Overview

1. Iterator Design Pattern
2. Composition
3. Decorator
4. Factory
The idea behind iterators is that, no matter what container you’re using, you should be able to go through the container and do something to each element in the container, one at a time. An iterator is a pointer to the “current” element.
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();
}
Because the client has to do relatively little work with an iterator, that means we (as the creators of whichever container is being used) have to do more work ourselves. This can vary in complexity depending on the container we are creating.
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];
};
class ArrayIterator {
public:
    ArrayIterator(MyArray* a): array{a}, index{0} {} 
    void first();
    bool hasNext();
    Elem* next();
private:
    int index;
    MyArray* array;
};
Composite is a fairly overloaded word in CS. So far, we have seen:

- The UML relationship (e.g. a Deck is composed of Cards)
- The concept of simulating Inheritance through ownership of a base class.
- The recursive Design Pattern
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.
Composite Iterator Functions

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.

```cpp
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.
BaseClass* next() {
    if ( hasNext() ) {
        if (stack.top().cursor == -1) {
            stack.top().cursor++;
            return stack.top().data;
        }
        stack.push(stack.top().data.getChild(cursor++));
        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. It allows us to create a unified interface for our client code to work with an inner node or a leaf node without needing to know which it is.
A filesystem consists of two categories of objects: files and folders. A file is a leaf node (and can be a text file, image, executable, etc.), and a folder can contain any number of other objects (files and folders). Many operations such as `chmod` can be called on files and directories interchangeably. It is up to the filesystem code to properly implement functions such as `chmod` such that they do something appropriate when dealing with a folder versus a single file.
Decorator

When:
The Decorator pattern is used when you want to add “extra” features on top of a particular base class without creating subclasses.

Examples:
adding toppings to a pizza, putting on different combinations of clothing, customizing your car, add-ons for your internet browser, etc.
Why Decorate?

The decorator pattern lets us do two things:

- Create arbitrarily many combinations without having a subclass for each one
- Swap out decorations at run-time
Subclassing works well for very few combinations of decorations, but is not very extensible. Every time a new concrete decoration is added, the number of possible subclasses doubles.
Question:
Add toppings to a pizza and compute price for the pizza
A few toppings: Pepperoni GreenPeppers
Subclassing

Question:
Add toppings to a pizza and compute price for the pizza
A few toppings: Pepperoni GreenPeppers Mushrooms
The Decorator Pattern is set up like a linked-list. Each decoration, like a linked-list node, has some behaviour of its own, and a pointer to the next object - which could be another decorator, or the base object.
Because of this setup, functions tend to be recursive. A decorator’s “doSomething()” method will typically do a bit of the work, and then call the contained object’s “doSomething()”. The contained object will then do the same thing - do a bit of the work, then let the function call propagates down until it eventually gets to the original object that’s being decorated.
Decorator Chain Example

```java
Pizza* newPizza = new Pizza();
newPizza = new mushrooms(newPizza);
newPizza = new pepperoni(newPizza);
newPizza = new greenPeppers(newPizza);
```
newPizza->cost() =
    COST_OF_GREENPEPPERS +
    component()->cost()...

newPizza->cost() =
    COST_OF_GREENPEPPERS +
    COST_OF_PEPPERONI +
    component()->cost()...
newPizza->cost() =
    COST_OF_GREENPEPPERS +
    COST_OF_PEPPERONI +
    COST_OF_MUSHROOMS +
    component()->cost()...

newPizza->cost() =
    COST_OF_GREENPEPPERS +
    COST_OF_PEPPERONI +
    COST_OF_MUSHROOMS +
    COST_OF_PIZZA
A Factory is an object that makes other objects. With a factory, you can use:

- Encapsulation: client code is not directly tied to specific classes, so classes can be changed, added, or refactored without changing client code.

- Polymorphism: delegating the creation of an object to the right factory means that we can decide what kind of object to create at runtime.
A simple factory’s “create” method can look like this:

```cpp
Enemy* Factory::createEnemy(Difficulty difficulty){
    if (difficulty == BEGINNER)
        return new LittleDude();
    if (difficulty == STANDARD)
        return new BigDude();
    if (difficulty == EXPERT)
        return new WhatIsThatThing();
}
```

This design allows us to change what enemy corresponds with what difficulty as the design of the game changes.
The previous example is simple, practical, and commonly used, but does not take advantage of Polymorphism. The Factory Method pattern does this by having a separate factory for each subclass, then using a pointer to a base factory class. This allows us to use Polymorphism, swapping factories at runtime as necessary.
The same factory from before can be refactored using the Factory Method pattern like so:

```cpp
Enemy* EnemyFactory::createEnemy { return create(); }  
Enemy* EnemyFactory::create() = 0;  
Enemy* LittleDude::create() { return new LittleDude; }  
Enemy* BigDude::create() { return new BigDude; }
```

Note that this is also an example of the Template Method pattern. The `createEnemy` function is not abstract, but the private `create()` method is abstract.
Abstract Factory

After finding a good way to produce a single class, the next logical step is to create a factory that produces multiple classes. We can create an Abstract Factory that can create multiple types of abstract products, then each Concrete Factory can define which concrete products it wants to produce. This lets each Concrete Factory represent a family of related concrete objects.
1. Now we still want to generate enemies (WeakEnemy or StrongEnemy) based on difficulty levels (BEGINNER or EXPERT).
2. For each type of enemies, we need two types of spawners (FireLevelSpawner and IceLevelSpawner).
3. Fire Elemental and Ice Elemental are for WeakEnemy.
4. Dragon and Ice Golem are for StrongEnemy.
Enemy* EnemySpawner::createEnemy(Difficulty difficulty) {
    if (difficulty == BEGINNER)
        return createWeak();
    if (difficulty == EXPERT)
        return createStrong();
}
Abstract Factory Example

- **EnemySpawner**
  - `createWeak()`: WeakEnemy
  - `createStrong()`: StrongEnemy

- **FireLevelSpawner**
  - `createWeak()`: WeakEnemy
  - `createStrong()`: StrongEnemy
  - **Fire Elemental**
  - **WeakEnemy**

- **IceLevelSpawner**
  - `createWeak()`: WeakEnemy
  - `createStrong()`: StrongEnemy
  - **Ice Elemental**
  - **Dragon**
  - **Ice Golem**
  - **StrongEnemy**
End