# Creational Patterns

Jsou návrhové ustálené vzory, které se používají při vytváření nebo získávání nových objektů. Každý z nich nabízí možnosti zpřehlednění nebo zefektivnění kódu. Mezi tyto vzory patří např:

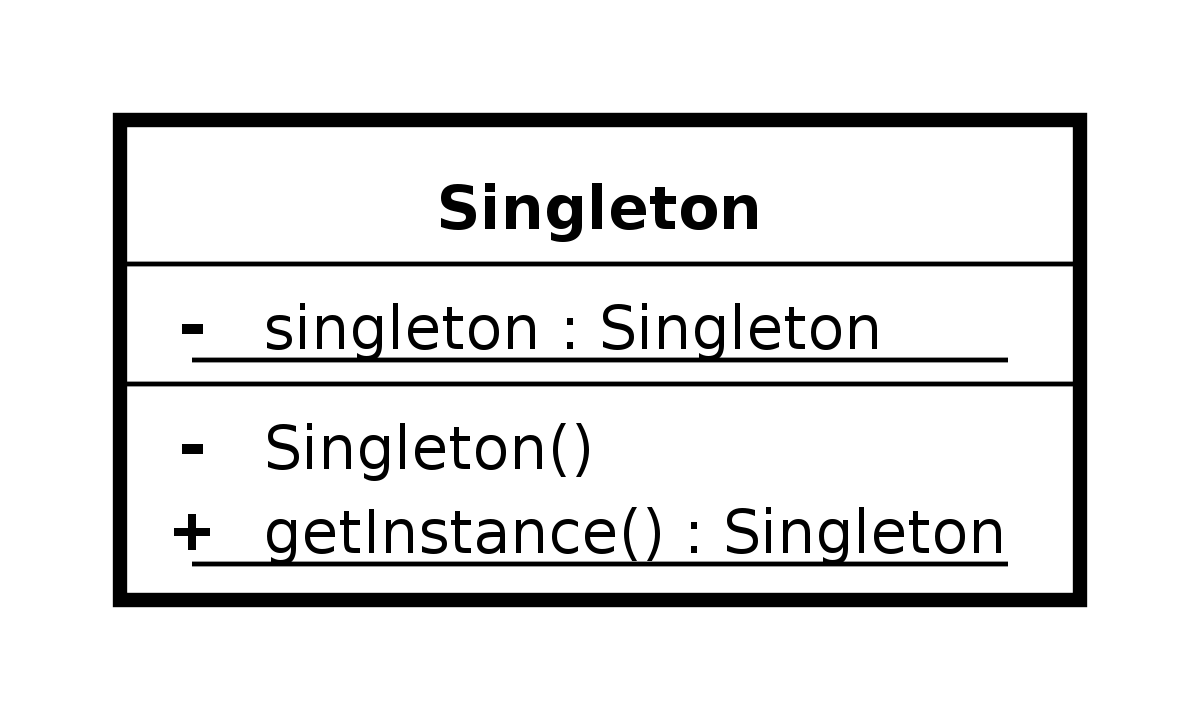
* Singleton
* Object Pool
* Factory Metoda

# Singleton

Návrhový vzor, který nám umožňuje vytvořit pouze jedinou instanci dané třídy.

Používáme ho např. v manažeru připojení k databázi nebo v místech, kde chceme globální správu pouze jednou entitou. Návrhový vzor singleton je doporučeno spíše nepoužívat.

Jeden z důležitých prvků singletonu je nastavení přístupových práv konstruktoru na private, jelikož nechceme, aby jakákoliv jiná třída měla přístup ke konstruktoru a mohla vytvářet vlastní instance. Singleton v sobě obsahuje field Instance, který odpovídá datovému typu třídy, kterou měníme v singleton. Dále máme metodu GetInstance, která napřed zkontroluje, zdá field instance není null. Pokud není null, vrací referenci z fieldu. Pokud je null, pokusí se vytvořit nebo jinak získat novou instanci, dosadí jí do fieldu instance a vrátí.



Příklad možného použití v Unity: Když máme canvas, který je navázaný na určitou třídu, která ovládá veškeré jeho vlastnosti. Tato ovládací třída může být singleton. Díky tomu můžeme k UI přistupovat odkudkoliv ve světě, kde existuje objekt, který má na sobě skript tohoto ovladače UI. Pokud ve světě není, metoda getInstance vrátí null.

Příklad v Unity scriptu:

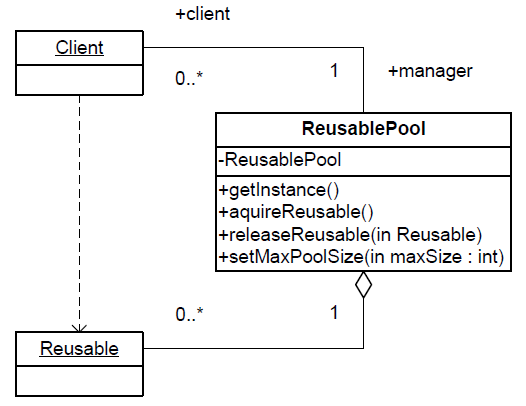
|  |
| --- |
| public class UIController : MonoBehaviour {  private bool Initialized = false;     **private** **static** UIController manager;  **private** Dictionary<**int**, UIElement> WindowResolver = **new** Dictionary<**int**, UIElement>();  // Start is called before the first frame update  **void** **Start**()  {  **if** (!Initialized)  {  Initialise();  }  }   **public** **static** UIController **GetInstance**()  {  **if**(manager != **null**)  {  **return** manager;  }  **return** **new** UIController();  }   **private** **UIController**()  {  WindowResolver = **new** Dictionary<**int**, UIElement>();  manager = **this**;  }   DialogueReader.GetInstance());   **foreach** (**var** v **in** WindowResolver)  {  Debug.Log(v.Key);  }  Initialized = **true**;  } } |

# Object pool

Object pool je návrhový vzor, který používáme v situacích kdy je vytvoření nové instance velmi náročné. Dovoluje vytvořit maximálně námi nastavené množství objektů, které dále využíváme. Když nějaký prvek z poolu vezmeme, není dostupný dokud ho znovu nevrátíme.

V podstatě taková banka na objekty.

Struktura poolu:



Pooly samotné jsou navržené podle vzoru Singleton. To znamená že existuje pouze jedna instance poolu se kterou pracujeme. V ní je také uloženo těch několik málo objektů, ke kterým přistupujeme. Všechny vypůjčené objekty musí být pak vráceny.

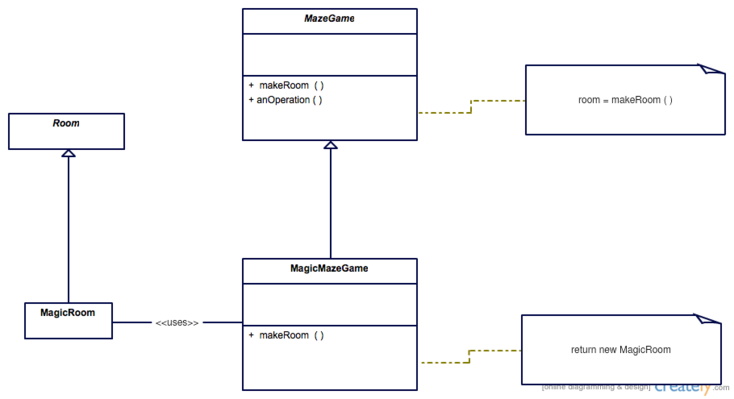
Object pool tak obsahuje metody na

* Získání instance
* Získání objektu z poolu
* Uvolnění objektu poolu

|  |
| --- |
| **namespace** **DesignPattern.Objectpool**  {  // The PooledObject class is the type that is expensive or slow to instantiate,  // or that has limited availability, so is to be held in the object pool.  **public** **class** **PooledObject**  {  DateTime \_createdAt = DateTime.Now;    **public** DateTime CreatedAt  {  **get** { **return** \_createdAt; }  }    **public** **string** TempData { **get**; **set**; }  }   **public** **static** **class** **Pool**  {  **private** **static** List<PooledObject> \_available = **new** List<PooledObject>();  **private** **static** List<PooledObject> \_inUse = **new** List<PooledObject>();    **public** **static** PooledObject **GetObject**()  {  **lock**(\_available)  {  **if** (\_available.Count != 0)  {  PooledObject po = \_available[0];  \_inUse.Add(po);  \_available.RemoveAt(0);  **return** po;  }  **else**  {  PooledObject po = **new** PooledObject();  \_inUse.Add(po);  **return** po;  }  }  }    **public** **static** **void** **ReleaseObject**(PooledObject po)  {  CleanUp(po);    **lock** (\_available)  {  \_available.Add(po);  \_inUse.Remove(po);  }  }    **private** **static** **void** **CleanUp**(PooledObject po)  {  po.TempData = **null**;  }  } } |

# Factory Method

Factory používáme, když máme několik tříd, implementujících stejný interface nebo , ze kterých můžeme udělat instance a chceme nechat rozhodnout program, která z těchto tříd bude nejlepší. Factory metodu také používáme když chceme vytvářet instance z těchto tříd na základě určité business logiky nebo jiných podmínek.



Factory metoda se dá rozdělit do dvou částí, Factory a Třídy které jsou tvořeny.

Factory jsou odděděné od jedné hlavní factory(nebo interface) které vytváří třídy (např. typu Room) s ohledem na logiku, kterou vložíme do metody (např. makeRoom (nebo jakoukoliv metodu, která tvoří samotný objekt)). Tato tvořící metoda musí mít stejný datový typ jako factory nadřazená této factory. Jako návratový typ se dává Třída, od které jsou odděděné produkty, které chceme tvořit.

|  |
| --- |
| **public** **interface** **IPerson** {  **string** **GetName**(); }  **public** **class** **Villager** : **IPerson** {  **public** **string** **GetName**()  {  **return** "Village Person";  } }  **public** **class** **CityPerson** : **IPerson** {  **public** **string** **GetName**()  {  **return** "City Person";  } }  **public** **enum** PersonType {  Rural,  Urban }  **///** **<summary>** **///** Implementation of Factory - Used to create objects **///** **</summary>** **public** **class** **Factory** {  **public** IPerson **GetPerson**(PersonType type)  {  **switch** (type)  {  **case** PersonType.Rural:  **return** **new** Villager();  **case** PersonType.Urban:  **return** **new** CityPerson();  **default**:  **throw** **new** NotSupportedException();  }  } } |