# Draft Website Documentation for Issue #222: Multithreaded Use Cases

## Overview

Thread safety can be evaluated from two perspectives:

1. Can multiple Threads create unique instances of a class without unexpected side effects or a shared global state?
2. Can a unique instance of a class be shared and manipulated by more than one Thread simultaneously?

Starting in DDMSence v2.5.0, every class has been implemented in a way that prevents unexpected side effects or shared global state, but only a subset of classes can have their instances shared safely among Threads. This table provides a summary of the relative thread safety of every custom class in DDMSence. The class categories are discussed in-depth below.

|  |  |  |
| --- | --- | --- |
| Class Category | Free of Side Effects? | Shareable Across Threads? |
| immutable | Yes | Yes |
| stateless utility | Yes | Yes |
| synchronized | Yes | Yes |
| thread-localized | Yes | Not Applicable |
| unsafe | Yes | No |

## Class Categories

### Immutable Classes

Immutable classes are read-only and cannot change after instantiation. They can be instantiated in multiple Threads without side effects, and can also be shared by more than one Thread simultaneously. Whenever one of these classes exposes a "get" accessor with a mutable return value (such as an XMLGregorianCalendar, an ArrayList, a XOM Element, or a XOM Attribute), the returned object will always be a new copy, detached from the original. Classes in this category include:

* All DDMS component classes (classes which implement the IDDMSComponent interface) and their superclasses
* All attribute group classes (classes which extend from AbstractAttributeGroup) and their superclass
* All interfaces (classes which have a capital letter "I" as a class name prefix)
* The OutputFormat enumeration
* The ValidationMessage class

### Stateless Utility Classes

These classes are simple sets of stateless utility methods, whose output depends solely on input parameters. These functional methods can be called from multiple Threads without side effects, and are implicitly shared by all Threads. The classes may perform some private caching for performance reasons, but it is done in a thread-safe way.

* The ISMVocabulary class
* The Util class

### Synchronized Classes

Although these classes might contain mutable fields, access to those fields is protected with the Java synchronized keyword. They can be instantiated in multiple Threads without side effects. They can also be shared by more than one Thread simultaneously, but are generally intended for read-only, informational purposes.

* All custom exception classes (classes which extend from java.lang.Exception)

### Thread-Localized Classes

Although these classes might contain mutable fields, access to those fields is protected with Java ThreadLocal wrappers, ensuring that each running Thread has its own copy. These localized classes can be instantiated in multiple Threads without side effects, but by their nature, cannot be shared by more than one Thread simultaneously.

* The DDMSVersion class
* The PropertyReader class

### Unsafe Classes

These classes are mutable, and may change after instantiation. They can be instantiated in multiple Threads without side effects, but cannot be shared by more than one Thread simultaneously.

* All DDMS component builder classes (classes which implement the IBuilder interface) and their superclasses
* The DDMSReader class
* The LazyList class

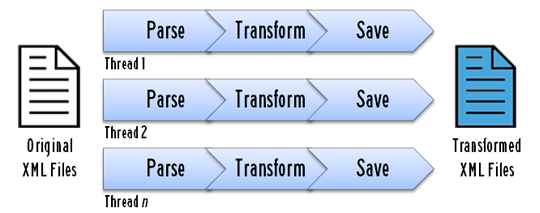
## Multithreaded Use Cases

DDMSence is primarily designed to support multithreaded use cases where Threads have no dependency on each other. Multiple Threads can be parsing, validating, building, or transforming DDMS components, with the expectation that no two Threads are operating on the same components at the same time.

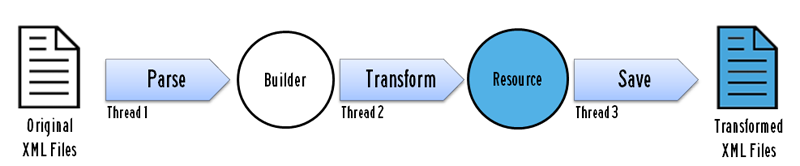
As an example, consider an input set of millions of DDMS 2.0 XML records. You wish to use DDMSence to transform them all up to DDMS 4.1. This operation has 3 discrete steps:

1. **Parse** the old XML file and set up a mutable Builder.
2. **Transform** the record using Builder method calls.
3. **Save** the record as a DDMS 4.1 XML file.

A basic approach would be to perform all 3 steps within a single Thread, but load-balance the millions of records across many Threads. Each Thread owns a separate batch of records, and is guaranteed to run without any dependency on (or interference from) other Threads.



A more advanced approach would be to assign the responsibility of each step to a different Thread. By maintaining a collection of partially completed objects, the outputs of one Thread become the inputs of the next Thread. You might consider this approach to isolate and parallelize one of the steps for some reason, such as managing I/O bottlenecks.



1. Thread #1 parses the old XML file and sets up the initial mutable Builder.
2. Thread #2 listens to some data source for Builders in need of transformation. When one becomes available, this Thread transforms the the record to DDMS 4.1 and commits it as an immutable Resource.
3. Thread #3 listens to some data source for finished Resources. When one becomes available, this Thread saves it to disk.

Although the Threads perform sequentially on a single metacard, they are not directly dependent on each other (assuming that the collection of partially completed objects exists in some sort of publish/subscribe architecture). The different representations of the XML file (Resource and Builder) are shared between Threads, but the unsafe classes (the Builders) are only operated on by a single Thread.

In the advanced approach, you could parallelize Thread #2 responsibilities as long as each copy of Thread #2 was working on a separate Builder instance. Multiple Threads should never manipulate the same Builder instance at the same time.