Overview

1. Liskov Substitutability Principle
2. Law of Demeter
3. Iterator Design Pattern
4. Composition
The principles covered in class were:

- Open Closed Principle
- Favour Composition over Inheritance
- Single Responsibility Principle
- Liskov Substitutability Principle
- Law of Demeter
What’s Liskov Substitutability Principle?
Liskov Substitutability Principle

A derived class must be substitutable for its base class.
Liskov Substitutability Principle

- Objects accept the base class’s messages.
- Methods require no more than base class methods.
- Methods promise no less than base class methods.
Q: Is Square substitutable for Rectangle?

Example: lsp1.cc lsp2.cc lsp3.cc
source: Robert Martin, Agile Principles, Patterns, and Practices in C#
Law of Demeter

Also known as the principle of least knowledge.

Method A::m1 can only call methods of
- A itself
- A’s data members
- m1’s parameters
- any object constructed by A’s methods
Law of Demeter Violations

class Foo {
public:
    void example(Bar b) {
        C & c = b.getC();

        c.doIt();

        b.getC().doIt();

        D d{};

        d.doSomethingElse();
    }
};
The most basic iterators should, at the very least, have the following functions:

MyObject* next()
bool hasNext()
void first()
next() will return a pointer to the current element in the container, then move the iterator to the next element. hasNext() will check whether or not there are any more elements left over which to iterate. first() doesn’t return a pointer, but simply “resets” the pointer to point at the first element.
Collection* c = new Collection;
Iterator* iter = c->createIterator();
iter->first();
// reset the iterator; could be done in the constructor
while (iter->hasNext()) {
    Elem* e = iter->next();
    e->doSomething();
}
class MyArray {
public:
    MyArray();
    void addElem(Elem*);
    Elem* getElem(int) const;
    int getSize() const;
    int getSpace() const;
    ArrayIterator* createIterator();
private:
    int space;
    int size;
    Elem* array[space];
};
ArrayIterator

class ArrayIterator {
public:
    ArrayIterator(MyArray* a): array{a}, index{0} {}
    void first();
    bool hasNext();
    Elem* next();

private:
    int index;
    MyArray* array;
};
Because a tree data structure is more complex than an array or a linked list, iterating through it is non-trivial. One important element is the stack, which keeps track of a "path" from the root of the tree to the current node (this works very similar to the depth-first search algorithm). This lets the iterator recursively explore each child of a node in sequence.
For a leaf node, the iterator doesn’t have to do much work.

`first()` sets the cursor to the leaf

`hasNext()` simply checks if the cursor is `nullptr`

`next()` returns the item if the cursor isn’t `nullptr`, and resets cursor to `nullptr`

example code: /iter/fileiterator.cc
The iterator for our internal nodes is more complicated.

```c
void first()
{
    clearStack();
    stack.push(root);
}
```

This ensures our iterator is ”reset” between successive iterations.
bool hasNext() {
    while (!stack.empty()) {
        if (stack.top().cursor < stack.top().size) {
            return true;
        }
        stack.pop();
    }
    return false;
}

Note that we keep popping nodes from the stack until we find one that still has unexplored elements.
Composite Iterator Functions

BaseClass* next() {
    if ( hasNext() ) {
        if (stack.top().cursor == -1) {
            stack.top().cursor++;
            return stack.top().data;
        }
        stack.push(stack.top().data.getChild(cursor++))
        // pushed with a cursor of -1
        next();
    }
}

If we haven’t started exploring the top node, then we can return that node and advance the cursor.
If we have partially explored the top node, then explore the next child and advance the cursor.
The Composite Design Pattern can be used to make tree-like, recursive structures. The client can iterate through the tree without having to know, in advance, if they are looking at an inner node or a leaf node.
End