ALLOCATION LIBRARY: PROJECT STATUS

This document describes the status of the SAR Allocation Library project (November 2001 – February 2002) at its conclusion. It includes a list of the completed tasks, and a list of possible future developments (things that were planned, but not implemented due to a lack of time).

**COMPLETED TASKS**

DESIGN:

* Design document “allocation-design-6.doc” completed, although design changes have since made parts of this document obsolete (and some further design changes are listed under “Possible Future Developments” below).

The design covers the Allocation and Washburn classes (although these classes have a different form to those implemented in code and discussed below).

The document should still be useful when writing the code that implements the Washburn algorithm, as most of the Washburn class will remain the same (it is only the interface that will need to change).

CODE:

Allocation class and CharnesCooper class:

* The abstract class Allocation and the concrete class CharnesCooper are operational.

However, a number of the “Possible Future Developments” (below) involve changes to these classes.

Washburn class:

* No code written.

Design completed (and then altered), as noted above.

TESTING:

* Testing completed on:
* CharnesCooper::calcAllocation()

This function is tested in CC-test.cpp.

* CharnesCooper::setAllocation()

CharnesCooper::getCoverage()

CharnesCooper::getPOD()

CharnesCooper::getPOS()

CharnesCooper::getNewPOC()

CharnesCooper::getTotalPOS()

These functions are tested in Clc-test.cpp.

* CharnesCooper::getAssignmentsForResource()

CharnesCooper::getAssignmentsForArea()

CharnesCooper::getAssignedAreas()

These functions, and the associated iterators, are tested in Itr-test.cpp.

Each of the files listed above contains a main() which runs tests on the indicated functions.

* Testing incomplete on:
* Although getAssignmentsForResource(), getAssignmentsForArea(), getAssignedAreas(), getCoverage(), getPOD(), getPOS(), getNewPOC() and getTotalPOS() have been tested within the CharnesCooper class, they have not been tested in cases where multiple resources may be used.

Much of this code is either within the (abstract) Allocation class or is likely to be copied into other concrete classes derived from Allocation, such as Washburn or ArbitraryAllocation (see below). If these changes are made, the functions listed above will need to be tested in cases where there are multiple resources.

* No private functions have been directly tested.

POSSIBLE FUTURE DEVELOPMENTS

* Although the public functions of the CharnesCooper class have been tested, none of its private functions have been (directly) tested. It would be useful to directly test these functions to ensure that they are operating as expected.
* Alter parameter passing to use variable‑length arrays, where possible.
* Add methods getNumAreas() and getNumResources() to the Allocation class, enabling the numbers of areas and resources to be extracted from an Allocation object. These values may then be used by a caller to iterate through Areas or Resources when calling getAssignmentsForResource(), getAssignmentsForArea(),getCoverage(), getPOD(), getPOS() and getNewPOC() to extract information from an Allocation object.
* Make the CharnesCooper class force the number of resources to be 1.

The argument int p\_no\_resources should be removed from the constructor, which should simply set myNumResources = 1.

* Simplify and improve the class hierarchy.

At present, the three types of iterators (ResourceIterator, AreaIterator and AssignedAreasIterator) have a (common) abstract base class Iterator, whilst the types of data they return (ResourceAssignment, AreaAssignment and AssignedArea) have a (common) base class Container. The iterators and containers are then accessed via pointers to Iterators and Containers, to allow them to be used polymorphically.

However, this structure does not makes sense, as it implies that the three types of iterators provide a common interface and the three types of containers provide a common interface. The containers provide access to different types of data and therefore have different interfaces, whilst the iterators have different interfaces because they provide access to different types of containers.

NOTE: At present, the iterators and containers are forced to have common interfaces, which is causing problems:

* ResourceAssignment does not store a resource number, yet has a member function getResourceNum() which always returns –1. (Similar problems exist for the other container types.)
* The pure virtual Iterator::operator()() function must return a Container\* (rather than a container object of a specific type), in order for this function to be included in the abstract base class Iterator, where the actual container type is unknown.

When the user calls the operator()() function of an iterator, they must therefore store the returned Container\* value so that the (dynamically allocated) object that it points to may be subsequently deleted. This is very inconvenient for the user.

Note that there is no need for iterators and containers to be used polymorphically with respect to the type of iterator – when would you ever not know what type of iterator or container you were using?

This class hierarchy should be simplified as follows:

* ResourceAssignment and AreaAssignment should be made separate concrete classes, with no base class.
* AssignedAreas should be removed altogether.

(AssignedAreasIterator::operator()() is supposed to return an integer only.)

* Separate abstract base classes should be created for ResourceIterators, AreaIterators and AssignedAreasIterators. The iterators may then be used in a polymorphic manner with respect to the type of object that they are iterating over, but not with respect to the type of iterator (which is unnecessary as you will always know whether you are using a ResourceIterator, AreaIterator or AssignedAreasIterator).

The operator()() functions for the abstract base classes of ResourceIterator, AreaIterator and AssignedAreasIterator should be altered to return objects of type ResourceAssignment, AreaAssignment and int, respectively. Returning objects, rather than pointers to them, eliminates the need for the user to deallocate the memory for the object being pointed to.

Some of the functions currently implemented in Iterator may need to be made pure virtual. (The current implementations may rely on the fact that ResourceIterators, AreaIterators and AssignedAreasIterators currently iterate over a 2D array of assignments, so these implementations cannot be included in the abstract base classes.)

The Iterator::Index() function can be removed, as it is not needed.

* Iterator may be retained as an abstract base class, but this is probably unnecessary. If it is retained, the pure virtual operator()() should be removed, as this member function must return a different type of container for each type of iterator.
* Add constructors of the form shown below to all concrete classes derived from Allocation, enabling new objects of these classes to be constructed from pointers to current Allocation objects. (Note that the actual (concrete) type of the object pointed to by previousAllocation should not matter.) This allows the Allocation object for a sortie to be constructed from the Allocation object for the previous sortie.

New constructors for Allocation-type classes:

* Allocation(int numResources, const double\* effectiveness, const Allocation\* previousAllocation)

numAreas is the same as that for previousAllocation.

POC values are the NewPOC values (after the previous sortie is completed) from previousAllocation.

* Allocation(const double\* effectiveness, const Allocation\* previousAllocation)

numAreas is the same as that for previousAllocation.

POC values are the NewPOC values (after the previous sortie is completed) from previousAllocation.

numResources is the same as that for previousAllocation.

* Allocation(const Allocation\* previousAllocation)

numAreas is the same as that for previousAllocation.

POC values are the NewPOC values (after the previous sortie is completed) from previousAllocation.

numResources is the same as that for previousAllocation.

effectiveness is the same as that for previousAllocation.

* Remove the pure virtual function calcAllocation() from the (abstract) Allocation class and alter all derived classes so that they perform any allocation calculations upon construction. (The derived classes may keep their calcAllocation() functions as private functions, which are called from the constructor.)

This change makes sense because an Allocation object is almost useless until it has performed the allocation calculations.

* Create an ArbitraryAllocation class with a constructor that accepts an arbitrary set of assignments as an argument. It does not perform any allocation calculations. This class is used to calculate coverage, POD, POS, NewPOC and TotalPOS for arbitrarily specified allocations.

Note that there is no need for other derived classes of Allocation to have setAllocation() functions (except possibly for testing purposes), as this class should handle all arbitrarily specified allocations.

Once this class is created, getAssignmentsForResource(), getAssignmentsForArea(), getAssignedAreas(), getCoverage(), getPOD(), getPOS(), getNewPOC() and getTotalPOS() should all be tested for cases with multiple resources (this testing is currently impossible because the CharnesCooper class can only handle one resource).

* Create the functions:

Allocation\* bestAllocation(int numResources, int numAreas, const double\* effectiveness, const double\* POC, const double\* searchEndurance)

Allocation\* bestAllocation(int numResources, const double\* effectiveness, const Allocation\* previousAllocation, const double\* searchEndurance)

Allocation\* bestAllocation(const double\* effectiveness, const Allocation\* previousAllocation, const double\* searchEndurance)

Allocation\* bestAllocation(const Allocation\* previousAllocation, const double\* searchEndurance)

which use any or all of numResources, numAreas, effectiveness, POC and searchEndurance to select the best type of Allocation object to use, construct an object of that type (the allocation will be performed upon construction) and return a pointer to the object. Note that the values for numResources, numAreas, effectiveness and POC are obtained from the previous Allocation object (\*previousAllocation) if they are not included as arguments to bestAllocation, in the same way as for the additional Allocation class constructors (see above).

* Add additional range‑checking to Allocation and its derived classes (currently only CharnesCooper). Specifically:
* The constructor must check that all arguments are within the required ranges.

For classes that calculate allocations, these arguments are: numResources, numAreas, effectiveness, POC and searchEndurance.

For the ArbitraryAllocation class, these arguments are: numResources, numAreas, effectiveness, POC and assignments.

* getAssignmentsForResource() must check that the argument resourceNum is a valid resource number.
* getAssignmentsForArea() must check that the argument areaNum is a valid area number.
* getCoverage(), getPOD(), getPOS() and getNewPOC() must check that the argument areaNum is a valid area number.

If values are found to be out of range, exceptions should be thrown. Note that exceptions should also be thrown in the following cases:

* The operator()() functions for the three types of iterators currently call exit(1) if they are called when the iterator does not point to an item in the list. This behaviour should be altered so that they instead throw an exception.