly and honestly. Maintain an historical record of transactions that have been applied to an object or set of objects. The record should contain enough detail to determine how the objects affected by the transactions reached their current state.

Context

Suppose that you are designing software for a business that will serve as a clearinghouse for barter exchanges. Each transaction will involve the exchange of a combination of goods and services.

Each day, the clearinghouse's clients make deals. At the end of each day, each client is expected to consummate the their trades through the exchange of certificates promising the future delivery of goods or services. The clearinghouse provides its clients with the necessary digital certificates it expects them to digitally sign and forward to the indicated recipient. The recipient on the certificate may be a different party than anyone that the client made any direct deal with, because of subsequent trades.

Clearing house clients must trust the clearinghouse to correctly identify the recipients of the goods or services that they have traded away. For that reason there must be a way to verify the correctness and honesty of the clearinghouse. One way to do that is to keep an audit trail.

The audit trail will consist of a record of all of the trades. By reviewing randomly selected sequences of transactions, it is possible for auditors to verify that the transactions are handled correctly and honestly.

Forces

- You need a record of the transactions that have modified the state of an object or set of objects in order to determine if the current state of the object(s) is correct.

- You need to account for the actions of an object. The need for accountability can come from the application domain. For example, in accounting and finance applications real world financial events drive the actions of objects. These applications generally have requirements that it be possible to audit the actions of these objects so that they can be compared with real world events. Such audits provide an opportunity to detect human error and dishonesty in recording financial events.

  The need for object accountability may come from internal design considerations such as the need to review an object's actions for security or debugging purposes. A record of an object's actions may help detect patterns in a hacker's actions. When debugging a program, comparing a record of an object's actions with its expected actions can help in tracking down bugs.

- Once a record is made of a transaction, it must not be possible to alter that record. If it is possible to alter it, then you cannot be sure of what actually happened.

- The number of recorded transactions consistently grows but the amount of available on-line storage does not.

- The purpose of keeping an historical record of transactions is to enable auditors or troubleshooters to verify that the transactions in the record satisfy a set or expectations or requirements. In many cases, the volume of transactions makes it impractical for people to examine every individual transaction. In such cases, people will need a way to examine samples or summaries of the transactions.

Solution

Maintain an historical record of transactions. The record should include all transactions that effect the state of objects of interest. In order to use the historical record to determine if the objects are currently in the correct state, it is necessary to determine the object's original state. For this purpose, you also store the original state of the object. Knowing the original state of an object makes it possible to determine if the transactions applied to an object after it was in that state should have brought the object to its current state or not.

The record should also include transactions initiated by objects of interest. The purpose of such an historical record is to record the behavior of the object that initiates the transactions. In many cases it is necessary to for the historical record to include information about the object's state at the time it initiated a transaction in order to evaluate the object's behavior.

To facilitate the analysis of transaction records, the transaction records should be under the control of a mechanism such as a database manager that allows people to extract samples or summaries of the transaction. For example, consider the situation described under the context heading. It would be desirable to be able to pick an arbitrary clearinghouse client and review the sequence of trades that resulted in the client being told to send his trade goods to a particular recipient.

Consequences

- If you use the Audit Trail pattern to keep track of the transactions that an object is given to process, then you can determine if the object is in its correct state by auditing the transactions in the audit trail.

- If you used the Audit Trail pattern to keep track of the transactions an object initiates, you will have a way to validate its behavior or debug it.

- The audit trail pattern adds complexity to designs.

- The storage requirements for maintaining an audit trail can be very large. As time goes on, an audit trail will continue to grow. If the audit trail is to be kept on-line, there is generally a limit to the amount of storage that is available to for storing the audit trail. For that reason, it is common to move portions of an on-line audit trail that exceed a particular age to removable storage such as a tape.

Implementation

When an audit trail is mandated by application requirements, the collection of historical transactions that constitutes the audit trail is called a *journal*. As is the case with all audit trails, the transactions in a journal never change because they are a true and accurate record of history. If an application processes a transaction whose purpose is to correct the effects of a previous erroneous transaction, the transaction that corrects the problem is called an *adjustment transaction*.

There are many ways to implement the Audit Trail pattern. Figure 9 shows a sample design to implement the Audit Trail pattern.

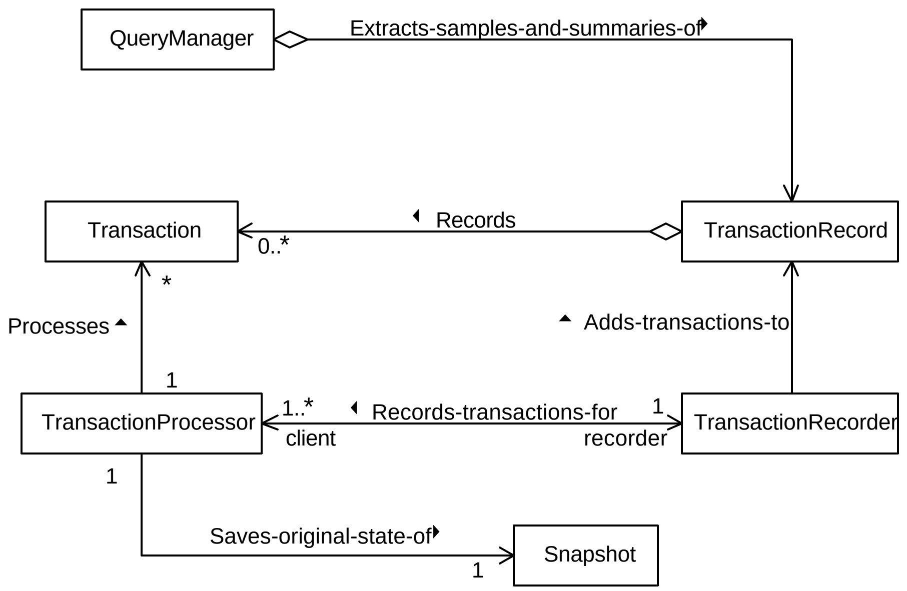


Figure 9: Audit Trail Pattern

In the design shown in Figure 4.9, `TransactionProcessor` objects process transactions encapsulated in `Transaction` objects. As it processes transactions, it passes them to a `TransactionRecorder` object, which adds the transactions to the collection of `Transaction` objects it manages. `TransactionRecorder` objects are responsible for adding information to the transactions that will be needed for the historical record. One such commonly needed piece of information is the time that a transaction was processed.

Figure 9 also includes a snapshot object that encapsulates the original state of the `TransactionProcessor` object. The contents of that object are used when analyzing the historical transaction record.

One other detail in Figure 9 is a `QueryManager` class. It is responsible for generating subsets and summaries of the historical transaction record as they are requested by auditors.

To validate an object's current state from an historical record of transactions, the historical record does not need to include failed transactions. There may be advantages to including failed transactions in the historical record. It can facilitate debugging and detection of security problems. Because those are not continuing needs, if there is support for failed transactions in the historical record it should be possible to turn off the inclusion of failed transactions. If it is possible to include failed transactions in an historical record some of the time, then all transactions in the record must include an indication of whether each transaction was successful or not.

The volume of transactions in some applications may make it impractical to store and manage a complete audit trail. In such cases, it may be possible to keep a partial audit trail that still allows valid audits to be performed.

When moving transaction off-line the object's state that they gave rise to should be determined and stored on-line. That makes it possible to analyze the on-line portion of the historical record without having to access off-line information.

For some applications, it is not possible to sufficiently limit the time historical transaction records are kept on-line to keep the on-line storage requirement small enough. If the application processes transactions for many objects, it may be sufficient to only keep an historical transaction record for randomly selected objects.

Known Uses

Accounting applications intended for medium to large applications support the Audit Trail pattern.

Workflow applications generally provide an audit trail that allows people to find out who did what with a work item.

Source code management systems such as CVS provide an audit trail describing changes that have been made to source code and who made them.

Related Patterns

ACID Transaction

If the transactions in an historical record do not have the ACID properties, then it may not be possible to unambiguously determine the effect of each transaction on an object.

Snapshot

The Snapshot pattern (described in [Grand98A]) provides advice on how to capture the state of an object.

Bibliography

[Date94] Chris J. Date. An Introduction to Database Systems. Addison-Wesley, Reading, MA, 1994.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[Grand98A] Mark Grand. Patterns in Java, Volume 1. John Wiley & Sons, New York, NY, 1998.

[Grand98B] Mark Grand. Patterns in Java, Volume 2. John Wiley & Sons, New York, NY, 1998.

[Gray-Reuter93] Jim Gray, Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufman Publishers, San Mateo, California, 1993.