Design Patterns (Iterator)

Reading: Freeman, Robson, Bates, Sierra, *Head First Design Patterns*, O'Reilly Media, Inc. 2004

Ch 9: Composite and Iterator Patterns

Electronic text available from UW Library Web site
An *iterator* provides a way to iterate (i.e., loop through) the elements of a collection of objects.

- Doesn’t expose the underlying representation of the collection.
- Moves responsibility for access and traversal order into a separate “iterator” object.

**Motivation:**

- Want to apply a function (e.g., `transform()`) to all items in collection.
- Want to be able to have multiple, simultaneous traversals of the collection.
Metaphor: Iterator Pattern

“You sit down at a full dinner table and the host wants to introduce you to all the guests—what order does he use? In some circumstances, he will simply go around the table; but if the Overlord Dictator of the Central Galaxies Concern happens to be sitting somewhere near the middle, perhaps he will prefer to introduce in order of importance. The collection of people has a fixed order, but the iteration strategy is flexible.”

http://mahemoff.com/paper/software/gofMetaphors/
Iterator: The Basics

Collection
- addChild(Child)
-removeChild(Child)
-Operation()
-createIterator()
-size()
-getElem(int)

<<create>>

Iterator
- cursor
-first()
-hasNext()
-next()

Child
Simple Iterator Pattern

- Collection
  - addChild(Child)
  - removeChild(Child)
  - Operation()
  - createIterator()
  - size()
  - getElem(int)

- ConcreteCollection
  - addChild(Child)
  - removeChild(Child)
  - Operation()
  - createIterator()
  - size()
  - getElem(int)

- Client

- Iterator
  - first()
  - hasNext()
  - next()

- ConcreteIterator
  - first()
  - hasNext()
  - next()
// client code
Collection* b = new Book;
...
Iterator* iter = b->createIterator();

iter->first();
while ( iter->hasNext() ) {
    Page* p = iter->next();
    p->doSomething();
}
Book

Composite class is augmented with operations to support the Iterator Pattern.

```cpp
class BookIterator;

class Book {
   public:
      void addPage(Page*);
      Page* getPage(int) const;
      int size() const;
      BookIterator* createIterator();

   private:
      std::vector<Page*> pages_;
};

BookIterator* Book::createIterator() {
   return new BookIterator(this);
}
```
class BookIterator {
    public:
        BookIterator(Book* b) : book_{b}, cursor_{0} {} 
        Page* next();
        bool hasNext() const;
        void first() { cursor_ = 0; }
    private:
        Book* book_; 
        int cursor_; 
}; 

bool BookIterator::hasNext() const { 
    return cursor_ < book_->size(); 
}

Page* BookIterator::next() { 
    if (!hasNext()) { 
        return nullptr; 
    } 
    Page* result = book_->getPage(cursor_); 
    cursor_++; 
    return result; 
}
// client code
Collection* b = new Book;
...
Iterator* iter = b->createBackIterator();

iter->first();
while ( iter->hasNext() ) {
    Page* p = iter->next();
    p->doSomething();
}

U Waterloo CS247 (Spring 2017) — p.9/22
Book (Revisited)

Composite class is augmented with operations to support the Iterator Pattern.

```cpp
class BackBookIterator;

class Book {
    public:
        void addPage(Page*);
        Page* getPage(int) const;
        int size() const;
        BookIterator* createIterator();
        BackBookIterator* createBackIterator();

    private:
        std::vector<Page*> pages_;
};

BackBookIterator* Book::createBackIterator() {
    return new BackBookIterator(this);
}
```
class BackBookIterator {

public:
    BackBookIterator(Book* b) : book_{b}, cursor_{book_->size()-1} {}
    Page* next();
    bool hasNext() const;
    void first() { cursor_ = book_->size()-1; }

private:
    Book* book_;  
    int cursor_; 

};

bool BackBookIterator::hasNext() const {
    return cursor_ <= 0;
}

Page* BackBookIterator::next() {
    if (!hasNext()) {
        return nullptr;
    }
    Page* result = book_->getPage(cursor_);
    cursor_--;  
    return result;
}
Iteration over a Composite Object

The more interesting case is when the aggregate is a **composite object**, in which case we need to construct an Iterator that understands and navigates the composite.
Composite Iteration

- **TeamMember**
  - name()
  - salary()
  - add (TeamMember)
  - remove (TeamMember)
  - createlIterator()
  - size()
  - getChild(int)

- **Team**
  - add (TeamMember)
  - remove (TeamMember)
  - createlIterator()
  - size()
  - getChild(int)

- **Hero**
  - salary()
  - createlIterator()

- **Iterator**
  - first()
  - hasNext()
  - next()

- **TeamIterator**
  - TeamIterator(Team)
  - first()
  - hasNext()
  - next()

- **HeroIterator**
  - HeroIterator(Hero)
  - first()
  - hasNext()
  - next()

- **collection**

- **Client**

- **<<create>>**
Client Code

Iterate through all members in the composite.

TeamMember* employees = new Team (... Iterator* iter = employees->createIterator(); // Team Iterator
iter->first();
while ( iter->hasNext() ) { TeamMember* m = iter->next(); m->doSomething(); }

Iterate through all members in a leaf (not very interesting).

TeamMember* employees = new Developer (... Iterator* iter = employees->createIterator(); // Dev Iterator
iter->first();
while ( iter->hasNext() ) { TeamMember* m = iter->next(); m->doSomething(); }
Create Iterator

Each concrete subclass in the composite knows how to create its own corresponding Iterator.

```cpp
Iterator* Developer::createIterator() {
    return new DevIterator(this);
}

Iterator* Team::createIterator() {
    return new TeamIterator(this);
}
```
Developer Iterator

class HeroIterator : public Iterator {
private:
    Hero* hero_;  // Hero to iterate over
    Hero* cursor_;  // Current element
public:
    HeroIterator(Hero* hero);  // Constructor
    virtual void first() override;  // Set the iterator to the beginning
    virtual bool hasNext() override;  // Check if there are more elements
    virtual TeamMember* next() override;  // Get the next element
};

HeroIterator::HeroIterator(Hero* hero) :
    hero_{hero}, cursor_{hero} {}
void HeroIterator::first() { cursor_ = dev_; }
bool HeroIterator::hasNext() { return cursor_ != nullptr; }
TeamMember* HeroIterator::next() {
    if ( !hasNext() ) return nullptr;
    cursor = nullptr;
    return hero_;
The Composite objects contribute to iteration with operations to retrieve \textit{child elements}.

```cpp
class Team : public TeamMember {
    private:
        std::vector<TeamMember*> members_;  
    public:
        ...

        virtual void add( TeamMember* newMember ) {
            members_.push_back( newMember );
        }

        virtual int size() const { return members_.size(); }
        virtual TeamMember* getChild(int i) const {
            return members_.at(i);  
        }
};
```
Team Iterator

Each composite node maintains a collection of child nodes. As the composite iterator walks through the tree, it:

- keeps an iterator (cursor) for each collection node along partially searched path, and
- puts iterators on stack as the nodes are encountered (depth-first search).

class TeamIterator : public Iterator {
private:
  TeamMember* members_; // pointer to composite
  struct IterNode; // < node*, cursor>
  std::stack<IterNode*> istack; // stack of iterators

public:
  TeamIterator(TeamMember* m) : members_{m} { first(); }

  virtual void first(); // initialize Iterator stack
  virtual bool hasNext();
  virtual TeamMember* next();
};
TeamIterator::first()

```cpp
struct TeamIterator::IterNode {
    TeamMember* node_;  
    int cursor_;  // ranges from -1 .. collect_->size()

    IterNode(TeamMember* m) : node_{m}, cursor_{-1} {} 
};

Initialize the iterator stack with a cursor for the whole composite.

void TeamIterator::first() {
    while ( !istack.empty() ) { 
        delete istack.top();  
        istack.pop(); 
    }

    istack.emplace( new IterNode( members_ ) );
}
```
TeamIterator::hasNext()

Check if stack contains an iterator that has not retrieved all children of its respective node.

```cpp
bool TeamIterator::hasNext() {
    while ( !istack.empty() ) {
        IterNode *top = istack.top();
        if ( top->cursor_ < top->node_->size() ) {
            return true;
        }
        istack.pop();
        delete top;
    }

    return false;
}
```
TeamIterator::next()

TeamMember* TeamIterator::next() {  // preorder iteration
  if ( ! hasNext() ) {  // have cursors reached their limit?
    return nullptr;
  }
  IterNode* top = istack.top();
  istack.pop();

  // if cursor == -1, advance cursor and return node
  if (top->cursor_ == -1) {
    top->cursor_ += 1;
    istack.emplace(top);  // advance cursor to first child
    return top->node_;  // return node
  }

  // else, advance cursor AND create new cursor for child
  TeamMember *elem = top->node_->getChild(top->cursor_);
  top->cursor_ += 1;
  istack.emplace(top);  // advance cursor to next child
  istack.emplace(new IterNode(elem));  // push new cursor
  return next();  // recurse
The goal of design patterns is to **encapsulate change**.

**Composite Pattern:** encapsulates the structure of a heterogeneous, possibly recursive data structure.

**Iterator Pattern:** encapsulates the **iteration** of a heterogeneous, possibly recursive data structure.