# Паттерн Синглтон (Singleton)

*Все, что вы хотели знать о синглтонах с точки зрения дизайна и реализации, но боялись спросить.*

**Назначение**: гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Другими словами:** Синглтон эмулирует глобальные переменные в объектно-ориентированных языках программирования.

## Мотивация

Практически в любом приложении возникает необходимость в "глобальных переменных" или объектах с ограниченным числом экземпляров. Даже в таком простом приложении, как импорт логов может возникнуть необходимость в логировании. И самый простой способ решить эту задачу - создать глобальный объект, который будет доступен из любой точки приложения.

По своему определению, Синглтон гарантирует, что у некоторого класса есть лишь один экземпляр. В некоторых случаях, анализ предметной области строго требует, чтобы класс существовал лишь в одном экземпляре. Однако на практике, использование паттерна Синглтон обычно используется для обеспечения доступа к некоторому ресурсу, который требуется разным частям приложения.

## Классическая диаграмма паттерна Синглтон

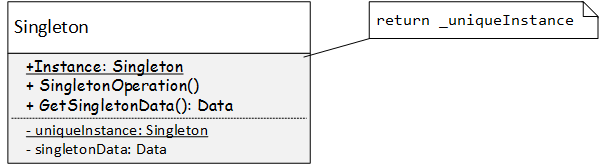


Рисунок 1.1 - Диаграмма классической реализации паттерна Синглтон

## Варианты реализации в .NET

В оригинальном описании паттерна Синглтон бандой четырех на его реализацию не накладывались никакие ограничения, однако на практике (в частности, для платформы .NET) любая реализация должна отвечать двум требованиям: (1) возможность доступа к синглтону в многопоточной среде и (2) «ленивость» (\*) создания.

(\*) Сноска: "Ленивость" создания подразумевает, что экземпляр потенциально дорогостоящего класса будет создан лишь перед первым использованием.

Потокобезопасность является необходимым свойством, поскольку представить себе реальное однопоточное .NET приложение довольно сложно. Ленивость же является, скорее, желательным свойством реализации.

**ПРИМЕЧАНИЕ**  
Все приведенные ниже реализации потокобезопасны с точки зрения количества экземпляров, когда при первом одновременном доступе к синглтону из разных потоков мы не получим лишних экземпляров. Для обеспечения же **потокобезопасного использования** сам объект синглтона (т.е. все его экземплярные методы) должен быть потокобезопасным. Это одна из причин, почему настоятельно не рекомендуется использовать синглтоны с изменяемым состоянием, потокобезопаность которых обеспечить становится сложнее.

### 1. Реализация на основе Lazy of T

public sealed class LazySingleton  
{  
 private static readonly Lazy<LazySingleton> \_instance =   
 new Lazy<LazySingleton>(() => new LazySingleton());  
   
 LazySingleton() {}  
   
 public static LazySingleton Instance { get { return \_instance.Value; } }  
}

Листинг 1.1 - Реализация на основе Lazy<T>

Это самая простая реализация, отвечающая исходным требованиям (потокобезопасности и ленивости). Она настолько проста, что устраняет необходимость в обобщенных решениях, типа Singleton<T>, поскольку требует минимального количества усилий.

**Достоинства**

* Простота + потокобезопаность + ленивость!

**Недостатки**

* Доступна только в .NET 4.0+

### 2. Блокировка с двойной проверкой

public sealed class DoubleCheckedLock  
{  
 // Поле должно быть volatile!  
 private static volatile DoubleCheckedLock \_instance;  
 private static readonly object \_syncRoot = new object();  
   
 DoubleCheckedLock()  
 {}  
   
 public static DoubleCheckedLock Instance  
 {  
 get  
 {  
 if (\_instance == null)  
 {  
 lock (\_syncRoot)  
 {  
 if (\_instance == null)  
 {  
 \_instance = new DoubleCheckedLock();  
 }  
 }  
 }  
 return \_instance;  
 }  
 }  
}

Листинг 1.2 - Реализация на основе блокировки с двойной проверкой

Эта реализация также отвечает критериям потокобезопасности и ленивости, но она существенно сложнее предыдущей реализации. Для того чтобы решение было корректным с точки зрения многопоточности необходимо, чтобы поле \_instance было помечено ключевым словом volatile! Без этого, вполне возможна ситуация, когда другой поток, обращающийся к свойству Instance получит доступ к частично валидному экземпляру синглтона, конструктор которого еще не завершен.

Причина такого поведения в следующем. Создание экземпляра с помощью \_instance = new DoubleCheckedLock(); не является атомарной операцией, а состоит (упрощенно) из следующих этапов:

* выделение памяти в управляемой куче;
* конструирование объекта по указанному адресу (вызов конструктора);
* инициализация поля \_instance.

Если поле \_instance не помечено ключевым словом volatile, то компилятор имеет право изменить порядок этих операций. И если он поменяет 2-й и 3-й шаги местами для одного потока, то другой поток вполне сможет «увидеть», что поле \_instance уже проинициализировано и начнет использовать еще не сконструированный объект (\*) (рис. 1.2).

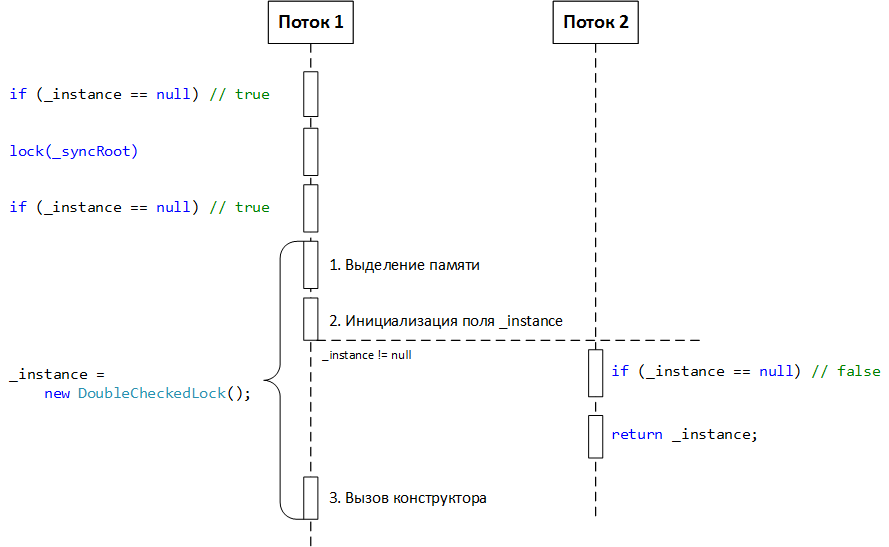


Рисунок 1.2 – Конкурентный порядок конструирования экземпляра

(\*) Сноска: Подробнее о порядке конструирования объектов можно почитать в разделе “Publication via Volatile Field” статьи Игоря Островского "The C# Memory Model in Theory and Practice" MSDN Magazine, Декабрь 2012.

**Достоинства**

* потокобезопасность + ленивость;
* распространенность решения;
* доступно под любой версией .NET Framework.

**Недостатки**

* сложность + потенциальная хрупкость;
* большой объем кода;

### 3. На основе инициализатора статического поля

#### 3.1. «Почти» ленивая реализация

public sealed class FieldInitSingleton  
{  
 // Как вариант, можно перенести инициализацию синглтона прямо   
 // в статический конструктор  
 private static readonly FieldInitSingleton \_instance =   
 new FieldInitSingleton();  
   
 FieldInitSingleton() {}  
   
 // Добавление явного статического конструктора говорит компилятору  
 // не помечать тип атрибутом beforefieldinit  
 static FieldInitSingleton() {}  
   
 public static FieldInitSingleton Instance { get { return \_instance; } }  
}

Листинг 1.3 - Реализация на основе статического поля

#### 3.2. Полностью ленивая реализация

public sealed class LazyFieldInitSingleton  
{  
 private LazyFieldInitSingleton() {}  
  
 public static LazyFieldInitSingleton Instance   
 {   
 get { return SingletonHolder.\_instance; }   
 }  
   
 // Именно вложенный класс делает реализацию полностью ленивой  
 private static class SingletonHolder  
 {  
 public static readonly LazyFieldInitSingleton \_instance =   
 new LazyFieldInitSingleton();  
  
 // Пустой статический конструктор уже не нужен, если мы будем  
 // обращаться к полю \_instance лишь из свойства Instance   
 // класса LazyFieldSingleton  
 }  
}

Листинг 1.4 - Реализация на основе внутреннего класса

#### 3.3. Использование открытого статического поля

В некоторых случаях вместо свойства можно воспользоваться неизменяемым статическим полем (readonly field):

class FieldBasedSingleton  
{  
 public static readonly FieldBasedSingleton Instance =   
 new FieldBasedSingleton();  
}

Листинг 1.5 - Реализация на основе статического поля

У всех представленных выше реализаций есть несколько важных особенностей:

1. **Обработка исключений.** Поскольку инициализация синглтона происходит в статическом конструкторе, то в случае генерации исключения, все клиенты получат его «завернутым» в TypeInitializationException. И в отличие от предыдущих реализаций, попытка инициализации синглтона будет лишь одна.
2. **Время создания синглтона**. Если «забыть» пустой статический конструктор, то время инициализации синглтона станет недетерминированным. Если у типа не определен статический конструктор явно, то компилятор помечает тип атрибутом beforeFieldInit, что позволит вызвать сгенерированный статический конструктор отложенным (“relaxed”) образом, задолго до первого обращения к синглтону.

Так, в следующем примере (при запуске в релизе и без подключенного отладчика), синглтон будет проинициализирован еще до вызова метода Main, даже если условие выполняться во время исполнения не будет.

static void Main(string[] args)  
{  
 Console.WriteLine("Starting Main...");  
 if (args.Length == 1)  
 {  
 var s = SingletonWithoutStaticCtor.Instance;  
 }  
 Console.ReadLine();  
}

Листинг 1.6 - Пример инициализации синглтона до вызова метода

**ПРИМЕЧАНИЕ**  
Причина такого поведения заключается в том, что это дает среде исполнения дополнительные возможности по оптимизации. Более детальное описание тонкостей работы статических конструкторов, а также проблемы, к которым это может привести выходит за рамки данной книги. Об этих особенностях можно почитать в моих статьях, ссылки на которые приведены в конце главы.

1. **Ленивость**. Еще одна проблема реализации без вложенного класса в том, что эта реализация не полностью ленива. Инициализация такого синглтона происходит во время вызова статического конструктора, который может быть вызван не только при использовании синглтона, но и при обращении к статическому члену этого класса.

**Достоинства**

* относительная простота реализации;

**Недостатки**

* особенности генерации исключений;
* возможные проблемы с ленивостью (без вложенного класса);
* проблемы со временем инициализации при отсутствии статического конструктора.

### Какую реализацию выбрать?

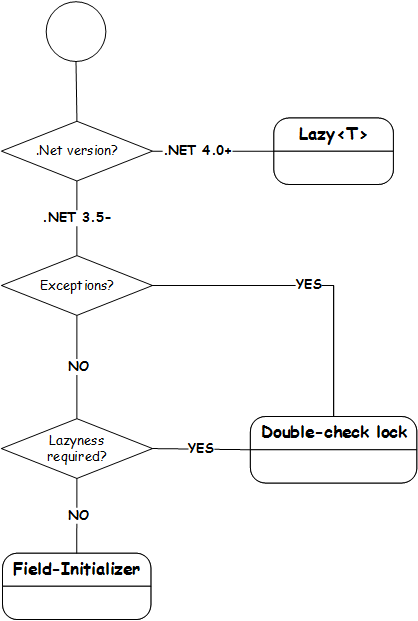


Рисунок 1.3 - Выбор реализации паттерна Синглтон

## Обсуждение паттерна Синглтон

На вопрос «Какие паттерны проектирования вы знаете?» семь человек из десяти первым назовут синглтон, двое назовут его вторым, после фабрики, а оставшийся скажет, что не знает никаких паттернов проектирования. Это самый обсуждаемый и, наверное, самый коварный паттерн проектирования, у которого есть масса особенностей реализации, как с технической точки зрения, так и с точки зрения дизайна.

### Singleton vs. Ambient Context

В оригинальном описании паттерна Синглтон бандой четырех дается ряд особенностей реализации, на которые не всегда обращают внимание. Одна из них звучит так:

Синглтон допускает уточнение операций и представления. От класса Singleton можно порождать подклассы и приложение легко можно сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время исполнения.

Главными недостатками синглтонов считается невозможность юнит-тестирования классов, которые пользуются услугами синглтона, и низкая гибкость. Но если воспользоваться этой оговоркой и дать возможность приложению устанавливать нужный экземпляр синглтона, то многие недостатки уйдут сами собой. По сути, использование данного аспекта приводит к вариации синглтона, называемой Ambient Context (\*).

(\*) Сноска: Ambient Context очень хорошо описан в книге Марка Сиимана "The Dependency Injection in .NET". Я также описывал этот паттерн в статье "Инверсия зависимостей на практике" в разделе "Ambient Context" - http://sergeyteplyakov.blogspot.com/2013/01/blog-post.html#AmbientContext.

Основная суть такой модификации состоит в том, что статическое свойство Instance вместо возврата конкретного класса, возвращает экземпляр абстрактного класса или интерфейса. Также появляется setter свойства, который позволяет установить нужный экземпляр Синглтона при старте приложения, во время смены «контекста» или во время инициализации юнит-тестов. В результате, решение будет более гибким и тестируемым, чего так не хватает классической реализации паттерна.

public interface ILogger  
{  
 void Write();  
}  
  
internal class DefaultLogger : ILogger  
{  
 public void Write() {}  
}  
  
public class GlobalLogger  
{  
 private static ILogger \_logger = new DefaultLogger();  
 // Классы этой сборки (или друзья) смогут задать нужный экземпляр логера  
 public static ILogger Logger  
 {  
 get { return \_logger; }  
 internal set { \_logger = value; }  
 }  
}

Листинг 1.7 - Пример паттерна Ambient Context

Иногда достаточно, чтобы клиенты синглтона могли сконфигурировать и установить глобальное значение, даже если оно и не представлено интерфейсом или абстрактным классом.

Вместо интерфейса, свойство Instance может возвращать экземпляр абстрактного класса или даже экземпляр незапечатанного (non-sealed) конкретного класса, поведение которого может быть переопределено клиентами. В этом случае конструктор класса SingletonBase должен быть не закрытым, как обычно, а защищенным.

Синглтон решает две задачи: гарантирует наличие одного экземпляра класса и обеспечивает глобальную точку доступа. Данный вариант паттерна синглтон не гарантирует наличие одного экземпляра, а обеспечивает лишь глобальную точку доступа к некоторой зависимости. Ambient Context хорошо подходит для использования зависимости (обычно инфраструктурной) разными слоями приложения, делая эти слои менее зависимыми друг от друга. При этом, определенная гибкость обеспечивается за счет возможности установить глобальное состояние при старте приложения.

### Singleton vs. Static Class

Альтернативой паттерну синглтон в объектно-ориентированном мире является использование класса с исключительно статическими членами. Синглтон явно обладает большей гибкостью, но статическими функциями проще пользоваться. Как выбрать между этими двумя подходами?

Можно предложить следующее эмпирическое правило: при отсутствии состояния и при наличии небольшого числа операций статические методы являются более подходящим решением. Если же глобальный объект обладает состоянием, то реализация на основе паттерна Синглтон будет проще.

Существует компромиссное решение: статический класс с небольшим набором методов может выполнять роль фасада (\*) над реализацией на основе Синглтона. ThreadPool.QueueUserWorkItem является хорошим примером такого подхода.

(\*) Сноска: паттерн Фасад будет рассмотрен в третьей части книги.

### Особенности и недостатки

Синглтон – это самый критикуемый паттерн, описанный "бандой четырех", главный недостаток которого кроется в его определении: *синглтон гарантирует, что у класса есть только один экземпляр, и* ***предоставляет к нему глобальную точку доступа***. Это определение состоит из двух частей, и каждая из них несет в себе потенциальную опасность.

Проблема #1: **«Синглтон гарантирует, что у класса есть только один экземпляр…»**

И хотя в классическом описании паттерна говорится, что Синглтон прячет от пользователя количество экземпляров и мы всегда сможем добавить создание еще нескольких экземпляров, на практике сделать это оказывается сложно. Поскольку приложение завязано на определенное статическое свойство (например, Instance), то попытка добавить еще один экземпляр путем добавления нового статического свойства (AnotherInstance?) будет выглядеть нелепо, а сделать параметризованный метод, типа GetInstance(name) поломает весь существующий код.

На самом деле, бизнес-логика очень редко накладывает жесткие ограничения на количество экземпляров класса. Обычно это наши с вами уловки и попытки оправдать ошибки дизайна: обычно легче связать несколько кусков системы с помощью синглтонов, вместо того, чтобы изменить дизайн и передать классам лишь нужные зависимости.

Очень часто в приложении используется несколько синглтонов, которые используют друг друга по цепочке (рис. 1.4):

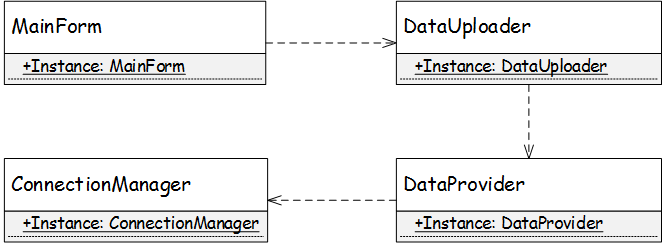


Рисунок 1.4 - Типичная диаграмма приложения, использующего несколько Синглтонов

Данный дизайн легко упростить, заменив его композицией и созданием корневого объекта (DataUploader) в корне приложения - в классе Program.

Проблема #2: **«… и представляет глобальную точку доступа.»**

Главная же проблема паттерна Синглтон заключается в том, что синглтон по своему определению является глобальной переменной, со всеми ее недостатками.

* **Необходимость конструктора по умолчанию**. В большинстве случаев создание экземпляра синглтона происходит ленивым образом, а это значит классу синглтона требуется конструктор по умолчанию. Это, в свою очередь, приводит к тому, что синглтону нельзя передать требуемые зависимости и он будет использовать другие глобальные объекты. В результате легко прийти к дизайну приложения, состоящего из набора глобальных объектов.
* **Неявные зависимости**. Самый простой способ определить сложность класса - проанализировать список полей и аргументов конструктора. Если полей много и/или конструктор принимает слишком большое число аргументов (больше 4-х?!), то это четко говорит, что класс сложный и с его дизайном что-то не так. Но что, если класс не содержит полей и не принимает никаких зависимостей через конструктор, но использует несколько синглтонов? Тогда, чтобы понять его сложность придется проанализировать все закрытые методы.
* **Состояние**. Синглтон с изменчивым состоянием является источником очень коварных ошибок. Внесение изменений в одной части системы может изменить работу произвольного числа модулей, у которых были определенные предположения относительно состояния синглтона. Синглтон может обладать невидимым состоянием, например, кешированием, но полноценной изменяемости нужно избегать всеми силами.

## Применимость. Паттерн или антипаттерн?

Уже не для кого не секрет, что у синглтона достаточное количество недостатков, чтобы считать его не столько паттерном, сколько антипаттерном. Бездумное и бесконтрольное его использование однозначно приведет к проблемам сопровождения, но это не значит, что у него нет сферы применения.

* **Синглтон без видимого состояния**. Нет ничего смертельного в использовании синглтона, через который можно получить доступ к стабильной справочной информации или некоторым утилитам.
* **Настраиваемый контекст**. Аналогично, нет ничего смертельного в протаскивании инфраструктурных зависимостей в виде Ambient Context, т.е. использовать синглтон, возвращающий абстрактный класс или интерфейс, который можно установить в начале приложения или при инициализации юнит-теста.
* **Минимальная область использования**. Ограничьте использование синглтона минимальным числом классов/модулей. Чем меньше у синглтона прямых пользователей, тем легче будет от него избавиться и перейти на более продуманную модель управления зависимостями. Помните, что чем больше у классов пользователей, тем сложнее его изменить. Если уж вы вынуждены использовать синглтон, возвращающий бизнес-объект, то пусть лишь несколько высокоуровневых классов-медиаторов используют синглтоны напрямую и передают его экземпляр в качестве зависимостей классам более низкого уровня.
* **Сделайте использование синглтона явным**. Если передать зависимость через аргументы конструктора не удается, то сделайте использование синглтона явным. Вместо обращения к синглтону из нескольких методов, сделайте статическую переменную и проинициализируйте ее экземпляром синглтона:

// Внутри класса, который использует синглтон  
// Теперь тот факт, что мы используем синглтон становится явным  
private static IRepository \_repository = Repository.Instance;

Листинг 1.8 - Предпочтительное использование синглтонов

Так, по крайней мере, всем будет очевидно, что с дизайном что-то не так и нужно его менять.

## Примеры синглтонов в .NET Framework

**1. Классические синглтоны**

Большинство классических реализаций паттерна Синглтон в составе .NET Framework являются внутренними типами: System.ServiceModel.Dispatcher.PeerValidationBehavior.Instance, System.Net.NetworkingPerfCounters.Instance, System.Threading.TimerQueue.Instance etc. Довольно редкими примерами использования синглтонов в открытой части библиотек являются SystemClock.Instance из библиотеки NodaTime и SqlClientFactory.Instance в .NET Framework.

**2. Фасады в виде статических методов** Примерами такого подхода могут служить ThreadPool.QueueUserWorkItem, log4net.LogManager.GetLogger etc.

**3. Примеры конфигурируемых синглтонов (a.k.a. Ambient Context)** А вот примеров облегченного синглтона на удивление много: AppDomain.CurrentDomain, Thread.CurrentThread, SynchronizationContext.Current, TaskSceduler.Defult, Form.ActiveForm, HttpContext.Current, OperationContext.Current etc.

Глобальный контекст также часто применяется и в других библиотеках: ASP.NET MVC: ControllerBuilder.Current; Entity Framework: Database.SetInitializer.

Приведенные примеры показывают, что синглтоны в чистом виде в .NET Framework, и вообще в библиотеках, практически не применяются; они присутствуют в виде деталей реализации, но очень редко «торчат наружу» в виде открытых классов.

Это вполне логично, поскольку чистыми синглтонами тяжело пользоваться и они делают дизайн чрезмерно жестким. Поэтому классические синглтоны обычно прячутся за фасадными классами (типа ThreadPool.QueueUserWorkItem) или же используется изменяемая версия синглтона (в виде Ambient Context) для протаскивания некоторых зависимостей через разные уровни фреймворка или приложения.

## Дополнительные ссылки

* Jon Skeet "Implementing the Singleton Pattern in C#" - http://csharpindepth.com/articles/general/singleton.aspx
* Programming Stuff: "О синглтонах и статических конструкторах" - http://sergeyteplyakov.blogspot.com/2011/08/blog-post.html
* Programming Stuff: "О времени вызова статических конструкторов" - http://sergeyteplyakov.blogspot.com/2013/07/blog-post.html
* Programming Stuff: Цикл статей об управлении зависимостями - http://sergeyteplyakov.blogspot.com/2013/10/articles.html#dependency\_management.
* Игорь Островский, "The C# Memory Model in Theory and Practice" MSDN Magazine, Декабрь 2012