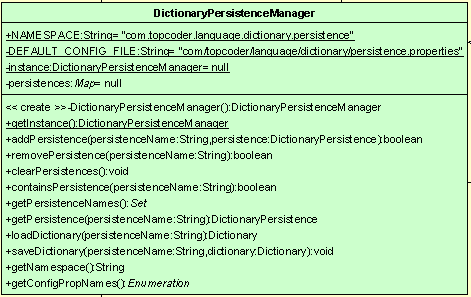
Definition  
"Ensure a class has only one instance, and provide a global point of access to it." [GoF]. This means that you have defined a class that should only be instantiated once during its lifetime, that you require everything to be accessible and that access can occur at any time. The singleton pattern is the most over-used pattern of all the patterns (almost 10% of the components implement this pattern) and is probably not appropriate for 99% of the implementations.   
  
When to use (and when not to!)  
Singletons are generally used in applications to provide access to a single instance of an object that will be shared across an application and will generally include managers, queues, models, pools, connections etc. Whether a given class should have a singleton instance or not is generally contextual to the application and the application's goal. An application designer will know when a class can implement a singleton as they have the full picture of the application in which the singleton will work. Likewise, an application designer knows that only their application will be using the singleton (if run with other applications within the same virtual machine) because the resulting class files will not be shared.   
  
Singletons used within a component have a slightly different viewpoint. A component designer generally does not know the context in which the component will be used and doesn't know if the component will be used by multiple applications within the same virtual machine. If the component designer chooses to use a singleton pattern within their design (especially a singleton-only pattern described below), then the component designer is making assumptions about how the component will be used and is imposing a pattern on the end-using application that may or may not be correct for them. Because of this, TCS strongly discourages the use of singleton patterns in component designs.   
  
Component designers should only use singletons when they must absolutely enforce uniqueness of data. A good example of when to use a singleton pattern would be in working with some 'master' file of some sort (where the uniqueness of information must be enforced). Another good example would be when a 'pool' of objects must be limited in some fashion.   
  
Many of the current component designs use the singleton pattern as either a convenience to the application or to make assumptions on its use - neither reason isvalid. The first reason imposes the pattern on the application and the second reason never works out well in reality. If a component designer feels strongly about either of these, then a variation of the singleton pattern called a singleton wrapper should be implemented (described below). The singleton wrapper variation is acceptable by the review board if the designer has addressed the considerations that are described in the next section.   
  
Considerations  
The foremost consideration is thread safety. A singleton pattern, by definition, will allow global access to it from multiple threads, which requiresthat the internals of a singleton must address threading issues. If the data within a singleton is mutatable, appropriate synchronization (either on the class or via locks) must be properly thought out and documented.   
  
The singleton's class will have a static reference to the singleton instance and will thereby prevent the instance from being garbage collected (under most circumstances). This has significant ramifications for the designer since the resources and memory will not be released. Because the singleton holds strong references to other objects and components, the designer must take into consideration the overhead involved when implementing a singleton (be assured that the review board will take that into consideration when determining whether a singleton pattern is appropriate).   
  
Lastly, the data within the singleton will have global visibility. Since the singleton can be accessed by anything running within the same virtual machine (regardless if it's a sub-system of the application or an entirely different application), the data exposed by the singleton will have global visibility. This may not be appropriate for most components.   
  
Implementation(s)  
**Singleton (only)**  
A typical implementation involves the following three items:

1. A private constructor or a protected constructor if subclasses are possible.
2. A static 'instance' variable of the class type initially set to null (if using lazy instantiation) or to a new instance of the class (if eager instantiation). If using eager instantiation, the variable should be marked 'final' to enforce its immutability.
3. A static getter method (or property for C#) that will create the 'instance' if it's null and return the 'instance' variable. The getter method is generally called 'getInstance' (Java) or the getter property is called 'Instance' (C#). If lazy instantiation is specified, the getter method/property should be marked synchronized or employ some type of locking strategy to ensure that multiple instances are not created from multiple threads (ie to prevent two threads from both checking the null state and both creating an instance at the same time).

Please note that it is recommended that the instance be lazily created and the class marked with the "<< singleton >>" stereotype - but not required.   
  
As stated above, the singleton only type of pattern is strongly discouraged for component design.

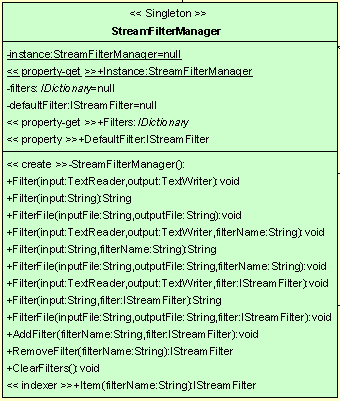
[**Dictionary Persistence Manager**](http://software.topcoder.com/catalog/c_component.jsp?comp=11955835&ver=1) **Component**  
The dictionary persistence manager component provides persistent strategies to persist the contents of a Dictionary (from the Dictionary component).



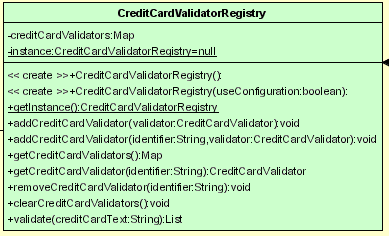
The designers' intent with this manager class is to provide the application with a single instance that can be used to manage all the persistent strategies and provide common convenience function(s) which the application can call. The application can simply make the following call to save a dictionary instance to some persistent datastore:

DictionaryPersistenceManager.getInstance().saveDictionary("custom.dic", dict);

The application does not have to worry about whether the persistence manager has already been created or not (one will be automatically created if not) and in this case, the persistent strategies may be automatically loaded from the configuration manager and do not need to be managed.   
  
This is an example of when a singleton was implemented for convenience to the application and not because the class must be a singleton. The designer assumes that the persistent strategies are global in nature and that the persistent names will not clash if more than one application within the same virtual machine uses this component. While I agree with the designer that this assumption has a fairly low risk, this component probably should have been implemented using a singleton wrapper variation (below) in case the assumption proves incorrect.   
  
The following is a similar singleton but shows the C# implementation of it (from the [Stream Filter](http://software.topcoder.com/catalog/c_component.jsp?comp=11952054&ver=1) component). Notice that the only change is in the implementation of the getter - which becomes a property rather than a method.



**Singleton Wrapper**  
This variation is the more acceptable version of the singleton pattern by the design review board. This variation provides a singleton pattern to the application as a convenience but does not depend on it while specifically addressing the visibility consideration. In other words, it allows the application to choose whether to use the component in a singleton mode, in a non-singleton mode or by both depending on the context of the application. This implementation of this variation simply replaces the private constructor with a public constructor. This allows the application to use a singleton via the getInstance() method and/or to create and pass around multiple instances - allowing the application to partition its information in whatever context is best.   
  
A good example of this would be the [Credit Card Validation](http://software.topcoder.com/catalog/c_component.jsp?comp=10515357&ver=1) component. An application may wish to simply manage credit card validators system-wide or to create multiple validation registries to limit what validators are visible to which system/sub-system.



**Singleton by Key**  
There is a second variation of a singleton that has not been used at TCS yet - a singleton per some defined key (there is some debateas to whether this is still considered a singleton pattern or not). This variation is a cross between the Singleton only and the Singleton wrapper variation. The difference being that the application does not have access to the constructor (and those instances could potentially be created automatically by the configuration manager). This allows the application to define multiple singletons that are globally accessible while alsoallowing the information contained within the singleton to have limited visibility. This variation replaces the instance with a map of instances and takes some key as an argument to the getter method. An example is shown below.   
  
When the application calls getInstance("acme"), the component will:

1. If the instances is null, create a HashMap and assign it to instances
2. If the key does not exist, create an instance of the class and put it in the map under the key.
3. Return the instance associated with the key.

Please note that this variation must be carefully thought out since resources and memory will quickly be consumed by multiple instances of the singletons.