### RAII в C#. Локальный Метод Шаблона vs. IDisposable

Одной из ключевой идиом языка С++ является идиома RAII – Recource Acquisition Is Initialization (Захват ресурсов есть инициализация) (\*). Главная ее идея заключается в том, что некоторый ресурс, например, память, дескриптор, сокет и т.п. захватывается в конструкторе и освобождается в деструкторе. А поскольку деструкторы локальных объектов вызываются обязательно, независимо от того, по какой причине управление покидает текущую область видимости, разработчик получаем полуавтоматическое управление ресурсами.

(\*)Сноска: эта идиома очень хорошо расписана в книгах Герба Саттера и Бьёрне Страуструпа. Познакомиться с ней также можно по следующему адресу: http://en.wikipedia.org/wiki/Resource\_acquisition\_is\_initialization.

Такое «автоматическое управление» применяется не только для ресурсов, но и для других целей. Так например, автоматический вызов деструктора используется в многопоточных приложениях для реализации критических секций в коде. Для этого используются классы std::mutex, а также класс std::unique\_lock, который захватывает блокировку в конструкторе и освобождает в деструкторе:

std::mutex m;  
int sh; // разделяемые данные  
// ...  
void f()  
{  
 // ...  
 std::unique\_lock lck(m);  
 // операции над разделяемыми данными:  
 sh += 1;  
}

В управляемых средах, таких как CLR идиома RAII распространена не настолько сильно. Проблемы управления памятью берет на себя сборщик мусора, а за детерминированное освобождение ресурсов отвечают классы, реализующие интерфейс [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx), метод Dispose которого затем вызывается в блоке finally. Но поскольку ручное освобождение ресурсов в блоках finally является неудобным, то в большинстве управляемых языков существует синтаксический сахар: оператор using в C#, "привязка" use (use binding) в F# и т.п.

Интерфейс IDisposable предназначен для освобождения ресурсов. Разработчик класса должен освобождать управляемые ресурсы в методе Dispose, а при наличии неуправляемых ресурсов класс должен содержать еще и финализатор и освобождать их. Когда речь заходит о разработке класса, содержащего управляемые или неуправляемые ресурсы, то решение довольно простое: нужно реализовать интерфейс IDisposable, а в случае неуправляемых ресурсов добавить еще и финализатор и реализовать полноценный Disposable Pattern (см. статью Джо Даффи ["Dispose, Finalization, and Resource Management"](http://joeduffyblog.com/2005/04/08/dg-update-dispose-finalization-and-resource-management/), а также [Programming Stuff: Dispose Pattern](http://sergeyteplyakov.blogspot.com/2011/09/dispose-pattern.html)).

**СНОСКА**. Разница между управляемыми и неуправляемыми ресурсами  
Существует достаточно формальное определение то

Однако что делать, если мы хотим "автоматизировать" не управление ресурсами, а упростить работу с критическими секциями и классом [ReaderWriterLockSlim](http://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim(v=vs.110).aspx), аналогично тому, как это сделано с std::unique\_lock в С++?

Одним из вариантов решения является создание нескольких методов расширения, возвращающих IDisposable объект, конструктор которого захватывает блокировку, а метод Dispose ее освобождает:

public static class DisposeBasedExtensions  
{  
 public static IDisposable UseReadLock(  
 this ReaderWriterLockSlim rwLock)  
 {  
 Contract.Requires(rwLock != null);  
   
 rwLock.EnterReadLock();  
 return new LambdaBasedWrapper(rwLock.ExitReadLock);  
 }  
   
 public static IDisposable UseWriteLock(  
 this ReaderWriterLockSlim rwLock)  
 {  
 Contract.Requires(rwLock != null);  
   
 rwLock.EnterWriteLock();  
 return new LambdaBasedWrapper(rwLock.ExitWriteLock);  
 }  
   
   
 private class LambdaBasedWrapper : IDisposable  
 {  
 private readonly Action \_releaseAction;  
   
 public LambdaBasedWrapper(Action releaseAction)  
 {  
 \_releaseAction = releaseAction;  
 }  
   
 public void Dispose()  
 {  
 \_releaseAction();  
 }  
 }  
}

Теперь при наличии общего ресурса (например, Dictionary<int,int> \_dictionary) и блокировки (ReaderWriterLockSlim \_rwLock) можно будет использовать методы расширения следующим образом:

// Используем блокировку на запись  
using (\_rwLock.UseWriteLock())  
{  
 \_dictionary[42]++;  
}  
   
// Используем блокировку на чтение  
int value;  
using (\_rwLock.UseReadLock())  
{  
 value = \_dictionary[1];  
}

С одной стороны такой подход полностью оправдан, поскольку он удобен и безопасен с точки зрения исключений. С другой стороны, тот же [Эрик Липперт](http://ericlippert.com/)в аннотированной спецификации языка C# ( ["](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) [The](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) [C](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) [#](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) [Programming](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) [Language](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765) ["](http://www.amazon.com/Programming-Language-Covering-Microsoft-Development/dp/0321741765)by Anders Hejlsberg) предостерегает от подобного использования Disposable-объектов.

8.13 Оператор using.  
**Эрик Липперт**. В данной спецификации явно сказано, что суть оператора using заключается в том, чтобы гарантировать захват и своевременное *освобождение ресурса*. Обычно, это означает существование некоторого неуправляемого ресурса, полученного от операционной системы, такие как дескриптор файла. Очень полезно завершить использование ресурсов как можно раньше; другая программа сможет прочитать файл после того, как вы завершите с ним работу. **Но я не рекомендую использовать оператор using для обеспечения инвариантов приложения**. Например, иногда можно встретить следующий код:

using(newTemporarilyStopReportingErrors() AttemptSomething();

Здесь TemporarilyStopReportingErrors – это тип, который в качестве побочного эффекта отключает уведомление об ошибках, а метод Dispose восстанавливает исходное состояние. Я считаю, что эта (довольно распространенная) практика является примером неправильного использования оператора using; побочные эффекты приложения не являются ресурсами, и глобальные побочные эффекты в конструкторе и методе Dispose не кажутся плохой идеей. В этом случае я бы предпочел использовать простой блок try/finally.

Но поскольку использовать каждый раз try/finally не очень удобно, то вместо реализации IDisposable-оболочки, мы можем написать простой вспомогательный метод на основе рассмотренного ранее "локального Шаблонного Метода":

public static void WithReadLock(this ReaderWriterLockSlim rwLock, Action action)  
{  
 try  
 {  
 rwLock.EnterReadLock();  
 action();  
 }  
 finally  
 {  
 rwLock.ExitReadLock();  
 }  
}

И теперь, при наличии все той же блокировки (\_rwLock) и разделяемого ресурса (\_ dictionary), мы сможем использовать данные методы расширения для потокобезопасной работы:

// Используем блокирвку на запись  
\_rwLock.WithWriteLock(  
 () =>  
 {  
 \_dictionary[42]++;  
 });  
   
// Используем блокировку на чтение  
int value;  
\_rwLock.WithReadLock(  
 () =>  
 {  
 value = \_dictionary[1];  
 });

**ПРИМЕЧАНИЕ**  
Реализация методов для захваты блокировки на запись или "обновляемой" (upgradeable) блокировок, аналогичны.

### Так что же выбрать?

У каждого из рассмотренных подходов есть свои особенности. Прежде всего, ни один из них не дотягивает до возможностей языка С++, в котором идиома RAII может применяться не только на уровне методов, но и на уровне полей классов. Так, ни один из этих подходов не будет работать, если мы захотим создать класс, который содержит два поля, управляющих разными ресурсами подобным образом.

Но даже в рамках одного метода у этих подходов есть свои достоинства и недостатки.

Так, хотя Эрик Липперт не рекомендует использовать\_Disposable\_-объекты и блок\_using\_не для освобождения ресурсов, такой подход действительно активно используется, причем не только сторонними разработчиками, но и разработчиками Microsoft. Достаточно вспомнить сценарии использования классов [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope(v=vs.110).aspx)или библиотеку [Microsoft](http://msdn.microsoft.com/en-us/library/hh549175.aspx) [Fakes](http://msdn.microsoft.com/en-us/library/hh549175.aspx), в которой используется этот же подход для «подделки» статических и невиртуальных методов с помощью Shims:

// Shim-ы могут использоваться только внутри ShimsContext  
using (ShimsContext.Create())  
{  
 // "Подделываем" DateTime.Now для возврата нужной даты:  
 System.Fakes.ShimDateTime.NowGet =  
 () => new DateTime(2001, 1, 1);  
   
 Assert.That(DateTime.Now.Date.Year, Is.EqualTo(2001));  
 Assert.That(DateTime.Now.Date.Day, Is.EqualTo(1));  
 Assert.That(DateTime.Now.Date.Day, Is.EqualTo(1));  
}

Но несмотря на распространенность этого подхода, все же он может сбивать с толку. Когда мы делаем оболочку ради побочных эффектов у читателя кода может сложиться впечатление, что мы что-то знаем и у нашей оболочки действительно появились управляемые ресурсы.

С другой стороны, подход на основе "локального Шаблонного Метода" тоже далек от идеала: ведь в этом случае мы создаем анонимные методы, обилие которых вполне может серьезно повлиять на его стоимость во время исполнения. Методы расширения, возвращающие Disposable-объекты тоже реализованы с помощью анонимных методов, но в случае проблем мы всегда сможем легко от них отказаться.

Поэтому если речь идет о критическом к производительности коде, то простой код на основе try/finally будет оптимальным решением. Если же это не performancecritical код, то тут есть выбор: "локальный Шаблонный Метод" или\_IDisposable\_-оболочка. И здесь все больше зависит от вашего вкуса и того, насколько конкретная реализация упростит читаемость кода.

## Реализация Dispose паттерна на основе Шаблонного Метода

Многие наверняка замечали, что классический паттерн для управления ресурсами (a.k.a Dispose Pattern) на платформе .NET является весьма тяжеловесным. Для "незапечатанных" классов его полноценная реализация подразумевает наличие виртуального метода Dispose( bool), который должен отвечать за освобождение управляемых и неуправляемых ресурсов в зависимости от переданного аргумента:

public class ComplexResourceHolder : IDisposable  
{  
 // Буфер из неуправляемого кода (неуправляемый ресурс)  
 private IntPtr \_buffer;  
 // Дескриптор события ОС (управляемый ресурс)  
 private SafeHandle \_handle;  
   
 public ComplexResourceHolder()  
 {  
 // Захватываем ресурсы  
 \_buffer = AllocateBuffer();  
 \_handle = new SafeWaitHandle(IntPtr.Zero, true);  
 }  
   
 protected virtual void Dispose(bool disposing)  
 {  
 // Неуправляемые ресурсы освобождаются в любом случае  
 ReleaseBuffer(\_buffer);  
   
 // Вызываем из метода Dispose, освобождаем управляемые ресурсы  
 if (disposing)  
 {  
 if (\_handle != null)  
 \_handle.Dispose();  
 }  
 }  
   
 ~ComplexResourceHolder()  
 {  
 Dispose(false);  
 }  
   
 public void Dispose()  
 {  
 Dispose(true);  
 GC.SuppressFinalize(this);  
 }  
 // Методы AllocateBuffer и ReleaseBuffer   
}

У этого паттерна есть несколько особенностей. Данный паттерн в чистом виде является избыточным для 99.999% случаев просто потому, что мы редко работаем с неуправляемыми ресурсами напрямую, и практически никогда не используем в одном классе управляемые и неуправляемые ресурсы. Значительно проще обернуть неуправляемый ресурс, такой как NativeHandle в управляемый ресурс и уже использовать два управляемых ресурса в одном классе.

Именно поэтому в "Framework Design Guidelines" предусмотрена упрощенная версия этого паттерна, который называется Basic Dispose Pattern (раздел 9.4.1), которая не содержит финализатора.

Хотя стандарты кодирования настаивают на том, чтобы метод Dispose был невиртуальным, к сожалению, даже в базовой версии паттерна предлагается использовать виртуальный метод Dispose(bool) и освобождать ресурсы только тогда, когда аргумент равен true (т.е. метод вызван из метода Dispose).

Было бы значительно проще вместо одного метода Dispose(bool) воспользоваться полноценным шаблонным методом и явно разбить процесс очистки ресурсов на две составляющие: очистку неуправляемых ресурсов с помощью метода DisposeNativeResources, и очистку управляемых ресурсов DisposeManagedResources.

public class ProperComplexResourceHolder : IDisposable  
{  
 // Поля и конструктор класса аналогичны   
   
 protected virtual void DisposeNativeResources()  
 {  
 ReleaseBuffer(\_buffer);  
 }  
   
 protected virtual void DisposeManagedResources()  
 {  
 if (\_handle != null)  
 \_handle.Dispose();  
 }  
   
 ~ProperComplexResourceHolder()  
 {  
 DisposeNativeResources();  
 }  
   
 public void Dispose()  
 {  
 DisposeNativeResources();  
 DisposeManagedResources();  
 GC.SuppressFinalize(this);  
 }  
   
 // Методы AllocateBuffer и ReleaseBuffer  
}

Да, метод Dispose стал немного сложнее, но при разработке повторно используемого кода больше внимания нужно уделить не количеству кода в базовом классе, а простоту реализации наследников. Тем не менее, теперь любому наследнику будет однозначно понятно, что можно делать в переопределенном методе, а что нет.

**ПРИМЕЧАНИЕ**  
Подробнее о Dispose паттерне, разнице между управляемыми и неуправляемыми ресурсами можно почитать в статье "Dispose Pattern" - http://sergeyteplyakov.blogspot.com/2011/09/dispose-pattern.html. Самым полным описанием этого паттерна является статья Джо Даффи (JoeDuffy) "DG Update: Dispoise, Finalization, and Resource Management - http://joeduffyblog.com/2005/04/08/dg-update-dispose-finalization-and-resource-management/.

## SafeHandle и Метод Шаблона

Несмотря на то, что предложенная версия паттерна освобождения ресурсов не является общепринятой, существуют классы .NET Framework, которые используют полноценный Шаблонный Метод для управления ресурсами. Одним из таких представителей является класс [SafeHandle](http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.safehandle(v=vs.110).aspx).

Так, помимо виртуального метода Dispose(bool) в классе SafeHandle определен абстрактный метод ReleaseHandle, предназначенный непосредственно для освобождения неуправляемого дескриптора и абстрактное свойство IsValid, которое должно сказать, валиден ли текущий дескриптор.

В результате, если вы захотите реализовать свой управляемый дескриптор, то сделать это можно путем создания наследника класса SafeHandle и путем переопределения "недостающих шагов алгоритма Шаблонного Метода":

// Класс HativeHelper в коде отсутствует  
class CustomSafeHandler : SafeHandle  
{  
 private readonly IntPtr \_nativeHandle = NativeHelper.InvalidHandler;  
   
 [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]  
 public CustomSafeHandler() : base(NativeHelper.InvalidHandler, true)  
 {  
 \_nativeHandle = NativeHelper.AcquireHandle();  
 }  
   
 public override bool IsInvalid  
 {  
 [SecurityCritical]  
 get { return \_nativeHandle == NativeHelper.InvalidHandler; }  
 }  
   
 protected override bool ReleaseHandle()  
 {  
 NativeHelper.ReleaseHandle(DangerousGetHandle());  
 return true;  
 }  
}

В результате, метод ReleaseHandle будет вызван и при вызове метод Dispose, и при вызове финализатора, но лишь в том случае, если текущий дескриптор будет валидным (свойство IsInvalid вернет false).

Хотя предложенный в начале статьи вариант паттерна освобождения ресурсов не является общепринятым, его легко можно использовать в своих проектах после согласования со всей командой разработчиков. Но даже если вы не сможете убедить всех в правильности этого подхода, вы можете использовать Шаблонный Метод для управления ресурсами в иерархии классов аналогично тому, как это сделано в классе SafeHandle и его наследниках.

### Дополнительныессылки

* Dispose, Finalization, and Resource Managementby Joe Duffy
* Programming Stuff: Dispose Pattern
* Блок Эрика Липперта
* The C# Programming Language by Anders Hejlsberg at al
* Об идиоме RAII и классе ReaderWriterLockSlim
* Гарантии безопасности исключений