**Exclusion 🡪 Synchronization 🡪 Traversal**

**2.2.3 Traversal**

In fully synchronized classes, you can add another atomic operation just by encasing it in a

synchronized method. For the sake of reusability and convenience, it is often a good idea to add

small suites of such operations to general-purpose classes or their subclasses. This avoids making

clients go through contortions trying to construct atomic versions of commonly used operations out of

smaller components. For example, it would be useful to define synchronized versions of

removeFirst, prepend, and similar methods to ExpandableArray, as found in

java.util.Vector and other collection classes.

However, this strategy doesn't work for another common usage of collections, *traversal.* A traversal

iterates through all elements of a collection and performs some operation on or using each element.

Since there are an unbounded number of operations clients might want to apply to the elements of a

collection, it is pointless to try to code all of them as synchronized methods.

There are three common solutions to this design problem, aggregate operations, indexed traversal, and

versioned iterators, each of which reflect different design trade-offs. (See § 2.4.1.3, § 2.4.4, and §

2.5.1.4 for additional strategies that apply to other kinds of collection classes.) The issues and tradeoffs

encountered in each approach are seen more generally in the design of many classes using locks.

***2.2.3.1 Synchronized aggregate operations***

One way to secure traversal is to abstract out the operation being applied to each element so that it can

be sent as an argument to a single synchronized applyToAll method. For example:

interface Procedure {

void apply(Object obj);

}

class ExpandableArrayWithApply extends ExpandableArray {

public ExpandableArrayWithApply(int cap) { super(cap); }

synchronized void applyToAll(Procedure p) {

for (int i = 0; i < size; ++i)

p.apply(data[i]);

}

}

This could be used, for example, to print all elements in collection v:

v.applyToAll(new Procedure() {

public void apply(Object obj) {

System.out.println(obj)

}

} );

This approach eliminates potential interference that could occur if other threads attempted to add or

remove elements during traversal, but at the expense of possibly holding the lock on the collection for

prolonged periods. While this is often acceptable, it may lead to the kinds of liveness and performance

problems that motivated the default rule in § 1.1.1.1 saying to release locks when making method calls

(here, to apply).

***2.2.3.2 Indexed traversal and client-side locking***

A second traversal strategy available with ExpandableArray is to require clients to use the

indexed accessor methods for traversal; for example:

for (int i = 0; i < v.size(); ++i) // Do not use

System.out.println(v.get(i));

This avoids holding the lock on v while performing each element operation, but at the expense of two

synchronized operations (size and at) per element. More importantly, the loop must be rewritten to

handle a potential interference problem resulting from the finer locking granularity: It is possible for

the check of i < v.size() to succeed but for another thread to remove the current last element

before the call to v.get(i). One way to deal with this is to employ *client-side locking* to preserve

atomicity across the size check and access:

for (int i = 0; true; ++i) { // Limited utility

Object obj = null;

synchronized(v) {

if (i < v.size())

obj = v.get(i);

else

break;

}

System.out.println(obj);

}

However, even this can be problematic. For example, if the ExpandableArray class supported

methods to rearrange elements, this loop could print the same element twice if v were modified

between iterations.

As a more extreme measure, clients can surround the entire traversal with synchronized(v).

Again, this is often acceptable but can induce the long-term locking problems seen in

synchronized aggregate methods. If the operations on elements are time-consuming, the client

can instead first make a copy of the array for traversal purposes:

Object[] snapshot;

synchronized(v) {

snapshot = new Object[v.size()];

for (int i = 0; i < snapshot.length, ++i)

snapshot[i] = v.get(i);

}

for (int i = 0; snapshot.length; ++i) {

System.out.println(snapshot[i]);

}

Client-side locking tends to be used more extensively in non-object-oriented approaches to

multithreaded programming. This style is sometimes more flexible, and can be useful in OO systems

when instances of a class are designed to be embedded within others (see § 2.4.5) and so must give up

internal responsibility for synchronization decisions.

But client-side locking avoids potential interference problems at the expense of encapsulation

breakdown. Correctness here relies on special knowledge of the inner workings of the

ExpandableArray class that may fail to hold if the class is later modified. Still, this may be

acceptable in closed subsystems. Client-side locking can also be a reasonable option when classes

document these usages as sanctioned. This also constrains all future modifications and subclasses to

support them as well.

***2.2.3.3 Versioned iterators***

A third approach to traversal is for a collection class to support *fast-fail* iterators that throw an

exception if the collection is modified in the midst of a traversal. The simplest way to arrange this is to

maintain a version number that is incremented upon each update to the collection. The iterator can

then check this value whenever asked for the next element and throw an exception if it has changed.

The version number field should be wide enough that it can never wrap around while a traversal is in

progress. An int normally suffices.

This strategy is used in the java.util.Iterator classes in the collections framework. We can

apply it here to a subclass of ExpandableArray that updates version numbers as an after-action

(see § 1.4.3):

class ExpandableArrayWithIterator extends ExpandableArray {

protected int version = 0;

public ExpandableArrayWithIterator(int cap) { super(cap); }

public synchronized void removeLast()

throws NoSuchElementException {

super.removeLast();

++version; // advertise update

}

public synchronized void add(Object x) {

super.add(x);

++version;

}

public synchronized Iterator iterator() {

return new EAIterator();

}

protected class EAIterator implements Iterator {

protected final int currentVersion;

protected int currentIndex = 0;

EAIterator() { currentVersion = version; }

public Object next() {

synchronized(ExpandableArrayWithIterator.this) {

if (currentVersion != version)

throw new ConcurrentModificationException();

else if (currentIndex == size)

throw new NoSuchElementException();

else

return data[currentIndex++];

}

}

public boolean hasNext() {

synchronized(ExpandableArrayWithIterator.this) {

return (currentIndex < size);

}

}

public void remove() {

// similar

}

}

}

Here, the print loop would be expressed as:

for (Iterator it = v.iterator(); it.hasNext();) {

try {

System.out.println(it.next());

}

catch (NoSuchElementException ex) { /\* ... fail ... \*/ }

catch (ConcurrentModificationException ex) {

/\* ... fail ... \*/

}

}

Even here, choices for dealing with failures are often very limited. A

ConcurrentModificationException often signifies unplanned, unwanted interactions

among threads that should be remedied rather than patched over.

The versioned iterator approach encapsulates the design choices underlying the data structure, at the

price of occasionally undue conservatism. For example, an interleaved add operation would not

interfere with the required semantics of a typical traversal, yet would cause an exception to be thrown

here. Versioned iterators are still a good default choice for collection classes, in part because it is

relatively easy to layer aggregate traversal or client-side locking on top of these iterators, but not vice

versa.

***2.2.3.4 Visitors***

The *Visitor* pattern described in the *Design Patterns* book extends the notion of iterators to provide

support for clients performing operations on sets of objects connected in arbitrary ways, thus forming

the nodes of some kind of tree or graph rather than the sequential list seen in ExpandableArray.

(Less relevantly here, the Visitor pattern also supports polymorphic operations on each node.)

The options and concerns for visitors and other extended senses of traversal are similar to, and can

sometimes be reduced to, those seen in simple iterators. For example, you might first create a list of all

nodes to traverse and then apply any of the above techniques for traversing the list. However, locks

here would lock only the list, not the nodes themselves. This is usually the best policy. But if you need

to ensure that all of the nodes are locked during the entire traversal, consider forms of confinement

(see § 2.3.3) or containment locking (see § 2.4.5).

Conversely, if traversal is arranged by every node supporting a nextNode method, and you do not

want to end up simultaneously holding all locks to all nodes encountered during traversal,

synchronization of each node must be released before proceeding to the next node, as described in §

2.4.1 and § 2.5.1.4.