# Применение шаблона проектирования Singleton для класса Controller

### Назначение

Гарантирует существование только одного объекта данного класса и обеспечивает глобальный доступ к этому объекту.

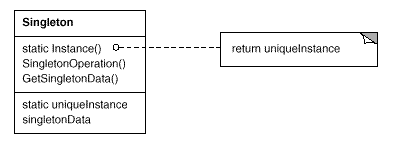
Мотивация

Иногда важно, чтобы существовал ровно один экземпляр какого-либо класса. Система может иметь много принтеров, но должен существовать только один спулер принтера. Должна существовать только одна файловая система и один администратор оконного режима. Цифровой фильтр всегда имеет один A/D преобразователь.

Как можно гарантировать, что класс имеет только один объект и что этот образец является легко доступным? Глобальная переменная делает объект доступным, но не предохраняет Вас от инстанцирования множества объектов.

Лучшее решение состоит в том, чтобы определить класс, непосредственно ответственный за слежение за собственным объектом. Класс может гарантировать, что никакой другой экземпляр не может быть создан (прерывая запросы на создание новых объектов). Этот шаблон - Singleton.

Структура



Составные части

* **Singleton**
  + Определяет операцию Instance, которая позволяет клиентуре обращаться к ее уникальному экземпляру. Instance - операция класса (то есть метод класса в Smalltalk и статический метод в C++).
  + Может быть ответствен за создание его собственного уникального экземпляра.

### Реализация

Класс Controller, координирует работу игры и должен существовать в единственном экземпляре. Для реализации такого класса одиночки идеально подходит паттерн Одиночка (Singleton).

В классе Controller добавляем статическую переменную, содержащую ссылку на единственный экземпляр класса:

static Controller\* \_instance; //!< Единственный объект в программе

Так же добавляем статическую функцию для получения указателя на объект:

//! Если объект уже создан, возвращает ссылку, иначе создает объект

static Controller\* Instance(CMainDlg\* dlg = 0);

Добавляем реализацию и начальную инициализацию переменной:

Controller\* Controller::\_instance = 0;

Controller\* Controller::Instance(CMainDlg\* dlg)

{

if (\_instance == 0)

\_instance = new Controller(dlg);

return \_instance;

}

Далее приведена строчка, где в функции WinMain первый раз запрашивается объект, что приводит к его созданию

Game::Controller\* g\_control = Game::Controller::Instance(dlgMain);

# Применение шаблона проектирования Iterator для класса PetriNet

### Назначение

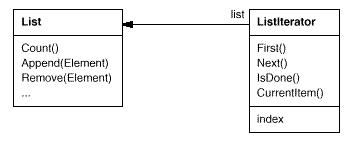
Предоставляет возможность обратиться к элементам агрегата последовательно, без затрагивания его основного представления.

### Мотивация

Составной объект типа «список» должен предоставлять возможность обратиться к своим элементам не затрагивая свою внутреннюю структуру. Кроме того, Вы можете захотеть обойти список различными способами, в зависимости от того, что вы хотите сделать. Но Вы, вероятно, не захотите засорять интерфейс List операциями для различных обходов списка. Кроме того, возможно, вам понадобится не один экземпляр такого итератора.

Шаблон Iterator позволяет Вам разрешить эту задачу. Идея шаблона состоит в том, чтобы выделить ответственность за доступ, и обход списка и поместить ее в объект **iterator**. Класс Iterator определяет интерфейс для доступа к элементам списка. Iterator ответствен за слежение за текущим элементом; то есть он знает, которые элементы уже были посещены.

Например, класс List вот так связан с ListIterator:



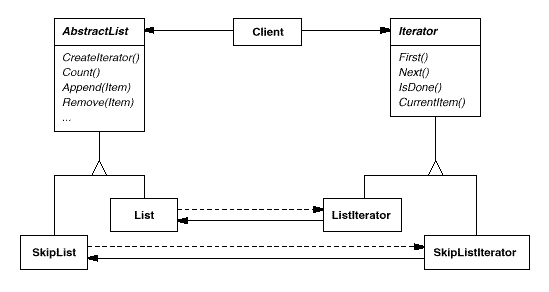
Прежде, чем Вы сможете инстанцировать ListIterator, Вы должны обеспечить его списком, который надо обходить. Как только у вас появился объект ListIterator, Вы можете обращаться к элементам списка последовательно. Операция CurrentItem возвращает текущий элемент в списке, First инициализирует текущий элемент как первый элемент, Next переходит к следующему, и IsDone говорит, прошли ли мы весь список.

Отделение механизма обхода из объекта List позволяет нам определять итераторы для различных целей без перечисления их в интерфейсе List. Например, FilteringListIterator мог бы обеспечивать доступ только к тем элементам, которые удовлетворяют специфическим условиям фильтра.

Обратите внимание, что iterator и список сдвоены, и клиент должен знать, с каким агрегатом он работает. Было бы лучше, если бы мы могли изменять класс агрегат не изменяя клиентский код. Мы можем добиться этого определяя понятие **полиморфный итератор**.

Например, пусть SkipList – реализация List. Skiplist - структура данных с характеристиками, подобными сбалансированному дереву. Мы хотим суметь написать код, который работает для объектов List и SkipList.

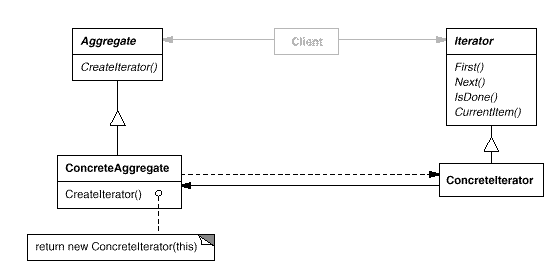
Мы определяем класс AbstractList, который обеспечивает общий интерфейс, управляющий списками. Точно так же мы должны описать Iteratorclass, который определяет общий интерфейс итератора. Затем мы можем определить конкретные подклассы Iterator для различных реализаций списка. В результате, итеративный механизм становится независящим от класса агрегата.



Остаётся ещё одна задача - как создать итератор. Так как мы хотим писать код, независящий от конкретного подкласса List, мы не можем просто инстанцировать определенный класс. Вместо этого, мы делаем объекты List ответственными за создание соответствующего итератора. Для этого необходимо описать операцию CreateIterator, с помощью которой клиент запрашивает объект - итератор.

CreateIterator - пример метода фабрики. Мы используем его здесь, чтобы позволить клиенту запрашивать у объекта списка соответствующий итератор. Фабрика описывает две иерархии классов, первую для списков, вторю для итераторов. CreateIterator метод фабрики "соединяет" эти иерархии.

### Структура



Составные части

* **Iterator**
  + Определяет интерфейс для доступа и обхода элементов.
* **ConcreteIterator**
  + Реализует интерфейс Iterator.
  + Следит за текущей позицией в обходе Agregate.
* **Agregate**
  + Определяет интерфейс для создания объекта Iterator.
* **ConcreteAggregate**
  + Реализует интерфейс Iterator, чтобы возвратить экземпляр соответствующего ConcreteIterator.

### Реализация

В контейнере существует два типа подсущностей: позиции и переходы, причем помимо обхода этих двух сущностей, нам нужен инструментарий для обхода входных позиций каждого перехода и выходных позиций. Для обхода позиций и переходов хорошо бы иметь схожий интерфейс. Тут нам на помощь приходит шаблон проектирования «Итератор».

Далее представлен интерфейс класса для обхода позиций:

//! Итератор для обхода позиций

template <typename MarkingType>

class position\_iterator

{

public:

position\_iterator();

position\_iterator(const position\_iterator& rv);

position\_iterator& operator=(const position\_iterator& rv);

bool end() const;

position\_iterator& operator++();

POSITION position() const;

…

};

У остальных итераторов интерфейс повторяет вышеприведенный за исключением типов возвращаемых значений и имен конструкторов. Для удобства было принято решение не использовать абстрактный класс как интерфейс для всех четырех итераторов.