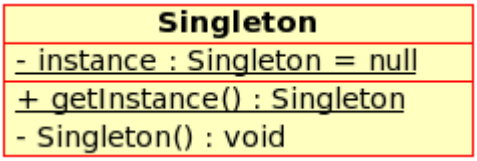
Singleton

Singleton е шаблон за дизайн от типа създаващ шаблон (creational design pattern), при който инстанцирането на класа се ограничава само до един обект. Това е необходимо, когато имаме нужда от единствен обект, който да координира работата на цялото приложение. Много автори считат, че този шаблон е по-скоро анти-шаблон, който прекалено често се използва, даже на места, където ограниченията за единствена инстанция всъщност не са необходими и създава ненужен глобален обхват в приложението. Мнозина други твърдят, че концепцията на Singleton може да бъде имплементирана с простото използване на подходящ статичен клас (естествено в езиците за програмиране, които го допускат).



Най-честа употреба

- Много други шаблони като **Abstract Factory, Builder, Prototype** използват Singleton обекти в тяхната имплементация.
- Обектите на **Facade** шаблона често са Singleton поради изискването за единствен такъв обект.
- **State** обектите често са от тип Singleton.
- Singleton-ите често се предпочитат пред глобалните променливи поради следните причини:
    - Те не задръстват глобалното пространство (namespace) с излишни променливи.
    - Позволяват „късно“ инициализиране и заделяне на памет за разлика от глобалните променливи, които винаги консумират необходимите им ресурси в началото.

Имплементация

Имплементацията на шаблона Singleton трябва отговаря на условията за една единствена инстанция (един обект) и глобален достъп до нея. Необходими са механизми за достъпване на класа без възможност за създаване на обект и поддържането на едни и същи стойности на неговите членове във всички други обекти. Това се постига чрез създаването на подходящ клас с метод, който създава нова инстанция само ако такава не съществува. В противен случай просто се връща референцията към съществуващия обект. За да е сигурно, че нов обект не може да бъде създаден по някакъв друг начин, конструктурът на Singleton класа трябва да бъде private.

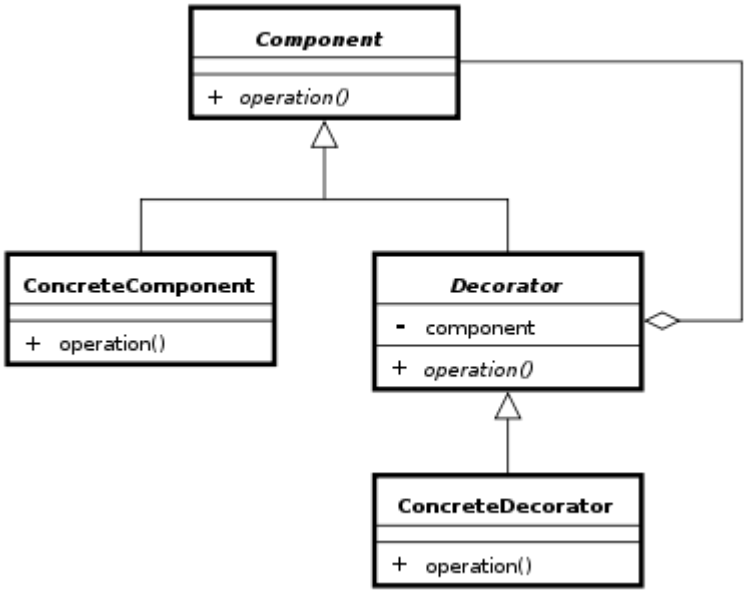
Има съществена разлика между Singleton класа и обикновения статичен клас – въпреки, че singleton може да бъде имплементиран като статичен, той също може да бъде с „късна“ инициализация, което не изисква памет и ресурси до неговата първа употреба. Друга важна особеност е, че статичните класове не могат да наследяват интерфейси, т.е. ако даден статичен клас трябва да „реализира договор“, предоставен от интерфейс, то очевидно този клас трабва да бъде Singleton.

Използването на този шаблон при многонишково програмиране трябва да бъде извършвано много внимателно. Това се обуславя от факта, че няколко паралелни нишки трябва да достъпват един и същи обект, като в даден момент само една от тях може да има права за достъп до него. Това се реализира чрез заключване на Singleton инстанцията и всички останали нишки чакат за освобождаване на ресурса. Много е важно всяка нишка да освобождава обекта своевременно.

Пример за шаблона Singleton е даден в приложения VisualStudio 2012 solution.

Decorator

Шаблонът **decorator** е от типа структурни шаблони за дизайн (structural design pattern), който позволява да към индивидуален обект да бъде добавяно определно поведение по статичен или динамичен път без да бъдат променяни характеристиките и поведението на другите обекти от същия клас.



Този шаблон може да бъде използван за разширяване (декориране) функционалността на определен обект статично (или динамично при определени условия), независимо от другите инстанции на същия клас, като разбира се върху класа на въпросния обект се извършат определни предварителни подготвителни действия във фазата на програмирането му. Прави се чрез изграждането на нов обвиващ клас decorator. Това обвиване може да бъде направено по следната схема:

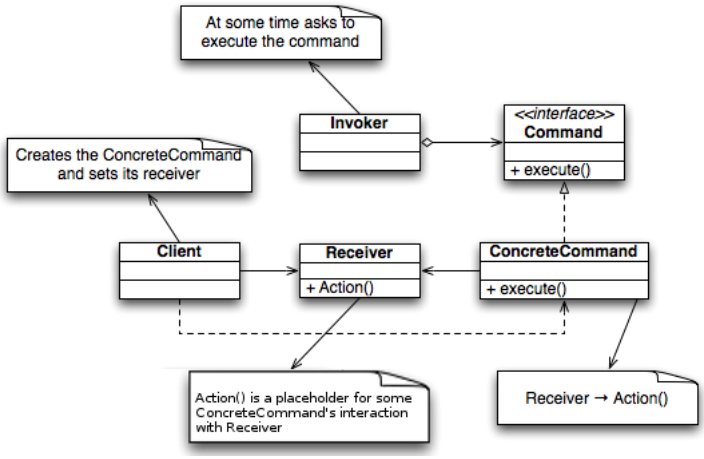
1. Оригиналният клас "Component" се затваря в клас "Decorator" (виж UML диаграмата);
2. В този клас Decorator се добавя поле от тип Component;
3. Обектът Component се предава на конструктора на Decorator за инициализиране на вътрешния екземпляр на Component;
4. Всички методи на класа Component се пренасочват в класа Decorator към вътрешния Component обект;
5. В класа ConcreteDecorator се override-ват всички методи на Component, чийто поведение трябва да бъде модифицирано.

По този начин множество decorator-и могат да бъдат закачени един върху друг, като всеки път се добавя нова и нова функционалност към override-натите методи.

Трябва да се отбележи, че декораторите и оригиналният обект поделят общ набор от функционалности. В диаграмата по-горе методът "operation()" е достъпен и чрез декораторите и чрез декорирания обект.
Пример за шаблона Decorator е даден в приложения VisualStudio 2012 solution.

Command

В ООП шаблонът **Command** е от типа поведенчески шаблон (behavioral design pattern), в който един обект се използва за да представи и енкапсулира цялата необходима информация за извикване на даден метод на по-късен етап. Тази информация включва името на метода, обекта на метода и входните му параметри.



Четири компонента се асоциират винаги към шаблона command - *command*, *receiver*, *invoker* и *client*. Конкретната команда (обект *concrete command)* е свързана с определен приемник (обект *receiver)* и извиква негов метод по начин, който е специфичен за класа receiver. Приемникът извършва необходимата работа за изпълнението на командата. Отделно от това конкретната команда (обект command) е изпратена към извикващия обект *invoker* и чрез която се извиква командата, а в *invoker* също се отчитат и резултатите и действията по изпълнение на командата (това впоследствие може да се използва за възстановяване на предишно състояние). Към обект *invoker* може да се изпрати всеки един обект *command*. Самият обект *invoker* и наборът от команди (множество обекти *command*) се съдържат в обекта *client*. Клиентът взема решение коя команда да изпълни и по кое време. За да се изпълни някоя команда клиентът изпраща неин обект към обекта *invoker*.

С други думи класът *client* разполага само с определени команди (обекти command) и не се интересува и не знае кой е реалния клас-изпълнител (receiver). По същия начин класът *receiver* получава от *invoker* обекти *command*, които изпълнява и не знае и не се интересува кой ги е изпратил.

Използването на обекти-команди прави много лесно да се изградят компоненти на по-глобално ниво, които трябва да делегират, подреждат и изпълняват извиквания към определени методи, като по време на избора кой метод и кога да се извиква, не се интересуват те към кой клас принадлежат и какви параметри са им необходими. Използването на извикващ обект (*invoker*) прави много лесно съхраняването на история на извикванията, реализирането на различни състояния на командите, като практически клиентът не е нужно да се грижи за точната им имплементация.

Обектите Command се използват главно за реализиране на следните функционалности - Бутони и елементи от менюта в графичния потребителски интерфейс, Запис на макроси, Преместваем код, Връщане назад в историята (undo) на много нива, Мрежова комуникация, Паралелна обработка, Индикатори за напредъка в извършването на операция (Progress bars), Пулове от нишки (Thread pools), Операции с транзакционен характер, Помощници (Wizards).