

```
^ Maze new
  addRoom: room1;
  addRoom: room2;
  yourself
```

Известные применения

Быть может, впервые паттерн прототип был использован в системе Sketchpad Ивана Сазерленда (Ivan Sutherland) [Sut63]. Первым широко известным применением этого паттерна в объектно-ориентированном языке была система Thing-Lab, в которой пользователи могли сформировать составной объект, а затем превратить его в прототип, поместив в библиотеку повторно используемых объектов [Bor81]. Адель Голдберг и Давид Робсон упоминают прототипы в качестве паттернов в работе [GR83], но Джеймс Коплиен [Cop92] рассматривает этот вопрос гораздо шире. Он описывает связанные с прототипом идиомы языка C++ и приводит много примеров и вариантов.

Etgdb – это оболочка отладчиков на базе ET++, где имеется интерфейс вида point-and-click (укажи и щелкни) для различных командных отладчиков. Для каждого из них есть свой подкласс DebuggerAdaptor. Например, GdbAdaptor настраивает etgdb на синтаксис команд GNU gdb, а SunDbxAdaptor – на отладчик dbx компании Sun. Набор подклассов DebuggerAdaptor не «защит» в etgdb. Вместо этого он получает имя адаптера из переменной среды, ищет в глобальной таблице прототип с указанным именем, а затем его клонирует. Добавить к etgdb новые отладчики можно, связав ядро с подклассом DebuggerAdaptor, разработанным для этого отладчика.

Библиотека приемов взаимодействия в программе Mode Composer хранит прототипы объектов, поддерживающих различные способы интерактивных отношений [Sha90]. Любой созданный с помощью Mode Composer способ взаимодействия можно применить в качестве прототипа, если поместить его в библиотеку. Паттерн прототип позволяет программе поддерживать неограниченное число вариантов отношений.

Пример музыкального редактора, обсуждавшийся в начале этого раздела, основан на каркасе графических редакторов Unidraw [VL90].

Родственные паттерны

В некоторых отношениях прототип и абстрактная фабрика являются конкурентами. Но их используют и совместно. Абстрактная фабрика может хранить набор прототипов, которые клонируются и возвращают изготовленные объекты.

В тех проектах, где активно применяются паттерны компоновщик и декоратор, тоже можно извлечь пользу из прототипа.

Паттерн Singleton

Название и классификация паттерна

Одиночка – паттерн, порождающий объекты.

Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Мотивация

Для некоторых классов важно, чтобы существовал только один экземпляр. Хотя в системе может быть много принтеров, но возможен лишь один спулер. Должны быть только одна файловая система и единственный оконный менеджер. В цифровом фильтре может находиться только один аналого-цифровой преобразователь (АЦП). Бухгалтерская система обслуживает только одну компанию.

Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает инстанцировать класс в нескольких экземплярах.

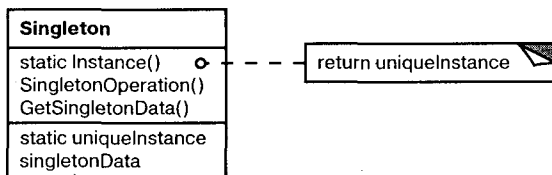
Более удачное решение – сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Это и есть назначение паттерна одиночка.

Применимость

Используйте паттерн одиночка, когда:

- ❑ должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- ❑ единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Структура



Участники

- ❑ **Singleton** – одиночка:
 - определяет операцию `Instance`, которая позволяет клиентам получать доступ к единственному экземпляру. `Instance` – это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция-член в C++;
 - может нести ответственность за создание собственного уникального экземпляра.

Отношения

Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance.

Результаты

У паттерна одиночка есть определенные достоинства:

- ❑ *контролируемый доступ к единственному экземпляру.* Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- ❑ *уменьшение числа имен.* Паттерн одиночка – шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;
- ❑ *допускает уточнение операций и представления.* От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения;
- ❑ *допускает переменное число экземпляров.* Паттерн позволяет вам легко изменить свое решение и разрешить появление более одного экземпляра класса Singleton. Вы можете применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton;
- ❑ *большая гибкость, чем у операций класса.* Еще один способ реализовать функциональность одиночки – использовать операции класса, то есть статические функции-члены в C++ и методы класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких экземпляров класса. Кроме того, статические функции-члены в C++ не могут быть виртуальными, так что их нельзя полиморфно заместить в подклассах.

Реализация

При использовании паттерна одиночка надо рассмотреть следующие вопросы:

- ❑ *гарантирование единственного экземпляра.* Паттерн одиночка устроен так, что тот единственный экземпляр, который имеется у класса, – самый обычный, но больше одного экземпляра создать не удастся. Чаще всего для этого прячут операцию, создающую экземпляры, за операцией класса (то есть за статической функцией-членом или методом класса), которая гарантирует создание не более одного экземпляра. Данная операция имеет доступ к переменной, где хранится уникальный экземпляр, и гарантирует инициализацию переменной этим экземпляром перед возвратом ее клиенту. При таком подходе можно не сомневаться, что одиночка будет создан и инициализирован перед первым использованием.

В C++ операция класса определяется с помощью статической функции-члена Instance класса Singleton. В этом классе есть также статическая

переменная-член `_instance`, которая содержит указатель на уникальный экземпляр.

Класс `Singleton` объявлен следующим образом:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

А реализация такова:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Клиенты осуществляют доступ к одиночке исключительно через функцию-член `Instance`. Переменная `_instance` инициализируется нулем, а статическая функция-член `Instance` возвращает ее значение, инициализируя ее уникальным экземпляром, если в текущий момент оно равно 0. Функция `Instance` использует отложенную инициализацию: возвращаемое ей значение не создается и не хранится вплоть до момента первого обращения.

Обратите внимание, что конструктор защищенный. Клиент, который попытается инстанцировать класс `Singleton` непосредственно, получит ошибку на этапе компиляции. Это дает гарантию, что будет создан только один экземпляр.

Далее, поскольку `_instance` – указатель на объект класса `Singleton`, то функция-член `Instance` может присвоить этой переменной указатель на любой подкласс данного класса. Применение возможности мы увидим в разделе «Пример кода».

О реализации в C++ скажем особо. Недостаточно определить рассматриваемый паттерн как глобальный или статический объект, а затем полагаться на автоматическую инициализацию. Тому есть три причины:

- мы не можем гарантировать, что будет объявлен только один экземпляр статического объекта;
- у нас может не быть достаточно информации для инстанцирования любого одиночки во время статической инициализации. Одиночке могут быть необходимы данные, вычисляемые позже, во время выполнения программы;
- в C++ не определяется порядок вызова конструкторов для глобальных объектов через границы единиц трансляции [ES90]. Это означает, что

между одиночками не может существовать никаких зависимостей. Если они есть, то ошибок не избежать.

Еще один (хотя и не слишком серьезный) недостаток глобальных/статических объектов в том, что приходится создавать всех одиночек, даже если они не используются. Применение статической функции-члена решает эту проблему. В Smalltalk функция, возвращающая уникальный экземпляр, реализуется как метод класса `Singleton`. Чтобы гарантировать единственность экземпляра, следует заместить операцию `new`. Получающийся класс мог бы иметь два метода класса (в них `SoleInstance` – это переменная класса, которая больше нигде не используется):

```
new
    self error: 'не удастся создать новый объект'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

- *порождение подклассов Singleton*. Основной вопрос не столько в том, как определить подкласс, а в том, как сделать, чтобы клиенты могли использовать его единственный экземпляр. По существу, переменная, ссылающаяся на экземпляр одиночки, должна инициализироваться вместе с экземпляром подкласса. Простейший способ добиться этого – определить одиночку, которого нужно применять в операции `Instance` класса `Singleton`. В разделе «Пример кода» показывается, как можно реализовать эту технику с помощью переменных среды.

Другой способ выбора подкласса `Singleton` – вынести реализацию операции `Instance` из родительского класса (например, `MazeFactory`) и поместить ее в подкласс. Это позволит программисту на C++ задать класс одиночки на этапе компоновки (скомпоновав программу с объектным файлом, содержащим другую реализацию), но от клиента одиночка будет по-прежнему скрыт.

Такой подход фиксирует выбор класса одиночки на этапе компоновки, затрудняя тем самым его подмену во время выполнения. Применение условных операторов для выбора подкласса увеличивает гибкость решения, но все равно множество возможных классов `Singleton` остается жестко «защитым» в код. В общем случае ни тот, ни другой подход не обеспечивают достаточной гибкости.

Ее можно добиться за счет использования *реестра одиночек*. Вместо того чтобы задавать множество возможных классов `Singleton` в операции `Instance`, одиночки могут регистрировать себя по имени в некотором всем известном реестре.

Реестр сопоставляет одиночкам строковые имена. Когда операции `Instance` нужен некоторый одиночка, она запрашивает его у реестра по имени. Начинается поиск указанного одиночки, и, если он существует, реестр возвращает его. Такой подход освобождает `Instance` от необходимости «знать» все

возможные классы или экземпляры Singleton. Нужен лишь единый для всех классов Singleton интерфейс, включающий операции с реестром:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Операция Register регистрирует экземпляр класса Singleton под указанным именем. Чтобы не усложнять реестр, мы будем хранить в нем список объектов NameSingletonPair. Каждый такой объект отображает имя на одиночку. Операция Lookup ищет одиночку по имени. Предположим, что имя нужного одиночки передается в переменной среды:

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // пользователь или среда предоставляют это имя на стадии
        // запуска программы

        _instance = Lookup(singletonName);
        // Lookup возвращает 0, если такой одиночка не найден
    }
    return _instance;
}
```

В какой момент классы Singleton регистрируют себя? Одна из возможностей – конструктор. Например, подкласс MySingleton мог бы работать так:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Разумеется, конструктор не будет вызван, пока кто-то не инстанцирует класс, но ведь это та самая проблема, которую паттерн одиночка и пытается разрешить! В C++ ее можно попытаться обойти, определив статический экземпляр класса MySingleton. Например, можно вставить строку

```
static MySingleton theSingleton;
```

в файл, где находится реализация MySingleton.

Теперь класс Singleton не отвечает за создание одиночки. Его основной обязанностью становится обеспечение доступа к объекту-одиночке из

любой части системы. Подход, сводящийся к применению статического объекта, по-прежнему имеет потенциальный недостаток: необходимо создавать экземпляры всех возможных подклассов Singleton, иначе они не будут зарегистрированы.

Пример кода

Предположим, нам надо определить класс MazeFactory для создания лабиринтов, описанный на стр. 99. MazeFactory определяет интерфейс для построения различных частей лабиринта. В подклассах эти операции могут переопределяться, чтобы возвращать экземпляры специализированных классов продуктов, например объекты BombedWall, а не просто Wall.

Существенно здесь то, что приложению Maze нужен лишь один экземпляр фабрики лабиринтов и он должен быть доступен в коде, строящем любую часть лабиринта. Тут-то паттерн одиночка и приходит на помощь. Сделав фабрику MazeFactory одиночкой, мы сможем обеспечить глобальную доступность объекта, представляющего лабиринт, не прибегая к глобальным переменным.

Для простоты предположим, что мы никогда не порожаем подклассов от MazeFactory. (Чуть ниже будет рассмотрен альтернативный подход.) В C++ для того, чтобы превратить фабрику в одиночку, мы добавляем в класс MazeFactory статическую операцию Instance и статический член _instance, в котором будет храниться единственный экземпляр. Нужно также сделать конструктор защищенным, чтобы предотвратить случайное инстанцирование, в результате которого будет создан лишний экземпляр:

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // здесь находится существующий интерфейс
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

Реализация класса такова:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Теперь посмотрим, что случится, когда у MazeFactory есть подклассы и определяется, какой из них использовать. Вид лабиринта мы будем выбирать с помощью переменной среды, поэтому добавим код, который инстанцирует нужный

подкласс MazeFactory в зависимости от значения данной переменной. Лучше всего поместить код в операцию Instance, поскольку она уже и так инстанцирует MazeFactory:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... другие возможные подклассы

        } else { // по умолчанию
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Отметим, что операцию Instance нужно модифицировать при определении каждого нового подкласса MazeFactory. В данном приложении это, может быть, и не проблема, но для абстрактных фабрик, определенных в каркасе, такой подход трудно назвать приемлемым.

Одно из решений – воспользоваться принципом реестра, описанным в разделе «Реализация». Может помочь и динамическое связывание, тогда приложению не нужно будет загружать все неиспользуемые подклассы.

Известные применения

Примером паттерна одиночка в Smalltalk-80 [Par90] является множество изменений кода, представленное классом ChangeSet. Более тонкий пример – это отношение между классами и их *метаклассами*. Метаклассом называется класс класса, каждый метакласс существует в единственном экземпляре. У метакласса нет имени (разве что косвенное, определяемое экземпляром), но он контролирует свой уникальный экземпляр, и создать второй обычно не разрешается.

В библиотеке InterViews для создания пользовательских интерфейсов [ICI+92] – паттерн одиночка применяется для доступа к единственным экземплярам классов Session (сессия) и WidgetKit (набор виджетов). Классом Session определяется главный цикл распределения событий в приложении. Он хранит пользовательские настройки стиля и управляет подключением к одному или нескольким физическим дисплеям. WidgetKit – это абстрактная фабрика для определения внешнего облика интерфейсных виджетов. Операция WidgetKit::instance() определяет конкретный инстанцируемый подкласс WidgetKit на основе переменной среды, которую устанавливает Session. Аналогичная операция в классе Session «выясняет», поддерживаются ли монохромные или цветные дисплеи, и соответственно конфигурирует одиночку Session.

Родственные паттерны

С помощью паттерна одиночка могут быть реализованы многие паттерны. См. описание абстрактной фабрики, строителя и прототипа.

Обсуждение порождающих паттернов

Есть два наиболее распространенных способа параметризовать систему классами создаваемых ей объектов. Первый способ – порождение подклассов от класса, создающего объекты. Он соответствует паттерну **фабричный метод**. Основной недостаток метода: требуется создавать новый подкласс лишь для того, чтобы изменить класс продукта. И таких изменений может быть очень много. Например, если создатель продукта сам создается **фабричным методом**, то придется замещать и создателя тоже.

Другой способ параметризации системы в большей степени основан на композиции объектов. Вы определяете объект, которому известно о классах объектов-продуктов, и делаете его параметром системы. Это ключевой аспект таких паттернов, как **абстрактная фабрика**, **строитель** и **прототип**. Для всех трех характерно создание «фабричного объекта», который изготавливает продукты. В **абстрактной фабрике** фабричный объект производит объекты разных классов. **Фабричный объект строителя** постепенно создает сложный продукт, следуя специальному протоколу. **Фабричный объект прототипа** изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип – это одно и то же, поскольку именно прототип отвечает за возврат продукта.

Рассмотрим каркас графических редакторов, описанный при обсуждении паттерна прототип. Есть несколько способов параметризовать класс `GraphicTool` классом продукта:

- применить паттерн **фабричный метод**. Тогда для каждого подкласса класса `Graphic` в палитре будет создан свой подкласс `GraphicTool`. В классе `GraphicTool` будет присутствовать операция `NewGraphic`, переопределяемая каждым подклассом;
- использовать паттерн **абстрактная фабрика**. Возникнет иерархия классов `GraphicsFactories`, по одной для каждого подкласса `Graphic`. В этом случае каждая фабрика создает только один продукт: `CircleFactory` – окружности `Circle`, `LineFactory` – отрезки `Line` и т.д. `GraphicTool` параметризуется фабрикой для создания подходящих графических объектов;
- применить паттерн **прототип**. Тогда в каждом подклассе `Graphic` будет реализована операция `Clone`, а `GraphicTool` параметризуется прототипом создаваемого графического объекта.

Выбор паттерна зависит от многих факторов. В нашем примере каркаса графических редакторов, на первый взгляд, проще всего воспользоваться **фабричным методом**. Определить новый подкласс `GraphicTool` легко, а экземпляры `GraphicTool` создаются только в момент определения палитры. Основной недостаток такого подхода заключается в комбинаторном росте числа подклассов `GraphicTool`, причем все они почти ничего не делают.