2 слайд:
1) Итератор - паттерн поведения объектов.
Он предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.
2) Составной объект, скажем список, должен предоставлять способ доступа к своим элементам, не раскрывая их внутреннюю структуру. Более того, иногда требуется обходить список по-разному, в зависимости от решаемой задачи. Но вряд ли вы захотите засорять интерфейс класса List операциями для различных вариантов обхода, даже если все их можно предвидеть заранее. Кроме того, иногда нужно, чтобы в один и тот же момент было определено несколько активных обходов списка.
Все это позволяет сделать паттерн итератор. Основная его идея в том, чтобы за доступ к элементам и способ обхода отвечал не сам список, а отдельный объект — итератор.

3 слайд:
В классе Iterator определен интерфейс для доступа к элементам списка. Объект этого класса отслеживает текущий элемент, то есть он располагает информацией, какие элементы уже посещались.
Прежде чем создавать экземпляр класса Listlterator, необходимо иметь список, подлежащий обходу. С объектом Listlterator вы можете последовательно посетить все элементы списка. Операция CurrentItem возвращает текущий элемент списка, операция First инициализирует текущий элемент первым элементом списка, Next делает текущим следующий элемент, а IsDone проверяет, не оказались ли мы за последним элементом, если да, то обход завершен.
Отделение механизма обхода от объекта List позволяет определять итераторы, реализующие различные стратегии обхода, не перечисляя их в интерфейсе класса List. Например, FilteringListlterator мог бы предоставлять доступ только к тем элементам, которые удовлетворяют условиям фильтрации.
Заметим: между итератором и списком имеется тесная связь, клиент должен иметь информацию, что он обходит именно список, а не какую-то другую агрегированную структуру. Поэтому клиент привязан к конкретному способу агрегирования. Было бы лучше, если бы мы могли изменять класс агрегата, не трогая код клиента. Это можно сделать, обобщив концепцию итератора и рассмотрев полиморфную итерацию.

4 слайд:
Например, предположим, что у нас есть еще класс SkipList, реализующий список. Список с пропусками (skiplist) - это вероятностная структура данных, по характеристикам напоминающая сбалансированное дерево. Нам нужно научиться писать код, способный работать с объектами как класса List, так и класса SkipList. Определим класс AbstractList, в котором объявлен общий интерфейс для манипулирования списками. Еще нам понадобится абстрактный класс Iterator, определяющий общий интерфейс итерации. Затем мы смогли бы определить конкретные подклассы класса Iterator для различных реализаций списка. В результате механизм итерации оказывается не зависящим от конкретных агрегированных классов.

Остается понять, как создается итератор. Поскольку мы хотим написать код, не зависящий от конкретных подклассов List, то нельзя просто инстанцировать конкретный класс. Вместо этого мы поручим самим объектам-спискам создавать для себя подходящие итераторы, вот почему потребуется операция Createlterator, посредством которой клиенты смогут запрашивать объект-итератор.
Createlterator - это пример использования паттерна фабричный метод. В данном случае он служит для того, чтобы клиент мог запросить у объекта-списка подходящий итератор. Применение фабричного метода приводит к появлению двух иерархий классов - одной для списков, другой для итераторов. Фабричный метод Createlterator «связывает» эти две иерархии.

5 слайд :
Теперь рассмотрим общую структуру
Здесь мы видим:
■ Iterator - итератор:
- определяет интерфейс для доступа и обхода элементов;
■ Concretelterator - конкретный итератор:
- реализует интерфейс класса Iterator;
- следит за текущей позицией при обходе агрегата;
■ Aggregate - агрегат:
- определяет интерфейс для создания объекта-итератора;
■ ConcreteAggregate - конкретный агрегат:
- реализует интерфейс создания итератора и возвращает экземпляр подходящего класса Concretelterator.
- у него есть метод Createlterator, посредством которой клиенты смогут запрашивать объект-итератор.

6 слайд:
1) Используйте паттерн итератор:
■ для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
■ для поддержки нескольких активных обходов одного и того же агрегированного объекта;
■ для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).

7 слайд:
Рассмотрим важные вопросы и решения реализации паттерна Iterator:
■ какой участник управляет итерацией. Важнейший вопрос состоит в том, что управляет итерацией: сам итератор или клиент, который им пользуется. Если итерацией управляет клиент, то итератор называется внешним, в противном случае - внутренним. Клиенты, применяющие внешний итератор, должны явно запрашивать у итератора следующий элемент, чтобы двигаться дальше по агрегату. Напротив, в случае внутреннего итератора клиент передает итератору некоторую операцию, а итератор уже сам применяет эту операцию к каждому посещенному во время обхода элементу агрегата. Внешние итераторы обладают большей гибкостью, чем внутренние. Например, сравнить две коллекции на равенство с помощью внешнего итератора очень легко, а с помощью внутреннего - практически невозможно. Но, с другой стороны, внутренние итераторы проще в использовании, поскольку они вместо вас определяют логику обхода;
■ что определяет алгоритм обхода.
Алгоритм обхода можно определить не только в итераторе. Его может определить сам агрегат и использовать итератор только для хранения состояния итерации. Такого рода итератор мы называем курсором, поскольку он всего лишь указывает на текущую позицию в агрегате. Клиент вызывает операцию Next агрегата, передавая ей курсор в качестве аргумента. Операция же Next изменяет состояние курсора.
Если за алгоритм обхода отвечает итератор, то для одного и того же агрегата можно использовать разные алгоритмы итерации, и, кроме того, проще применить один алгоритм к разным агрегатам.
С другой стороны, алгоритму обхода может понадобиться доступ к закрытым переменным агрегата. Если это так, то перенос алгоритма в итератор нарушает инкапсуляцию агрегата;
■ насколько итератор устойчив. Модификация агрегата в то время, как совершается его обход, может оказаться опасной. Если при этом добавляются или удаляются элементы,

то не исключено, что некоторый элемент будет посещен дважды или вообще ни разу. Простое решение - скопировать агрегат и обходить копию, но обычно это слишком дорого.
Поэтому применятся устойчивый итератор (robust). Он гарантирует, что ни вставки, на удаления не помешают обходу, причем достигается это без копирования агрегата. Есть много способов реализации устойчивых итераторов. В большинстве из них итератор регистрируется в агрегате. При вставке или удалении агрегат либо подправляет внутреннее состояние всех созданных им итераторов, либо организует внутреннюю информацию так, чтобы обход выполнялся правильно.
■ дополнительные операции итератора. Минимальный интерфейс класса Iterator состоит из операций First, Next, IsDone и Currentltem. Но могут оказаться полезными и некоторые дополнительные операции. Например, упорядоченные агрегаты могут предоставлять операцию Previous, позиционирующую итератор на предыдущий элемент. Для отсортированных или индексированных коллекций интерес представляет операция SkipTo, которая позиционирует итератор на объект, удовлетворяющий некоторому критерию;
■ итераторы могут иметь привилегированный доступ. Класс Iterator может включать защищенные операции для доступа к важным, но не являющимся открытыми членам агрегата. Подклассы класса Iterator (и только его подклассы) могут воспользоваться этими защищенными операциями для получения привилегированного доступа к агрегату;
■ пустые итераторы. Пустой итератор NullIterator - это вырожденный итератор, полезный при обработке граничных условий. По определению, NullIterator всегда считает, что обход завершен, то есть его операция IsDone неизменно возвращает истину. Применение пустого итератора может упростить обход древовидных структур. В каждой точке обхода мы запрашиваем у текущего элемента итератор для его потомков. Элементы-агрегаты, как обычно, возвращают конкретный итератор. Но листовые элементы возвращают экземпляр NullIterator. Это позволяет реализовать обход всей структуры единообразно.

8 слайд:
У паттерна итератор есть следующие важные особенности:
■ поддерживает различные виды обхода агрегата. Сложные агрегаты можно обходить по-разному. Например, нам нужно обходить дерево в прямом или обратном порядке. Итераторы упрощают изменение алгоритма обхода - достаточно просто заменить один экземпляр итератора другим.
■ итераторы упрощают интерфейс класса Aggregate. Наличие интерфейса для обхода в классе Iterator делает излишним дублирование этого интерфейса в классе Aggregate. Тем самым интерфейс агрегата упрощается;
■ одновременно для данного агрегата может быть активно несколько обходов.