

RESTRICTED ACCESS CONTAINERS*

Stephen Wong

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 2.0[†]

Abstract

Restricted access containers provided encapsulated storage of information for algorithms where the storage is decoupled from the details of insertion and removal of data from the storage.

1 Introduction

Stacks and queues are examples of containers with special insertion and removal behaviors and a special access behavior.

Insertion and removal in a stack must be carried out in such a way that the last data inserted is the first one to be removed. One can only retrieve and remove a data element from a stack by way of special access point called the "top". Traditionally, the insertion and removal methods for a stack are called push and pop, respectively. push inserts a data element at the top of the stack. pop removes and returns the data element at the top of the stack. A stack is used to model systems that exhibit **LIFO (Last In First Out)** insert/removal behavior.

Data insertion and removal in a queue must be carried out in such a way that the first one to be inserted is the first one to be removed. One can only retrieve and remove a data element from a queue by way of special access point called the "front". Traditionally, the insertion and removal methods for a queue are called **enqueue** and **dequeue**, respectively. enqueue inserts a data element at the "end" of the queue. dequeue removes and returns the data element at the front of the queue. A queue is used to model systems that exhibit **FIFO (First In First Out)** insertion/removal behavior. For example, one can model a movie ticket line by a queue.

We abstract the behaviors of special containers such as stacks and queues into an interface called `IRACContainer` specified as follows.

2 Restricted Access Containers

2.1 `IRACContainer.java`

```
package rac;

import listFW.*;
/**
 * Defines the interface for a restricted access container.
 */
```

*Version 1.1: Jun 26, 2008 4:08 pm -0500

[†]<http://creativecommons.org/licenses/by/2.0/>

```

public interface IRAContainer {
    /**
     * Empty the container.
     * NOTE: This implies a state change.
     * This behavior can be achieved by repeatedly removing elements from this IRAContainer.
     * It is specified here as a convenience to the client.
     */
    public void clear();
    /**
     * Return TRUE if the container is empty; otherwise, return
     * FALSE.
     */
    public boolean isEmpty();
    /**
     * Return TRUE if the container is full; otherwise, return
     * FALSE.
     */
    public boolean isFull();
    /**
     * Return an immutable list of all elements in the container.
     * @param fact for manufacturing an IList.
     */
    public IList elements(IListFactory fact);
    /**
     * Remove the next item from the container and return it.
     * NOTE: This implies a state change.
     * @throw an Exception if this IRAContainer is empty.
     */
    public Object get();
    /**
     * Add an item to the container.
     * NOTE: This implies a state change.
     * @param input the Object to be added to this IRAContainer.
     * @throw an Exception if this IRAContainer is full.
     */
    public void put(Object input);
    /**
     * Return the next element in this IRAContainer without removing it.
     * @throw an Exception if this IRAContainer is empty.
     */
    public Object peek();
}

```

1. Restrict the users from seeing inside or working on the inside of the container.
2. Have simple put(data) and get() methods. Note the lack of specification of how the data goes in or comes out of the container.
3. However, a "policy" must exist that governs how data is added ("put") or removed ("get"). Examples:
 - First in/First out (FIFO) ("Queue")
 - Last in/First out (LIFO) ("Stack")

- Retrieve by ranking ("Priority Queue")
 - Random retrieval
4. The policy is variant behavior → abstract it.
- The behavior of the RAC is independent of exactly what the policy does.
 - The RAC delegates the actual adding ("put") work to the policy.
 - The RAC is only dependent on the existence of the policy, not what it does.
 - The policy is a "strategy" for adding data to the RAC. See the Strategy design pattern¹.
 - Strategy pattern vs. State pattern² – so alike, yet so different!

The manufacturing of specific restricted access containers with specific insertion strategy will be done by concrete implementations of the following abstract factory interface.

2.2 IRACFactory.java

```
package rac;

/**
 * Abstract Factory to manufacture RACs.
 */
public interface IRACFactory {
    /**
     * Returns an empty IRACContainer.
     */
    public IRACContainer makeRAC();
}
```

3 Examples

The following is an (abstract) implementation of `IRACFactory` using `LRStruct` as the underlining data structure. By varying the insertion strategy, which is an `IAlgo` on the internal `LRStruct`, we obtain different types of RAC: stack, queue, random, etc.

¹<http://cnx.org/content/m17037/latest/>

²<http://cnx.org/content/m17047/latest/>

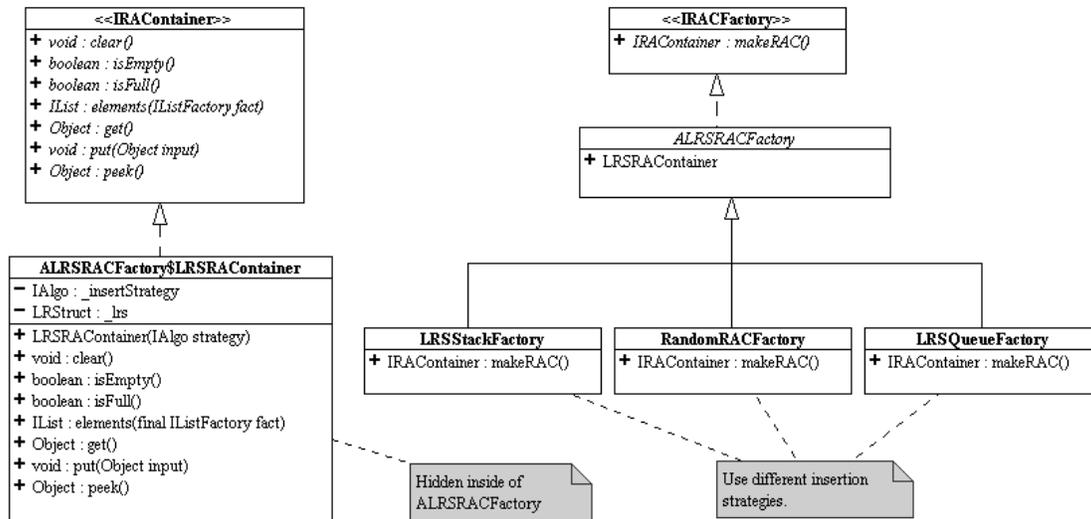


Figure 1: UML diagram of the abstract RAC and RAC factory definitions plus a few concrete RAC factories.

The source code for the following examples can be downloaded at this link³.

3.1 ALRSRACFactory.java

```

package rac;

import listFW.*;
import listFW.factory.*;
import lrs.*;

/**
 * Implements a factory for restricted access containers. These
 * restricted access containers are implemented using an LRStruct to
 * hold the data objects.
 */
public abstract class ALRSRACFactory implements IRACFactory {
    /**
     * Implements a general-purpose restricted access container using
     * an LRStruct. How?
     *
     * The next item to remove is always at the front of the list of
     * contained objects. This is invariant!
     *
     * Insertion is, however, delegated to a strategy routine; and
  
```

³<http://cnx.org/content/m17101/latest/rac.zip>

```

* this strategy is provided to the container. This strategy
* varies to implement the desired kind of container, e.g., queue
* vs. stack.
*
* This nested static class is protected so that classes derived from its
* factory can reuse it to create other kinds of restricted access
* container.
*/
protected static class LRSRAContainer implements IRAContainer {
    private IAlgo _insertStrategy;
    private LRStruct _lrs;

    public LRSRAContainer(IAlgo strategy) {
        _insertStrategy = strategy;
        _lrs = new LRStruct();
    }

    /**
     * Empty the container.
     */
    public void clear() {
        _lrs = new LRStruct();
    }

    /**
     * Return TRUE if the container is empty; otherwise, return
     * FALSE.
     */
    public boolean isEmpty() {
        return (Boolean)_lrs.execute(CheckEmpty.Singleton);
    }

    /**
     * Return TRUE if the container is full; otherwise, return
     * FALSE.
     *
     * This implementation can hold an arbitrary number of
     * objects. Thus, always return false.
     */
    public boolean isFull() {
        return false;
    }

    /**
     * Return an immutable list of all elements in the container.
     */
    public IList elements(final IListFactory fact) {
        return (IList)_lrs.execute(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... nu) {
                return fact.makeEmptyList();
            }
        });
    }
}

```

```

        public Object nonEmptyCase(LRStruct host, Object... nu) {
            return fact.makeNEList(host.getFirst(),
                (IList)host.getRest().execute(this));
        }
    });
}

/**
 * Remove the next item from the container and return it.
 */
public Object get() {
    return _lrs.removeFront();
}

/**
 * Add an item to the container.
 */
public void put(Object input) {
    _lrs.execute(_insertStrategy, input);
}

public Object peek() {
    return _lrs.getFirst();
}
}

}

/**
 * Package private class used by ALRSRACFactory to check for emptiness of its internal LRStruct.
 */
class CheckEmpty implements IAlgo {
    public static final CheckEmpty Singleton= new CheckEmpty();
    private CheckEmpty() {
    }

    public Object emptyCase(LRStruct host, Object... input) {
        return Boolean.TRUE;
    }

    public Object nonEmptyCase(LRStruct host, Object... input) {
        return Boolean.FALSE;
    }
}
}

```

3.2 LRStackFactory.java

```
package rac;
```

```
import lrs.*;

public class LRSSStackFactory extends ALRSRACFactory {
    /**
     * Create a ‘‘last-in, first-out’’ (LIFO) container.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }
        });
    }
}
```

3.3 LRSQueueFactory.java

```
package rac;

import lrs.*;

public class LRSQueueFactory extends ALRSRACFactory {
    /**
     * Create a ‘‘first-in, first-out’’ (FIFO) container.
     */
    public IRAContainer makeRAC() {
        return new LRSRAContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object... input) {
                return host.getRest().execute(this, input);
            }
        });
    }
}
```

3.4 RandomRACFactory.java

```
package rac;

import lrs.*;
```

```
/*
 * Implements a factory for restricted access containers, including a
 * container that returns a random item.
 */
public class RandomRACFactory extends ALRSRACFactory {
    /**
     * Create a container that returns a random item.
     */
    public IRAContainer makeRAC() {
        return new LRSRACContainer(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... input) {
                return host.insertFront(input[0]);
            }

            public Object nonEmptyCase(LRStruct host, Object input) {
                /*
                 * Math.Random returns a value between 0.0 and 1.0.
                 */
                if (0.5 > Math.random())
                    return host.insertFront(input[0]);
                else
                    return host.getRest().execute(this, input);
            }
        });
    }
}
```

But can we push the abstraction further? Is the difference between a stack and a queue really anything more than how the data is ordered?

Now, let's go on an look at the ordering object and priority queues... ⁴

⁴<http://cnx.org/content/m17064/latest/>