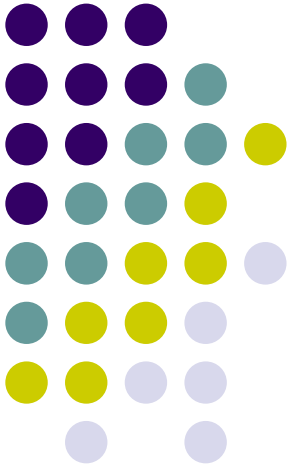
Patrones de Diseño

Patrón de comportamiento *Iterator*



Iterator
Propósito

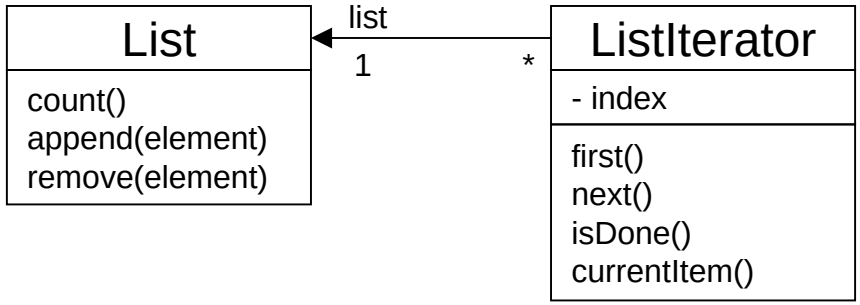
- Proporcionar acceso secuencial a los elementos de un agregado, sin exponer su representación interna

- También conocido como cursor

Iterator
Motivación

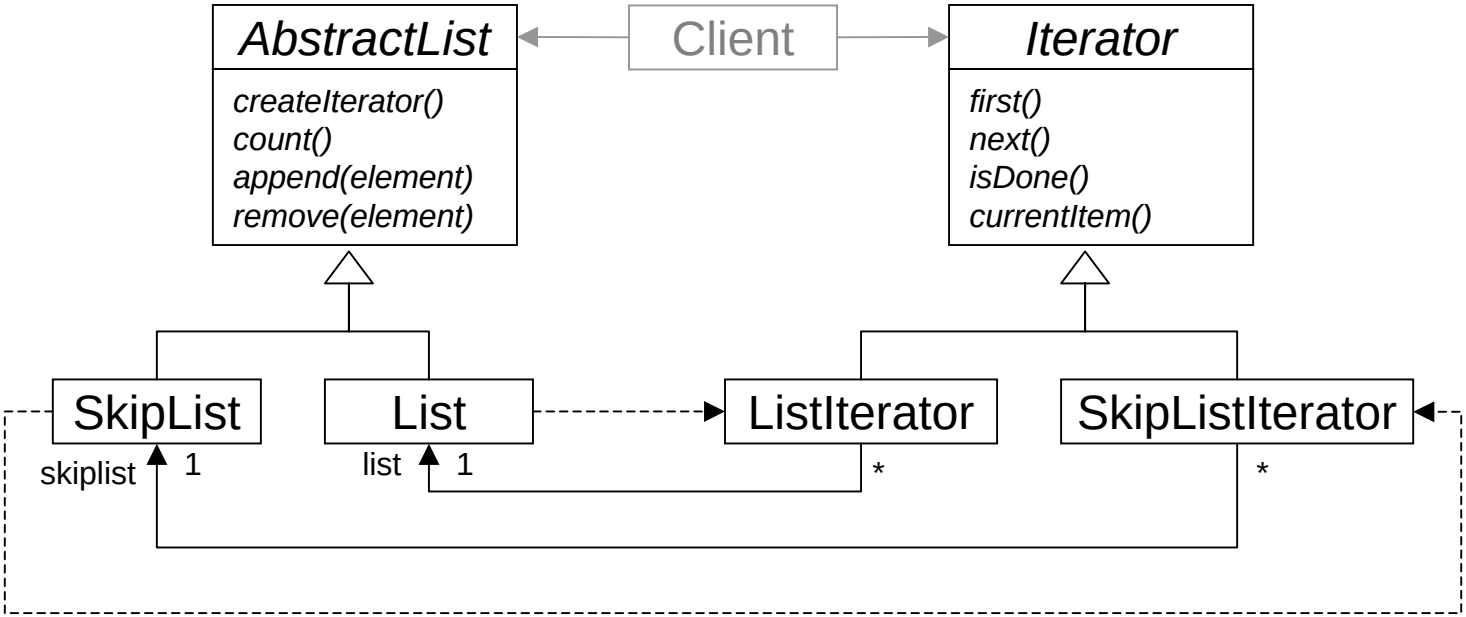
- Ej: Una lista debe proporcionar un medio de navegar por sus datos sin exponer su estructura interna
- Se debe poder atravesar la lista de varias maneras, pero no añadir operaciones a la lista por cada tipo de recorrido
- Se debe poder realizar varios recorridos simultáneamente
- Solución:
  - dar la responsabilidad de recorrer la lista a un objeto iterador

Iterator
Motivación



- Al instanciar *ListIterator* se debe proporcionar la lista
- Una vez instanciado el iterador, se puede acceder a los elementos de la lista
- Ventaja: separar el mecanismo de recorrido del objeto lista permite definir iteradores que implementen distintas estrategias, varios recorridos a la vez
- Inconveniente: el iterador y la lista están acoplados, el cliente sabe que lo que está recorriendo es una lista

Iterator
Motivación

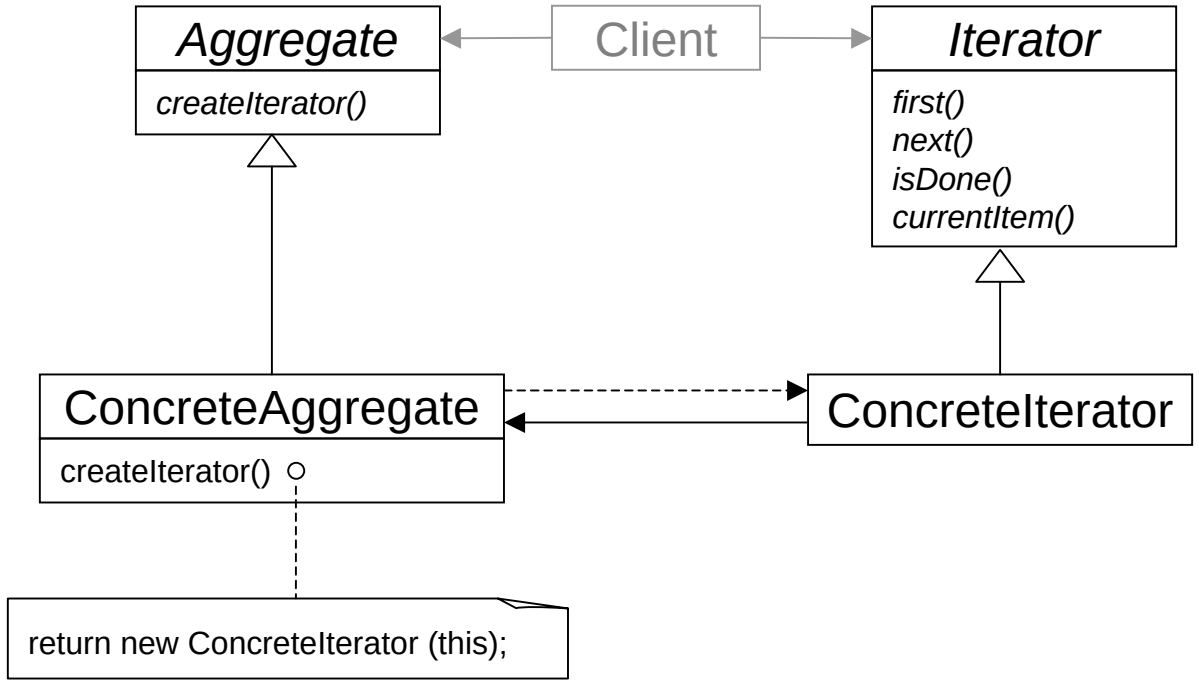


- Generalizar el iterador para que soporte iteración polimórfica
- Se puede cambiar el agregado sin cambiar el código cliente
- Las listas se hacen responsables de crear sus propios iteradores

Iterator
Aplicabilidad

- Usa el patrón *Iterator*:
  - Para acceder al contenido de un agregado sin exponer su representación interna
  - Para permitir varios recorridos sobre un agregado
  - Para proporcionar una interfaz uniforme para recorrer distintos tipos de agregados (esto es, permitir iteración polimórfica)

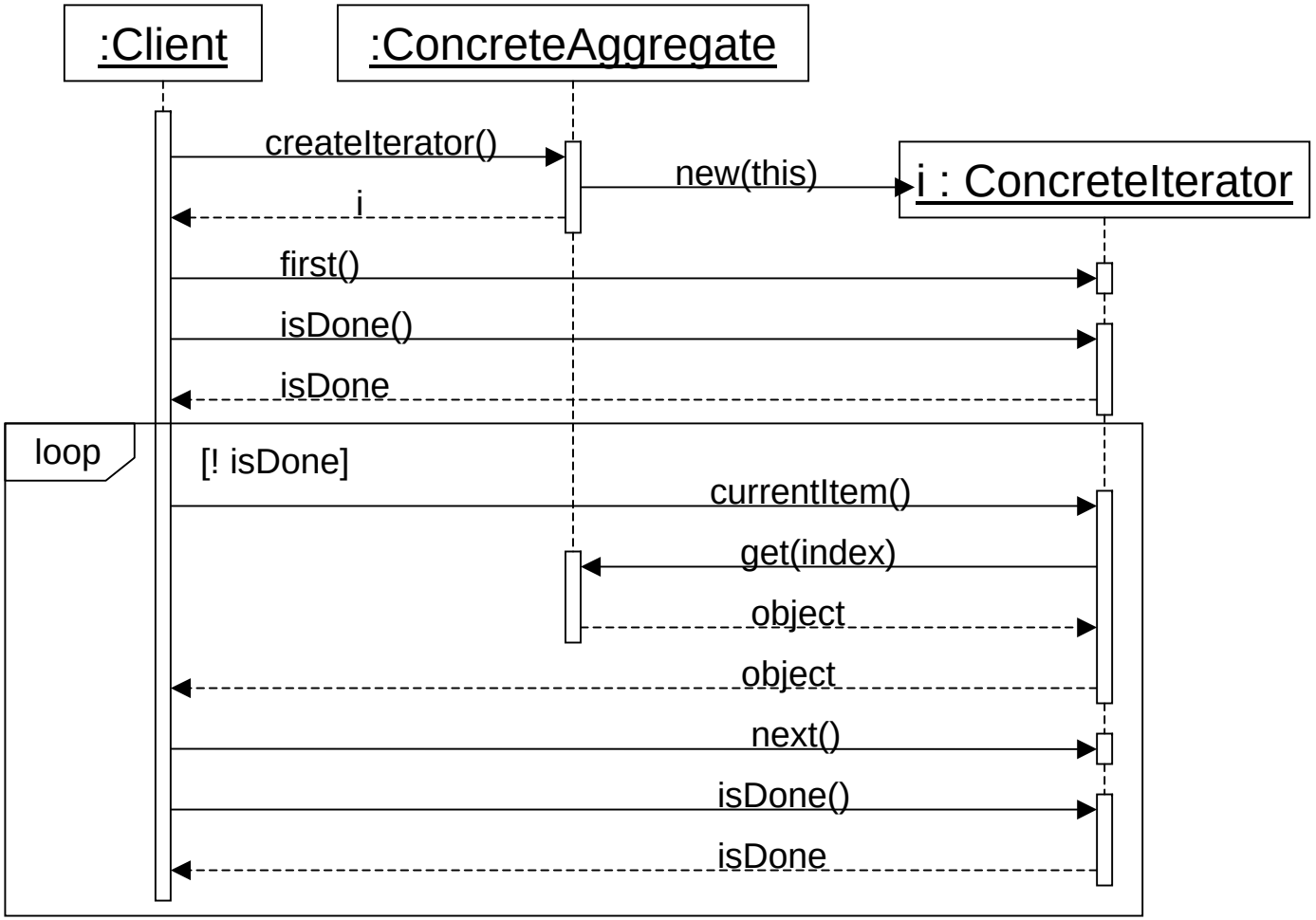
Iterator
Estructura



Iterator
Participantes

- **Iterator**: define interfaz para acceder y recorrer elementos
- **ConcreteIterator**:
  - Implementa la interfaz *Iterator*
  - Mantiene la posición actual en el recorrido del agregado
- **Aggregate***:* define interfaz para crear un objeto *Iterator*
- **ConcreteAggregate***:* implementa una interfaz de creación del *Iterator* para devolver la instancia de *ConcreteIterator* apropiada

Iterator
Colaboraciones



Iterator
Consecuencias

- Permite variaciones en el recorrido de un agregado
  - Para cambiar el algoritmo de recorrido basta cambiar la instancia de *Iterator* concreta
  - Nuevos recorridos mediante nuevas subclases de *Iterator*
- Los iteradores simplifican la interfaz del contenedor
- Puede hacerse más de un recorrido a la vez sobre un mismo agregado

Iterator
Implementación

- ¿Quién controla la iteración?
  - El cliente: iterador externo
  - El iterador: iterador interno. El iterador recibe una función a aplicar sobre los elementos del agregado, y el recorrido es automático
- ¿Quién define el algoritmo de recorrido?
  - El iterador: es más fácil implementar distintos recorridos y reutilizarlos en distintos agregados. Compromete la encapsulación
  - El agregado: el método *next()* está en el agregado y el iterador (llamado cursor) sólo almacena el estado actual de la iteración
- ¿Cómo de robusto es el iterador?
  - Iterador robusto: las inserciones y borrados no interfieren en el recorrido (y se hace sin copiar el agregado)
  - Implementación: registrar cambios del agregado en el iterador

Iterator
Implementación

- Operaciones adicionales en el iterador
  - Ej. operación *previous* para ir al objeto anterior
  - Ej. operación *skipTo* para ir a un objeto que cumpla cierto criterio

- Iteradores nulos *(NullIterators)*
  - Iterador degenerado que ayuda a manejar condiciones límite
  - El método *isDone()* siempre devuelve *true*
  - Útil para estructuras compuestas heterogéneas (*composite*)

Iterator

Código de ejemplo


```
public interface Iterator {
   public void first();
   public void next();
   public boolean isDone();
   public Object currentItem();
}

public interface Aggregate {
   public Iterator createIterator();
   public Object get(int);
   public int count();
}

public class ListIterator {
   private Aggregate a;
   private int current;
   ListIterator (Aggregate a); {
      this.a = a;
      current = 0;
   }
   public void first() { current = 0; }
   public void next() { current++; }
   public boolean isDone() { return current >= a.count(); }
   public Object currentItem() { return a.get(current); }
}
```

Iterator
En java…

- Marco de contenedores de java (*framework collection*)
  - **Aggregate**:
    - Las interfaces `Collection`, `Set`, `SortedSet`, `List`, `Queue` de `java.util`
    - Incluyen método `iterator()`, que devuelve un iterador genérico
  - **ConcreteAggregate**: implementaciones de esas interfaces
    - Set es implementada por las clases `HashTree`, `TreeSet`, `LinkedHashSet`
    - List es implementada por las clases `ArrayList`, `LinkedList`
  - **Iterator**: interfaz `java.util.Iterator`
    - `boolean hasNext()`
    - `Object next()`
    - `void remove()`
  - **ConcreteIterator**: implementaciones concretas de `Iterator`
  - **Ejemplo de cliente**:

```
java.util.Collection c = new java.util.LinkedList();
java.util.Iterator it = c.iterator();
while (it.hasNext()) {
        it.next();
}
```