Now, the problem with this approach is that we cannot use static classes as method arguments and this might severely limit our scope. We may need to pass an instance of the `EmailManager` class to some other method, but with static classes we cannot do so.

Also, if we want to have different implementations of `EmailManager` created with different functionality, but with the same behavior (by using interfaces and then having different implementational classes), we will not be able to do that with static classes. Anyways, the singleton pattern is about ensuring that only a single "instance" of our class is always created, and to achieve this goal making our classes static is neither a good approach nor is it feasible in many cases. So let us look at a better approach.

The next strategy would be:

To use the singleton design pattern. The following code shows our class (this code is a stripped-down and simplified version; to have working code it is recommended that you follow the code provided in the code bundle):

```
public sealed class EmailManager
{
         private static EmailManager _manager;
  //Private constructor so that objects cannot be created
         private EmailManager()
         {
         }
         public static EmailManager GetInstance()
         {
            // Use 'Lazy initialization'
           if (_manager == null)
           {
             //ensure thread safety using locks
            lock(typeof(EmailManager)
           {
             _manager = new EmailManager();
           }
           }
         return _manager;
         }
}
```

Now let us understand the code step-by-step:

1. `public sealed class EmailManager`: We have used the `sealed` keyword to make our `EmailManager` class uninheritable. This is not necessary, but there is no use having derived classes as there can be only one instance of this class in memory. Having derived class objects will let us create two or more instances which will be against the singleton's design objective.
2. `private static EmailManager _manager`: Next, we create a variable named `_manager`, which holds a reference to the single instance of our `EmailManager` class. We have used a static modifier because we will be accessing this variable from a static method—`GetEmailManager()`, and static methods can use only static variables.

   ```
    private EmailManager()
               {
         }
   ```

   We need to create a private constructor to make sure that we don't accidentally initialize an object of the `GetEmailManager` class. By only initializing via the static `GetInstance()` method, we should get a single instance of this class.

3.
   ```
   public static EmailManager GetInstance()
   {
                   // Use 'Lazy initialization'
               if (_manager == null)
                  {
       //ensure thread safety using locks
       lock(typeof(EmailManager)
            {
                                 _manager = new EmailManager();
                     }
      }
            return _manager;
   }
   ```

   `GetInstance()` is the static method that we will use from outside this code to get the current reference of the `EmailManager` class. In this method, we are using the lazy loading technique (discussed in Chapter 4) to load the instance on demand. We first check if the current instance is null or not. If it is null, then we create a new one; otherwise we return the existing static instance.

   `lock(typeof(EmailManager)`